Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Факультет інтелектуальних інформаційних технологій та автоматики
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Бакалаврська дипломна робота на тему:

«КОМП'ЮТЕРНА ГРА ГОМОКУ З БАГАТЬМА РІВНЯМИ СКЛАДНОСТІ»

Виконав: студент 4 курсу, групи 2КН-20б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Парванчук Д. С.
(прізвище та ініціали)
Керівник: асистент кафедри КН
Малініч І. П.
(прізвище та ініціали)
« 07 » 06 2024 р.
Рецензент: к.т.н., доцент каф. АІІТ
Кулик Я. А.
(прізвище та ініціали)
« 07 » 06 2024 р.

Допущено до захисту
Зав. кафедри Яровий А.А.
« 07 » 06 2024 р.

Вінниця ВНТУ – 2024 рік

Вінницький національний технічний університет

Факультет інтелектуальних інформаційних технологій та автоматизації

Кафедра Комп'ютерних наук

Рівень вищої освіти – перший (бакалаврський)

Галузь знань – 12 «Інформаційні технології»

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма – «Комп'ютерні науки»

ЗАТВЕРДЖУЮ

Завідувач кафедри КН

проф., д.т.н. Яровий А. А.

«07» 03 2024р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Парванчуку Денису Сергійовичу 1.

Тема роботи: Комп'ютерна гра Гомоку з багатьма рівнями складності керівник роботи: Малініч І.П. асистент каф. КН

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу "11" 05 2024 року №80

2. Термін подання студентом роботи: 27.05.2024

3. Вихідні дані до роботи: Кількість гравців: 1; Кількість складностей гри: 3; час відгуку не повинен перевищувати 500 мілісекунд; Ігрове поле – дошка го 15x15 4. Зміст текстової частини (перелік питань, які потрібно розкрити):Дослідження предметної області. Визначення інструментів програмної реалізації. Розробка алгоритму для гри Гомоку. Розробка графічної частини. Аналіз результатів тестування. Розробка інструкції користувача.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень): Блок-схеми алгоритмів, основні види комбінацій, графічний інтерфейс програми, результати роботи програми.

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завда прий
1, 2, 3	Малініч І.П., асистент каф. КН		

7. Дата видачі завдання 07.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання		Прим
		початок	закінчення	
1	Обгрунтування доцільності створення	06.05.24		09.05
2	Аналіз існуючих рішень	10.05.24		19.05
3	Проектування гри Гомоку	20.05.24		24.0
4	Реалізація гри Гомоку	25.05.24		31.05
5	Тестування гри Гомоку	01.06.24		06.06

Студент _____ Парванчук Д.С
(підпис) *(прізвище та ініціали*

Керівник роботи _____ Малініч І.П
(підпис) *(прізвище та ініціали*

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 75 сторінок формату А4, на яких є 34 рисунки, список використаних джерел містить 19 найменувань.

В даній бакалаврській роботі розроблено алгоритм для прийняття рішень в грі гомоку, покращено графічний інтерфейс та додано різні складності гри. Досліджено способи вирішення даних задач. Запропоновано використовувати підхід, що ґрунтується на застосуванні Python та Tkinter. Результатом дипломного дослідження є програмний додаток гра гомоку, що реалізована на мові програмування Python.

Ключові слова: гомоку, рендзю, алгоритм

ABSTRACT

The bachelor's thesis consists of 75 pages of A4 format, which contains 34 figures, the list of used sources contains 19 items.

In this bachelor's work, an algorithm for decision-making in the game of gomoku was developed, the graphical interface was improved, and various game difficulties were added. Ways of solving these problems have been studied. An approach based on the use of Python and Tkinter is suggested. The result of the diploma research is a software application for the game of homoku, implemented in the Python programming language.

Keywords: gomoku, renju, algorithm

ЗМІСТ

ВСТУП

Актуальність досліджень. В даний час різні комп'ютерні програми та ігри користуються великим попитом і досить популярні, що дуже вигідно для їх створення і реалізація.

Логічні ігри користуються особливим попитом адже надають здатність вирішувати проблеми та робити правильний вибір який заснований на характеристиках логічного мислення, яке описується як здатність виявляти зв’язки між інформацією у навколишньому середовищі, виводити правила, концепції та моделі поведінки.

Логічна ігри – це особлива форма гри, яка включає міркування та роздуми. Ці ігри зазвичай грунтуються на математиці або логіці. Попри те, що думає більшість людей, інтелект не є вродженою та незмінною рисою. Мозок, як і будь-який інший м’яз, тренується. Логічні ігри безперечно допомагають виконати цю роботу. Такі ігри допомагають розвивати логічне мислення та інтелект як для дорослих так і для дітей, а програмні додатки на смартфонах або комп’ютерах дозволяють робити це будь-де.

Гомоку стародавня японська гра на дошці посіченій ножем, назва гри походить від японського слова ґомокунарабе, що означає «п'ять штук в ряд». Ця гра в особливості допоможе розвинути пам’ять та інтелект, адже потребує запам’ятовування різних комбінацій та розуміння коли і де їх використовувати.

Метою дослідження є розширення функціональних можливостей та покращення графічного інтерфейсу.

Задачі дослідження:

1) Аналізувати предметну область та визначити актуальність дослідження.

2) Провести аналіз існуючих рішень комп’ютерної гри «Гомоку».

3) Розробити архітектуру програмного модуля комп’ютерної гри.

4) Розробити алгоритм функціонування програмного модулю комп’ютерної гри.

5) Здійснити програмну реалізацію модулю комп’ютерної гри.

6) Протестувати програмний модуль комп’ютерної гри.

7) Розробити інструкцію користувача програмного модулю комп’ютерної гри.

Об’єктом дослідження є процес вдосконалення графічного інтерфейсу.

Предметом дослідження є алгоритми та програмні засоби для гри Гомоку

Практична цінність. Розробка гри Гомоку з багатьма рівнями складності має значну практичну цінність, як з точки зору освітніх, так і з точки зору розважальних аспектів. Вона дозволяє глибше вивчити програмні технології, розробити алгоритми штучного інтелекту та створити захопливу гру з різними рівнями складності.

В ході виконання Бакалаврської Дипломної роботи був розроблений алгоритм який має значну практичну цінність оскільки він має багато шляхів вдосконалення для відкриття нових варіантів різних логічних ігор

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ІГОР ГОМОКУ

1.1 Аналіз сучасного рішення розробки комп'ютерної гри «Гомоку»

Логічні ігри – це ігри, які потребують не лише знань і навичок у певних галузях, а логічного та винахідливого підходу. Логічні ігри часто називають головоломка, адже кожна головоломка створювалась для того, щоб користувач, гравець, іншими словами особа використовувала не лише знання, які вони отримали, а логіку, яку вона повинна розвивати.

Головоломки навчають нас аналізувати ситуації, виявляти закономірності, робити висновки, розвивати абстрактне мислення. Вони також стимулюють креативність, адже рішення головоломок часто вимагає нестандартного підходу, що спонукає нас шукати нестандартні рішення. Логічні ігри тренують навички вирішення проблем, адже вони вчать розбивати завдання на частини, визначати ключові елементи і знаходити оптимальні шляхи до вирішення.

Крім того, логічні ігри мають позитивний емоційний вплив. Вони можуть розслаблювати, допомагаючи зосередитися на завданні та відволіктися від повсякденних проблем. Після вирішення складної головоломки гравець відчуває почуття досягнення, що підвищує його самооцінку та мотивує до подальшого розвитку.

Логічні ігри також мають значну суспільну користь. Вони можуть використовуватися в освітніх програмах для розвитку навичок вирішення проблем, стимулювання інтересу до математики, логіки та програмування. Багато логічних ігор можна грати з іншими людьми, що сприяє командній роботі, спілкуванню та соціальній інтеграції.

Логічні ігри зміцнюють концентрацію уваги, адже вони вимагають зосередження. Деякі головоломки, наприклад судоку, тренують пам'ять, оскільки вимагають запам'ятовування інформації та її використання для вирішення завдання. Ігри, що базуються на знаннях, наприклад кросворди, розширюють словниковий запас, ознайомлюють з новими фактами та ідеями.

Створення гри Гомоку має доцільність з кількох причин, які можна розділити на освітній, розважальний та технологічний аспекти.

З освітньої точки зору, Гомоку, як і рендзю, є грою, яка вимагає від гравців логічного мислення, планування та передбачення ходів суперника. Гра сприяє розвитку стратегічного мислення, вмінню аналізувати ситуації та вибирати оптимальні рішення. Гомоку також заснована на комбінаториці, тобто в ній гравці мають вміти розпізнавати та використовувати комбінації ходів, що сприяє розумінню комбінаторних принципів. Гомоку може бути використано в освітніх програмах для розвитку логічного мислення у дітей та підлітків. Вивчаючи правила та стратегії гри, діти розвивають навичку аналізу, планування та прийняття рішень.

З розважальної точки зору, Гомоку має прості правила, що робить її доступною для гравців будь-якого віку. Однак за простотою правил ховається глибока стратегія та велика кількість різноманітних комбінацій. Гомоку можна грати як з друзями на папері, так і онлайн, що робить її доступною для великої аудиторії. Вона може бути як розважальною, так і конкурентною грою, що додає до неї цікавості та мотивації для розвитку навичок.

З технологічної точки зору, створення гри Гомоку може стати відмінною навчальною платформою для розробки штучного інтелекту. Розробка алгоритму, що може грати в Гомоку на високому рівні, вимагає застосування таких методів, як алгоритми пошуку, машинне навчання та нейронні мережі. Створення гри Гомоку дозволить розробити інтерактивну гру з привабливим інтерфейсом та різноманітними режимами гри, що зробить її більш цікавою та доступною для широкої аудиторії[1].

Таким чином, створення гри Гомоку має доцільність з освітньої, розважальної та технологічної точок зору. Вона може стати чудовим інструментом для розвитку логічного мислення, а також привабливою та конкурентною грою.

У головоломці гравець повинен складати фігури логічним чином, щоб дійти до правильного або цікавого рішення головоломки. Існують різні жанри головоломок, такі як:

- Кросворди – це головоломка, що є переплетення рядів клітин, які заповнюються словами по заданим значеням(рис 1.1).

Зазвичай значення слів задаються описово під цією фігурою, спочатку значення слів, які мають вийти горизонталлю, потім — по вертикалі[2].

Рисунок 1.1 – Загальний вигляд комп’ютерної гри у жанрі кросворди

- Головоломки для пошуку слів – це головоломка, що є прямокутною таблицею з букв, у якій шукаються слова — вертикально, горизонтально і з діагоналі, у прямому і зворотному порядку. У деяких варіантах відразу дано список захованих слів, в інших він відсутній - навпаки, загадане слово, яке утворюється з літер, що не увійшли до жодного слова цієї головоломки. Найчастіше головоломки присвячені якійсь темі та використовуються при навчанні нових слів та правопису(рис 1.2)[3].

Рисунок 1.2 – Загальний вигляд комп’ютерної гри у жанрі головоломки з пошуком слів

- головоломки з числами наприклад гра 2048 – це гра-головоломка для одного гравця з ковзаючими плитками в якій потрібно об’єднувати плитки з однаковими числами, щоб отримувати загальну сума. Мета об’єднати в число 2048 (рис 1.3).

Рисунок 1.3 – Загальний вигляд комп’ютерної гри у жанрі головоломки з числами

- головоломка – це гра метою якою є вирішення логічних завдань, що вимагають від гравця задіяння логіки, стратегі, інтуіції, та іноді ерудиції й уважності. Головоломки можуть включатися до ігор інших жанрі як

ключові елементи ігрового процесу або ж для його урізноманітнення як міні-ігри (рис 1.4)[4].

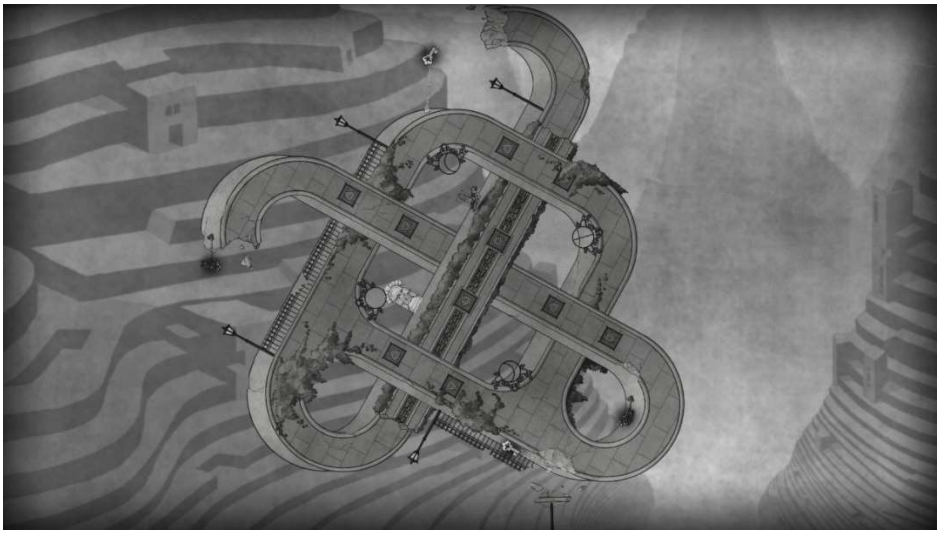


Рисунок 1.4 – Загальний вигляд комп’ютерної гри у жанрі відеогра з головоломками

Головоломки з візерунками - це захоплюючий світ, де ви можете продемонструвати свої просторові навички, логічне мислення та творчість. Вони засновані на маніпуляції з формами, розмірами та розташуванням елементів, щоб створити нові візерунки або розв’язати завдання (рис 1.5)[5].

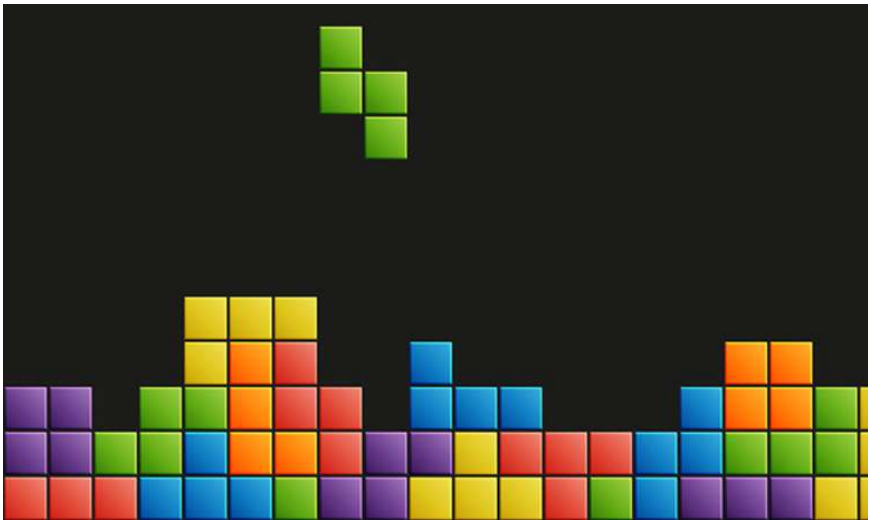


Рисунок 1.5 – Загальний вигляд комп’ютерної гри у жанрі головоломки з візерунками

Головоломки часто створюються як форма розваги, але вони також можуть виникати внаслідок серйозних математичних або логічних задач. У таких випадках їх вирішення може суттєво сприяти математичним дослідженням.

Існує чимало спеціалізованих жаргонів, що використовуються у зв'язку з проблемами шахів; дивіться глосарій шахових задач для списку. Термін "шахова проблема" чітко не визначений: немає чіткого розмежування між шаховими композиціями з одного боку, головоломками чи тактичними вправами, з іншого. Однак на практиці розрізнення є дуже чітким.

Є загальні характеристики, якими поділяються композиції в проблемному розділі шахових журналів, у спеціалізованих журналах із шахових завдань та у збірниках шахових завдань у формі книги. Не кожна шахова проблема має кожну з цих особливостей, але більшість із них має кілька:

- Позиція складена - тобто вона не взята з реальної гри, а винайдена з конкретною метою вирішення проблеми. Хоча обмеження для православних шахових проблем полягає в тому, що вихідна позиція може бути досягнута через низку законних ходів з вихідної позиції, більшість проблемних позицій не виникають у позаграшній грі[6].
- Існує конкретна умова , тобто мета, якої потрібно досягти; наприклад, поставити мат Чорному за певну кількість ходів.
- Існує тема (або поєднання тем), яку проблема склала для ілюстрації: шахові задачі, як правило, створюють конкретні ідеї.
- Проблема демонструє економію при її побудові: не застосовується більша сила, ніж та, що потрібна для того, щоб надати проблемі звук (тобто гарантувати, що передбачуване рішення проблеми справді є рішенням і що це єдине рішення проблеми).

- Проблема має естетичну цінність . Проблеми сприймаються не лише як головоломки, але і як предмети краси. Це тісно пов’язано з тим, що проблеми організовуються для демонстрації чітких ідей якомога економічніше.

Проблемам можна протиставити тактичні головоломки(Рис. 1.6), які часто зустрічаються в шахових колонках або журналах, в яких завдання полягає в тому, щоб знайти найкращий хід або послідовність ходів (як правило, що призводять до спаровування чи виграшу матеріалу) з певної позиції. Такі головоломки часто взяті з реальних ігор або, принаймні, мають позиції, які виглядають так, ніби вони могли виникнути під час гри, і використовуються для навчальних цілей. Більшість таких головоломок не демонструють вищезазначених особливостей.

Рисунок 1.6 – Загальний вигляд комп’ютерної гри у жанрі тактичні головоломки

Позиція в теорії рендзю розглядається як набір лінійних конструкцій, які взаємоперетинаються. Найбільше значення мають трійки та четвірки, про які йшлося раніше[7].

Для виграшу в партії гравець повинен прагнути до побудови своїх відкритих трійок та четвірок, одночасно блокуючи спроби створення їх суперником. Очевидно, що якщо гравець буде кожним своїм ходом будувати лише одну трійку або прикриту четвірку, то суперник відповідатиме блокуванням споруджуваного ряду. Тому для виграшу необхідно побудувати вилку — одним ходом створити більш як одну трійку або четвірку. У такому разі супротивнику не вистачить ходів на блокування і, якщо він не зможе створити форсований мат, то неминуче програє.

Найбільш вірний (але не завжди можливий) спосіб досягнення перемоги — мат серією шахів. Гравець робить ходи, кожний з яких будує четвірку, суперник вимушено блокує побудови. При цьому будується база для вилки, яка і формується передостаннім ходом. Захиститися від такої комбінації неможливо, тому в будь-якій позиції гравець завжди спочатку шукає шлях до виграшу серією шахів, і лише переконавшись, що його немає, застосовує інші варіанти.

Більше зусиль вимагає виграш за допомогою трійок (напівшахів), коли створюються не четвірки, а трійки. На відміну від попереднього випадку, у суперника залишається можливість зробити один додатковий хід, перш ніж гравець добудує переможну п'ятірку. Суперник може використовувати цей факт для створення контргри — постановки шаха і успішного захисту або навіть перехоплення ініціативи[8].

Ще один тактичний елемент, який відрізняє рендзю від хрестиків-нуликів — використання фолів. Оскільки для чорних існують заборонені ходи, то білі можуть використовувати цей факт, створюючи позиції, де чорним для захисту або переходу в атаку необхідний якраз такий хід, який будує заборонену вилку або довгий ряд. Перемога внаслідок фолу чорних називається «виграш фолом».

Для реалізації програмного додатку «Гомоку», можливо використати один алгоритм, який поступово можна вдосконалювати, відкриваючи все нові і нові варіанти гри.

1.2 Аналіз існуючих аналогів

Назва гри походить від японського слова ґомокунарабе, що означає «п'ять штук в ряд».

Історія Гомоку — це історія пошуку балансу між простотою правил та складністю стратегії. Спочатку, граючи на дошці 19×19, “Ґомоку без обмежень” (рис. 1.7) давало перевагу першому гравцеві. Пізніше правила були змінені, щоб компенсувати цю перевагу, що призвело до виникнення різних варіацій ігор.

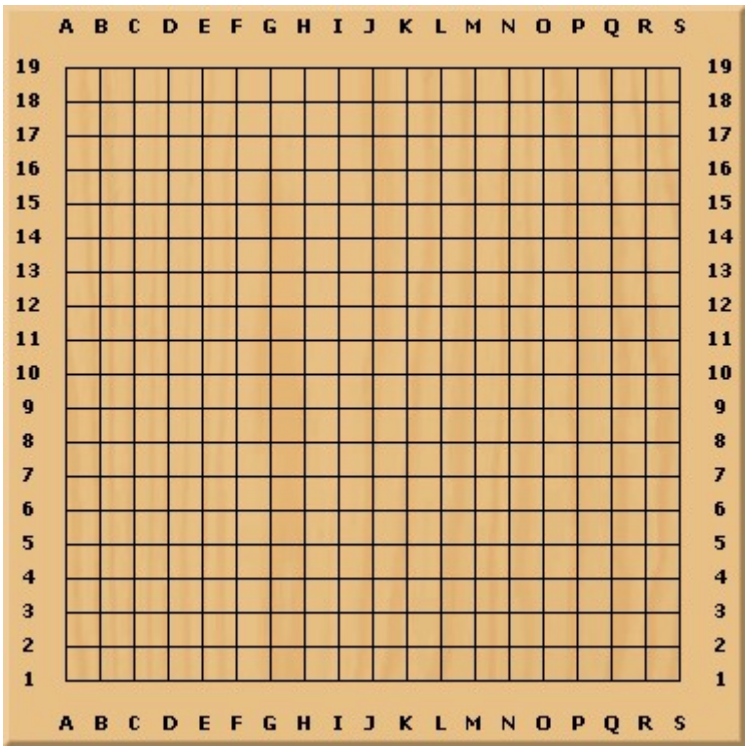


Рисунок 1.7 – Загальний вигляд комп’ютерної гри «Гомоку» без обмежень 19x19

Одним з найвідоміших варіантів є “Рендзю”, де заборонено будувати довгий ряд, а також “вилки” 3×3 та 4×4 для чорних. Цей варіант правил був введений в Японії в початку XX століття, щоб зробити гру більш збалансованою. У 1903 Японська Асоціація Професійного Рендзю ввела заборону на побудову вилок 3 × 3 для чорних, щоб компенсувати перевагу першого гравця, і обмежила розмір дошки до 15 × 15 (рис. 1.8)[9].

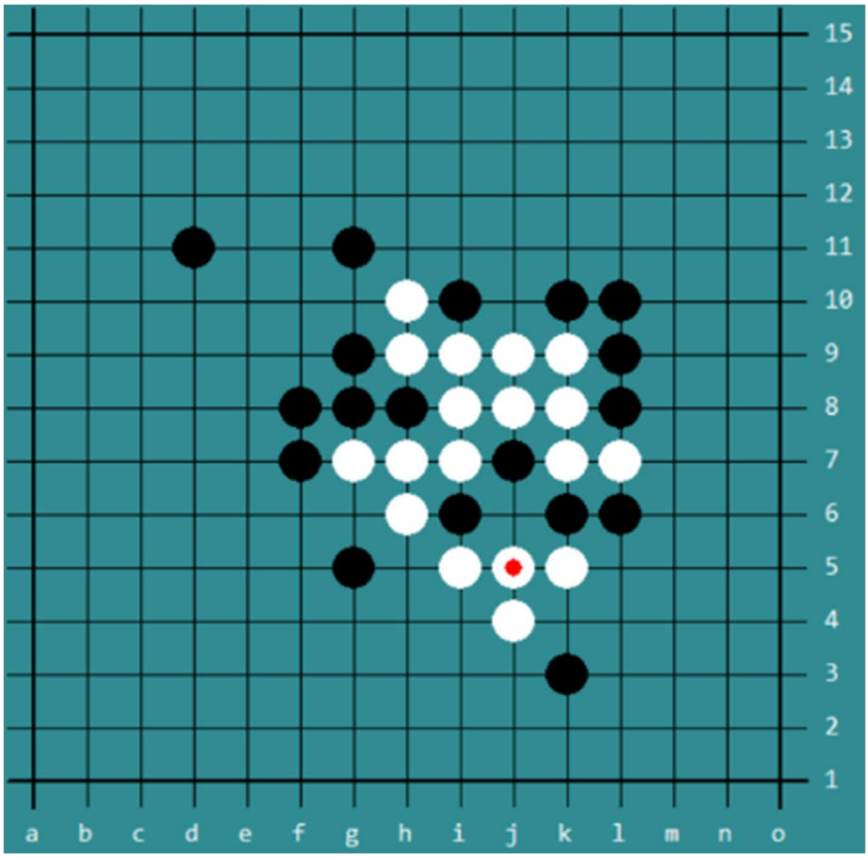


Рисунок 1.8 – Загальний вигляд комп’ютерної гри «Гомоку» 15x15

З’явилися також інші варіації, наприклад, “Вільне рендзю”, де заборонено ставити третій камінь в центральний квадрат 5×5.

Міжнародна Федерація Рендзю ввела спеціальні дебютні регламенти для перших п’яти ходів, щоб ще більше збалансувати гру та створити рівні умови для обох гравців[10].

Розвиток гри Ґомоку також не стояв на місці і йшов паралельно до розвитку рендзю. 1981 року японські професіонали рендзю Г. Саката та В. Ікава показали, що гра за правилами ґомоку на полі 15 × 15 з виграшем за допомогою побудови ряду з рівно п'яти каменів (довший ряд не виграє) завжди закінчується виграшем першого гравця за його оптимальної гри. Ґомоку, на відміну від рендзю — гра без фолів, і щоб компенсувати перевагу першого гравця застосовували інші обмеження. Так, 1980 року москвич В. Сапронов опублікував серію статей про класичний рендзю в журналі «Наука та Життя». Він, зокрема, запропонував центральний заборонений квадрат, чорним

заборонялося ставити свій третій камінь у межах центрального квадрата 5 × 5 клітин. Цей різновид правил отримав назви «сапронівка», «вільне рендзю», або, в міжнародному варіанті, «про-ґомоку» (рис. 1.9)[11].

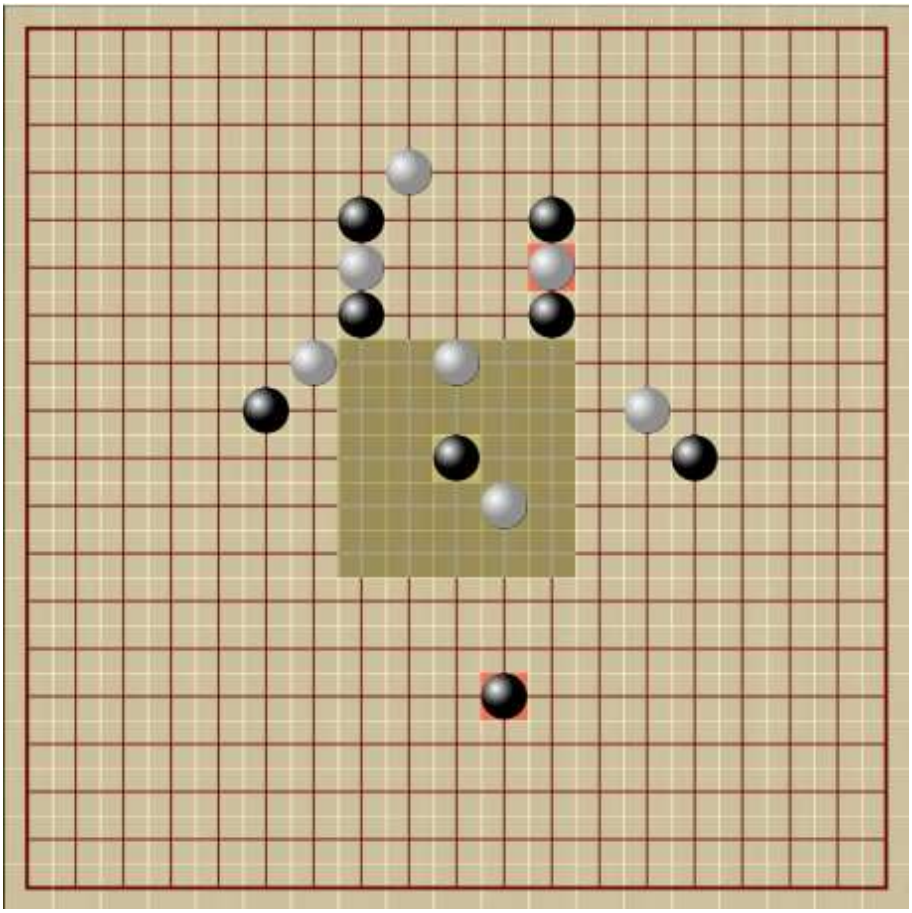


Рисунок 1.9 – Загальний вигляд комп’ютерної гри «Про-гомоку» 15x15

Існують також інші варіанти правил, наприклад, “Своп”, де гравці можуть змінити колір на початку гри, або “Своп-два”, де гравці мають три можливості для вибору кольору.

Ці різноманітні варіації правил свідчать про той факт, що Гомоку - це дуже гнучка та динамічна гра, яка продовжує еволюціонувати та залучати нових гравців.

Різновиди правил. Гомоку, як класична гра , має багато варіацій, які, хоч і зберігають основні принципи, відрізняються деталями правил та метою гри. Ці модифікації часто покликані урізноманітнити ігровий процес,

компенсувати перевагу першого гравця або просто додати новий шар складності.

- П'ять у ряд. Грається на нескінченному полі, мета - побудувати ряд з шести і більше фігур.
- Ґомоку із загальним центральним каменем. Правила аналогічні “Ґомоку”, але перший камінь в центрі дошки належить тому, хто ходить першим, що змінює баланс сил у грі. Автор — К. Коцев (Болгарія). В Україні гра поширення не набула.
- Тривимірне ґомоку. Гра ведеться в тривимірному просторі, з метою побудови ряду з п’яти фігур по діагоналі, горизонталі або вертикалі у будь-якому вимірі (рис. 1.10).

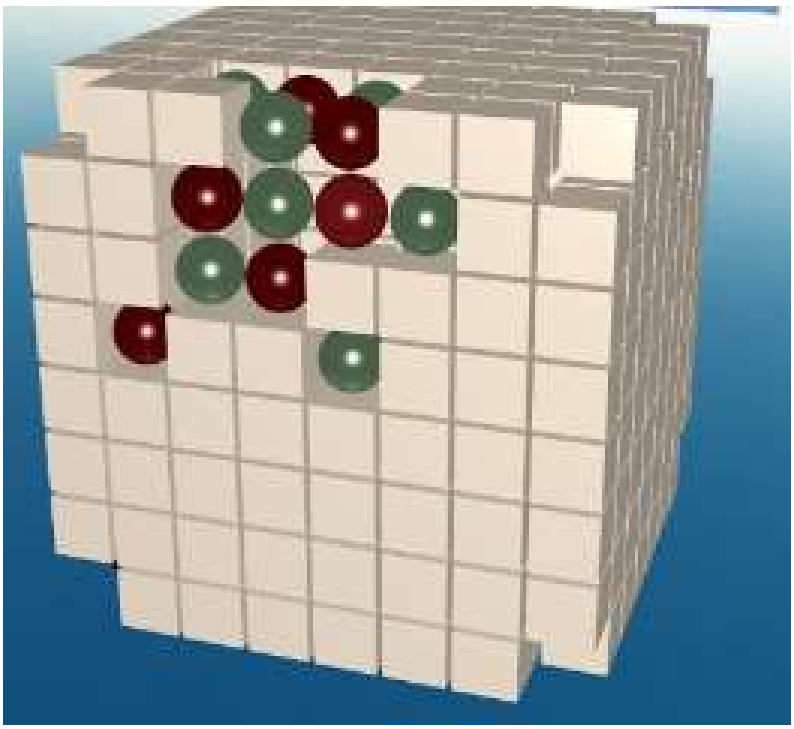


Рисунок 1.10 – Загальний вигляд комп’ютерної гри «Тривимірне гомоку» 15x15

- Ґомокунарабе. Гра ведеться в тривимірному просторі, з метою побудови ряду з п’яти фігур по діагоналі, горизонталі або вертикалі у будь-якому вимірі.

- Міжнародне ґомоку. Грається за правилами “Ґомоку” з додатковим правилом: другий гравець може змінити колір після третього ходу.
- Вільне рендзю. Гра на дошці 15×15, з забороною на хід у центральний квадрат 5×5 для другого ходу чорних (рис. 1.11).

Рисунок 1.11 – Загальний вигляд комп’ютерної гри «Гомоку» з центральним забороненим квадратом 15x15

- Керіо-пента. Гра на дошці 19×19, де ряди з двох або трьох каменів можна “знімати”, мета - побудувати ряд з п’яти або більше каменів або 15 разів забрати “здобич”.
- Каро. В’єтнамський варіант “Ґомоку”, де переможна “п’ятірка” не може бути заблокована з двох кінців.
- Ґомоку. Грається на дошці 15×15 або 19×19, мета - побудувати ряд з п’яти фігур, довші ряди не вважаються перемогою.
- Пента. Гра на дошці 19×19 з можливістю “знімати” з дошки ряди з двох каменів, закриті з двох боків камінням противника (рис. 1.12).

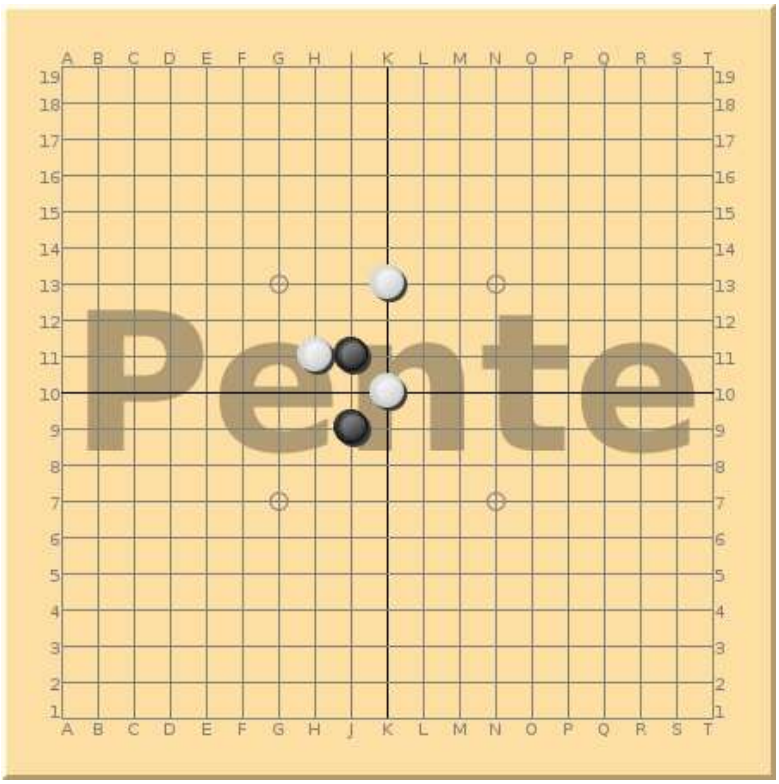


Рисунок 1.12 – Загальний вигляд комп'ютерної гри «Гомоку пента» 19x19

- Рендзю. Гра на дошці 15×15 з забороною для чорних будувати довгий ряд або "вилки" 3×3, 4×4.[12]
- Своп (від англ. Swap: міняти, обмін — означає можливий обмін кольором каменів між гравцями на початку гри). Варіант "Ґомоку", де гравці можуть змінити колір каменів на початку гри.
- Своп-два (від англ. swap — міняти, 2 — дві можливості зміни). Варіант "Ґомоку", де гравці мають три можливості для вибору кольору на початку гри (рис. 1.13).

1. вибрати чорний колір, тобто без постановки каменів передати чергу ходу першому;
2. вибрати білий колір, тобто поставити білий камінь і передати чергу ходу;
3. або ж поставити два камені, чорний та білий, надавши право остаточного вибору кольору першому гравцеві.

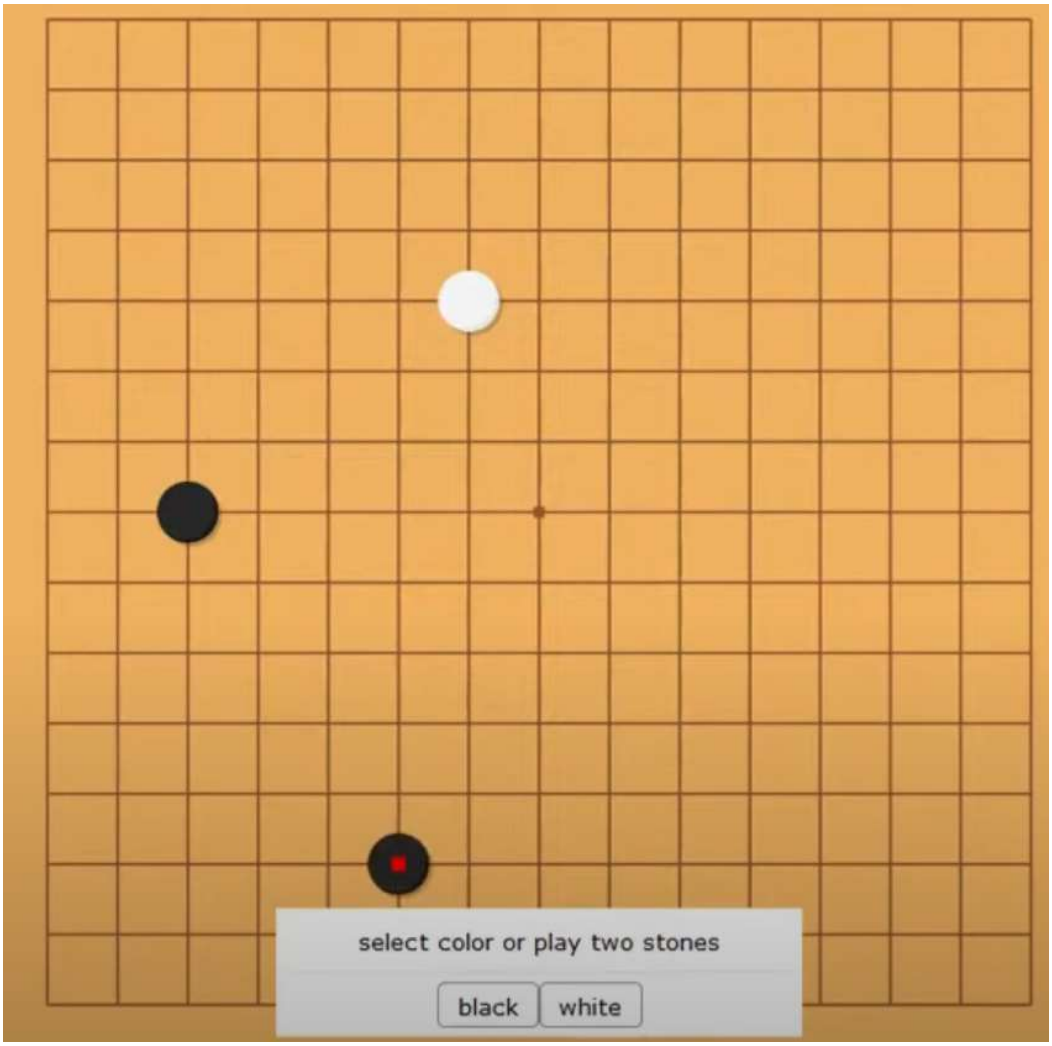


Рисунок 1.13 – Загальний вигляд комп’ютерної гри «Гомоку свап 2»

1.3 Постановка задачі дослідження

Задачею даної дипломної роботи є розробка програмного додатку гри Гомоку з різними видами складності

Метою цього дослідження є розширення функціональних можливостей та покращення графічного інтерфейсу гри Гомоку, створивши сучасну та інтуїтивно зрозумілу версію цієї класичної логічної гри. Дослідження охоплює процес вдосконалення графічного інтерфейсу, а його предметом є алгоритми та програмні засоби для гри Гомоку.

З метою досягнення поставленої мети, а саме створення гри Гомоку з різними рівнями складності, необхідно буде використати методи штучного інтелекту, такі як аналіз даних, оцінка варіантів, алгоритми пошуку та інші. Це

дозволить створити систему штучного інтелекту, яка буде здатна грати в Гомоку на різних рівнях складності.

Для успішної реалізації цього завдання буде розроблено комплекс алгоритмів, які дозволять штучному інтелекту ефективно визначати поточний стан гри, аналізувати доступні ходи та оцінювати їх на основі різних стратегій та тактик, що використовуються у грі Гомоку.

Одним з ключових компонентів системи буде використання алгоритмів пошуку, таких як мінімаксний алгоритм і алгоритм альфа-бета відсікання. Ці методи дозволять штучному інтелекту систематично оцінювати різні варіанти ходів, передбачати відповідь супротивника і вибирати оптимальні стратегії.

Оцінка потенційних ходів буде заснована на різних критеріях, включаючи їх вплив на хід гри, можливість досягнення перемоги та зменшення ризику поразки. Штучний інтелект буде намагатися побудувати модель, яка не тільки максимізує ймовірність виграшу, але й адаптується до різних сценаріїв ігрового процесу.

Такий підхід дозволить досягти різних рівнів складності в грі Гомоку, роблячи штучний інтелект відповідно до завдань відновлення, прогнозування та прийняття рішень.

Інтерфейс програми, що розробляється, повинен бути зручний і зрозумілий користувачу. Програма має реагувати на клацання мишкою, виводячи зображення на екран (рис. 1.14).

Додатково, для покращення ефективності системи штучного інтелекту будуть застосовані методи оптимізації, такі як евристики для прискорення процесу прийняття рішень. Це дозволить зменшити обчислювальні витрати при обробці великої кількості можливих варіантів ходів і збільшити швидкість реакції системи на ходи супротивника.

Рисунок 1.14 – Загальний вигляд інтерфейсного вікна комп'ютерної гри «Гомоку»

В результаті реалізації цієї системи буде створена гра Гомоку з різними рівнями складності, яка зможе грати з користувачем на рівних, демонструючи різні рівні майстерності.

Задачі дослідження:

- Обгрунтування доцільності створення. Потрібно описати інформацію про логічні ігри навести приклади цих ігор та довести доцільність створення програмного додатку
- Аналіз існуючих рішень. Проаналізувати всі можливі аналоги гри Гомоку, описати їх та навести всі недоліки та переваги цих ігор
- Проектування гри Гомоку. Створити та описати алгоритм для гри Гомоку у словесному та графічному вигляді, а також зробити те саме з UML-діаграмою класів
- Реалізація гри Гомоку. Обґрунтувати вибір інструментів технічної реалізації та описта основні класи та функції програмного коду з наведенням прикладу у вигляді скріншотів

- Тестування гри Гомоку. Перевірити правильність виконання графічного інтерфейсу та ігрових механік, що були спроектовані та розроблені під час виконання дипломної роботи

1.4 Висновок

В даному розділі було проведено аналіз предметної області. Обгрунтовано доцільність логічних ігор та гри Гомоку. Наведені приклади різних жанрів логічних ігор. Описано різновиди правил, тактики та стратегії. Проаналізовані аналоги гри та наведено їх переваги та недоліки. Розглянуто історію гри Гомоку та те як вона розвивалась з часом від початку і до теперішнього стану. Постановка задачі розробки програмного додатку гри Гомоку дозволила чітко визначити мету і завдання, які перед нами стоять.

Проведений аналіз підтверджує актуальність та перспективінсть логічних ігор та гри Гомоку, що є важливим підґрунтям для подальшої розробки власної гри в рамках дипломної роботи.

2 ПРОЕКТУВАННЯ ГРИ «ГОМОКУ»

2.1 Опис алгоритму у словесному вигляді

Розробка штучного інтелекту для гри Го - це складний і цікавий виклик. Вона вимагає комбінації ефективних алгоритмів, ретельно розробленої функції оцінки і вдосконалення за допомогою навчання на великих наборах даних. Штучний інтелект для гри Го може бути використаний для навчання гравців, створення комп'ютерних супротивників і далі досліджувати глибину і складність цієї класичної гри.

На дошці із перехрестями 15 * 15, два користувачі (один має чорні камені, а інший має білі камені) чергують один одного, кладучи один камінь їх кольору на порожньому перехресті. Поклавши, камені не рухаються і не знімаються з дошки.

Переможцем стає перший гравець, який отримав неперерваний ряд з п'яти каменів одного кольору по горизонталі, по вертикалі або по діагоналі. Зверніть увагу, що для перемоги не потрібно мати ряд з рівно п'ятьма каменями; ряд з більш ніж п'яти каменів також відповідає вимогам.

Для реалізації програмного додатку потрібно організувати два основних компоненти:

- алгоритм пошуку;
- об'єк для оцінки.

У алгоритмі пошуку потрібно встановити значення обмеженого часу коду, щоб кожен хід не був занадто довгим. Також потрібно генерувати і зберігати елементи на дошці.

В функції оцінки потрібно реалізувати:

- Створити список ваг для перехресть на дошці;
- Додати вагу 7 до центру, 6 до зовнішнього квадрату, потім 5, 4, 3, 2, 1, нарешті 0 до самого зовнішнього квадрату;
- Створити список для збереження поточного результату аналізу в рядку;
- Створити список для збереження поточних данних у рядку;

- Створити загальний список для збереження результату аналізу всієї дошки;
- Створити список для збереження кожного ходу, щоб запобігти його повторенню;
- Визначити функції для виходу з гри;
- Визначити функцію для аналізу поля в якій буде реалізовано аналіз по горизонталі, повертикалі, ліворуч по діагоналі та праворуч по діагоналі
- Створити функцію яка буде аналізувати рядок, щоб відслідкувати ходи гравців.

Також для створення програмного додатку використовувався алгоритм мінімакс з обрізкою alpha-beta.[14]

Алгоритм мінімакс – це рекурсивний алгоритм, який використовується для знаходження оптимального ходу в іграх з нульовою сумою, де дві сторони конкурують один з одним. Алгоритм обходить дерево можливих ходів, оцінюючи кожен з них, припускаючи, що суперник завжди грає оптимальним чином.

Принцип роботи:

- Алгоритм починає з поточного стану гри.
- Алгоритм рекурсивно переходить до кожного можливого ходу.
- Для кожного кінцевого стану гри алгоритм обчислює значення оцінки(наприклад, +1 для перемоги, -1 для програшу, 0 для нічиєї).
- Якщо гравець максимілізатор, він обирає хід з максимальним значення оцінки.
- Якщо гравець мінімізатор, він обирає хід з мінімальним значенням оцінки.
- Значення оцінки кожного ходу повертається назад до попереднього рівня рекурсії, дозволяючи вибрати оптимальний хід на кожному рівні.

Обрізання alpha-beta – це оптимізація алгоритму мінімакс, яка дозволяє значно скоротити кількість досліджуваних ходів. Вона працює на основі двох змінних:

- Alpha – це максимальне значення оцінки, яке було знайдено для поточного гравця на попередніх рівнях рекурсії
- Beta – це мінімальне значення оцінки, яке було знайдено для суперника на попередніх рівнях рекурсії

Логіка обрізання:

- Alpha > beta: Якщо значення alpha(максимальної оцінки для суперника) більше, ніж значення beta(мінімальної оцінки для суперника), то всі наступні ходи на цьому рівні рекурсії можуть бути проігноровані, оскільки вони точно не призведуть до кращого результату для поточного гравця.
- Beta < alpha: Якщо значення beta менше або дорівнює значенню alpha, то всі наступні ходи на цьому рівні рекурсії можуть бути проігноровані, оскільки вони точно не призведуть до кращого результату ждя суперника.

Переваги використання алгоритму:

- Обрізання непотрібних гілок: alpha-beta обрізає гілки дерева пошуку, що не можуть дати кращого результату для поточного гравця. Це значно зменшує кількість ходів, які алгоритм повинен перевірити.
- Збільшення глибини пошуку: Завдяки зменшенню обчислювальної складності, алгоритм може досліджувати дерево можливих ходів на більшу глибину. Це дозволяє отримати більш точне оцінювання стану гри і знайти кращі ходи.
- Оптимальний вибір: Алгоритм забезпечує оптимальний вибір ходу для поточного гравця, припускаючи, що суперник також грає оптимально
- Висока ефективність для детерміністичних ігор: Для ігор, де результат залежить лише від ходів гравців, алгоритм працює дуже ефективно.
- Легко зрозуміти: Алгоритм є відносно простим для розуміння і реалізації

- Широке застосування: Використовується в багатьох іграх, таких як шахи, шашки, ігри з картами та Рендзю і гомоку в тому числі. В шахах, алгоритм мінімакс дозволяє комп’ютерним програмам визначити оптимальні ходи, часто перевершуючи здібності навіть досвідчених гравців. В розробленому програмному додатку цей алгоритм допомагає комп’ютеру знаходити виграшны стратегії та перемагати гравця в 90% випадків.

Недоліки алгоритму:

- Висока обчислювальна складність: Алгоритм має високу часову слкадність, особливо для ігор з великими деревами рішень. Зростання складності є експоненціальним щодо глибини дерева пошуку
- Залежність від порядку ходів: Порядок перевірки ходів може значно вплинути на ефективність обрізання alpha-beta. Неоптимальний порядок може призвести до дослідження більшої кількості ходів, ніж необхідно.
- Неточність для ігор зі змішаною сумою: Алгоритм мінімакс припускає, що гра є з нульовою сумою, де виграш одного гравця дорівнює програшу іншого. Для ігор зі змішаною сумою, де результат не обов’язково є протилежним(наприклад, в деяких варіантах гри “Монополія”), алгоритм може не знайти оптимальний хід.
- Потреба в оцінковій функції: Алгоритм мінімакс з обрізанням alpha-beta може бути складною, особливо для ігор з великими деревами рішень.
- Не враховує випадковості: Алгоритм не може враховувати елемент випадковості в іграх, де ходи визначаються частково випадковими подіями (наприклад, кидання кубиків).
- Не враховує час: Алгоритм не враховує час, необхідний для виконання ходів. Це може бути важливим фактором в іграх, де час має вирішальне значення (наприклад в шахах з контролем часу).

Незважаючи на ці недоліки, алгоритм мінімакс з обрізанням alpha-beta залишається одним з найпоширеніших і ефектиних алгоритмів для пошуку оптимальних ходів у іграх з нульовою сумою.

Використання алгоритму мінімакс з обрізкою alpha-beta у грі Гомоку є досить ефективним, враховуючи, що Гомоку – це детерміністична гра з нульовою сумою, де результат одного гравця є протилежність результату іншого.

Алгоритм у Гомоку:

- Дерево пошуку: Алгоритм створює дерево можливих ходів, починаючи з поточного стану ігрового поля.
- Оціночна функція: Для кожного кінцевого стану (перемога, програш, нічия) визначається оцінка, що відображає результат для поточного гравця.
- Рекурсивний пошук: Алгоритм рекурсивно обходить дерево, оцінюючи кожен хід, припускаючи, що суперник грає оптимально.
- Обрізання alpha-beta: Алгоритм використовує обрізку alpha-beta для зменшення кількості досліджуваних ходів, ігноруючи ті, що не можуть дати кращого результату.
- Вибір оптимального ходу: Алгоритм вибирає хід з максимальною оцінкою для поточного гравця враховуючи, що суперник грає оптимально.

Переваги використання алгоритму в Гомоку:

- Ефективне знаходження оптимальних ходів: Алгоритм допомагає знаходити оптимальні ходи для комп’ютера, що може зробити його більш грізним суперником для гравця
- Зменшення часу обчислення: Обрізання alpha-beta значно зменшує кількість досліджуваних ходів, що дозволяє алгоритму працювати швидше
- Збільшення глибини пошуку: Завдяки скороченню часу обчислення, алгоритм може досліджувати дерево можливих ходів на більшу глибину, що призводить до більш точного вибору ходу.

Недоліки використання алгоритму в Гомоку:

- Складність обчислення: Гомоку має відносно велику кількість можливих ходів, особливо на початку гри. Збільшення глибини пошуку (щоб отримати точніший результат) експоненціально збільшує кількість обчислень, що може призвести до надмірного часу обробки, особливо для слабких пристроїв.
- Не враховує інтуїцію: Алгоритм мінімакс не враховує інтуїцію та творчість, які є важливими складовими в людській грі. Хоча алгоритм може знаходити оптимальні ходи в рамках своєї моделі, але не здатний помітити нестандартні та непередбачувані стратегії
- Не враховує елемент випадковості: Хоча Гомоку є детерміністичною грою, в ній є елемент випадковості, що пов’язаний з початковою розстановкою каменів та вибором першого ходу. Алгоритм мінімакс не може ефективно враховувати цей елемент.

Незважаючи на недоліки, алгоритм мінімакс з обрізкою alpha-beta є досить ефективним для використання в грі Гомоку. Тому що дозволяє створити комп’ютерного противника, що може грати на достатньо високому рівні та складати серйозний виклик для гравця. Однак, важливо враховувати його обмеження та шукати способи їх компенсації, наприклад, використовуючи ефективні евристики та розвиваючи якісні оціночні функції.

2.2 Опис алгоритму у вигляді малюнків і схем

Головною думкою алгоритму являється передбачити можливі варіанти ходу гри, наприклад у програмному коді ми відзначимо список, у якому буде інформація про хід гравця, і за кожен хів, буде відповідати конкретний список, при збігу списку з умовою, комп’ютер буде робити свій хід, і цей час який комп’ютер затрачає для виконання ходу, буде часом роботи алгоритму. Складемо блок схему(рис2.1) по якій буде розроблятись програмний додаток.

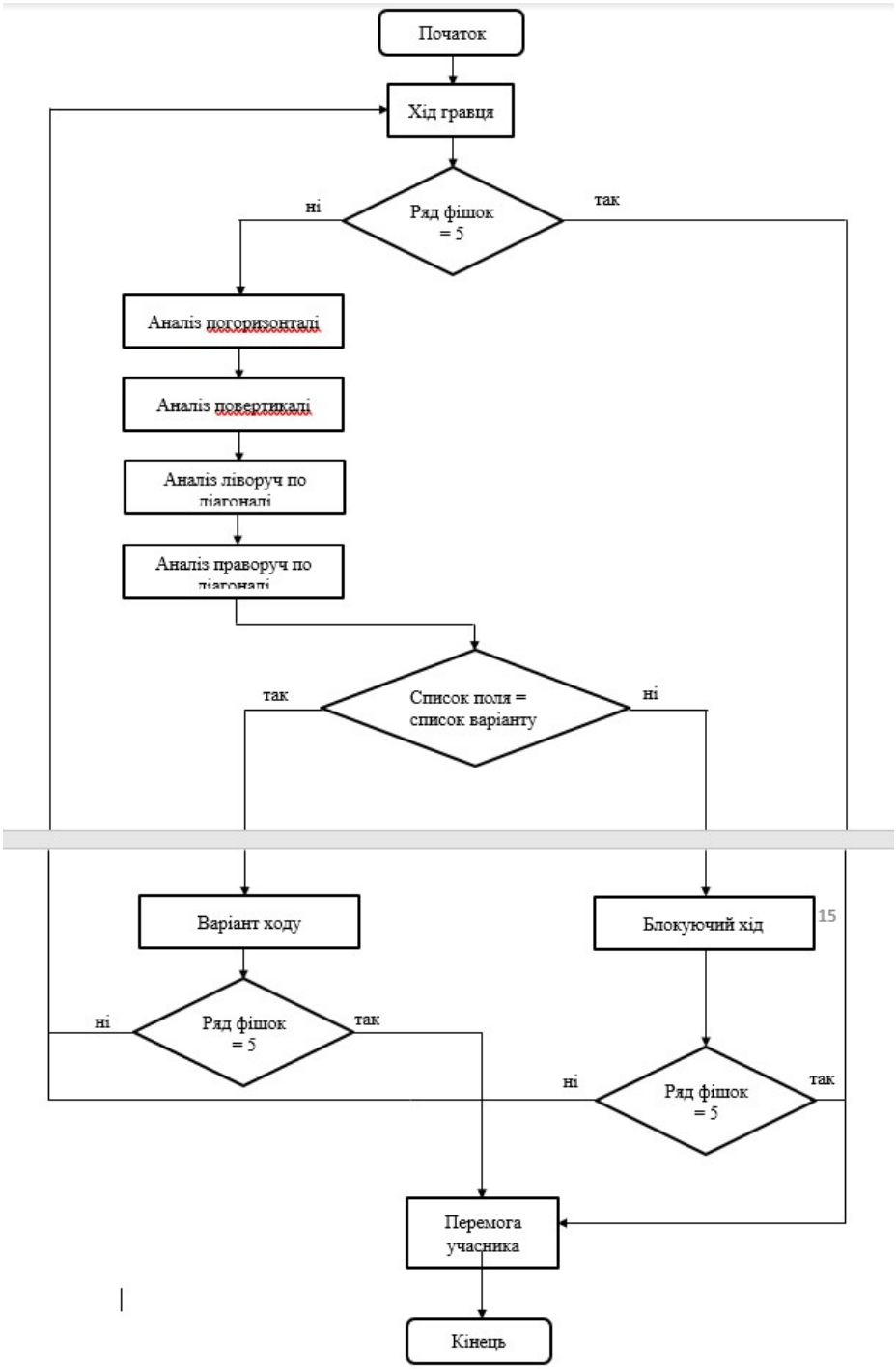


Рисунок 2.1 – Алгоритм функціонування програмного модулю комп’ютерної гри «Гомоку»

Важливим фактором для оцінки позиції є те, наскільки значимі постаті побудували суперники.

П'ятірка (рис. 2.2) - якщо така фігура знайдена на дошці, гра закінчена. Для стандартних правил Гомоку виграш не дають жодних шісток, сімок і т.д.

Тому п'ятірка, як, втім, і всі інші фігури, вимагає відсутності своїх каменів на сусідніх клітках в лінії.

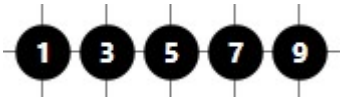


Рисунок 2.2 – Комбінація п’ятірка

Комбінація - безперервна послідовність загроз і захистів від більш значущих загроз суперника, яка веде до позитивного результату для гравця. Комбінації бувають на атакуючі (або виграшні) і захисні. Атакуюча або виграшна комбінація є успішною, якщо на будь-які захисні або атакуючі ходи суперника були знайдені у відповідь ходи, що призводять до виграшу.

Атакуюча комбінація завершується установкою вилки, яку суперник не може закрити. Захисна комбінація, навпаки, успішна, коли у суперник перестає створювати загрози, або перевищено ліміт ходів для розрахунку. Захисна комбінація складається з ходів захисту або створення більшої загрози для суперника.

При оцінці позиції виконується пошук виграшної комбінації. У разі успіху ми виграли. В іншому випадку якщо немає загроз від суперника - стан нейтральне.

При наявності загроз від суперника, виконуємо пошук захисної комбінації. У разі успіху стан нейтрально, при невдачі - ми програли. Так як кількість варіантів атакуючих і вимушених ходів досить обмежена, то пошук комбінацій допустимо виконувати на досить велику глибину.

При початковому побудові оптимальної стратегії дозволена глибина пошуку комбінацій встановлювалася більш 25 ходів.

Відкрита четвірка (рис. 2.3) - довжина 6 клітин, середні чотири зайняті камінням одного кольору, крайні обов'язково порожні. Ну і як для п'ятірки свої камені відсутні на сусідніх клітинах. Дуже сильна фігура, означає виграш навіть на чужому ході.

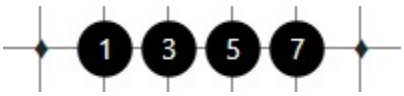


Рисунок 2.3 – Комбінація відкрита четвірка

Четвірка (рис. 2.4) - довжина 5 клітин, одна (будь-яка) з п'яти клітин вільна. Дає виграш на своєму ході. Створює загрозу і змушує суперника зробити хід у вільну клітину, якщо у нього немає своєї четвірки. Дає 5 балів до рейтингу позиції при захисті.

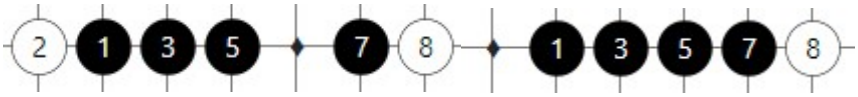


Рисунок 2.4 – Комбінація четвірка

Відкрита трійка (рис. 2.5) - довжина 6 або 7 клітин, крайні клітини обов'язково вільні. Для 6 клітин три з чотирьох середніх зайняті камінням одного кольору, одна вільна. Для 7 клітин - три середніх зайняті камінням одного кольору. Фігура на своєму ході стає відкритою четвіркою, якщо у суперника немає четвірки або відкритої трійки. На чужому ході створює загрозу і змушує суперника закривати трійку або ставити в відповідь свою четвірку. 6-й клітинна трійка має 1 підвищує хід і 3 закривають. 7-й клітинна трійка має 2 підвищують ходу і тільки 2 закривають. Дає від 2-х до 4-х балів до рейтингу позиції.

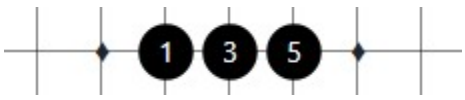


Рисунок 2.5 – Комбінація відкрита трійка

Трійка, або закрита трійка (рис. 2.6) – довжина 5 клітин, будь-які три з яких зайняті камінням одного кольору. Трійка на своєму ході може бути перетворена в четвірку і використовується при атаці і захисті, створюючи загрозу більше, ніж відкрита трійка суперника. Дає 1 бал до рейтингу позиції.

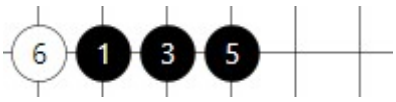


Рисунок 2.6 – Комбінація трійка

Якщо врахувати більшість комбінацій, то гра з комп’ютером не буде легкою, також складність гри буде легко змінювати, додаючи список комбінації, або видаляючи його.

2.3 UML діаграми класів

Діаграма класів є невід’ємною частиною об’єктно-орієнтованого програмування, надаючи візуальне представлення структури програмного забезпечення та його ключових компонентів. Вона служить як карта, яка допомагає розробникам зрозуміти взаємозв’язки між різними частинами системи, а також візуалізувати її архітектуру. Діаграма класів не показує як працює система, але демонструє, з яких частин вона складається та як ці частини взаємодіють між собою

Ключові елементи діаграми класів:

Клас: Представлений прямокутником, розділеним на три частини, що містить інформацію про ім’я класу, атрибути та методи.

- Ім’я класу: вказує на тип об’єкта, який представляє клас.
- Атрибути: Описують властивості об’єкта, наприклад, його колір, розмір, або значення. Це властивості, які характеризують об’єкт класу.
- Методи: Описують дії, які може виконувати об’єкт класу.

Видимість членів класу: Визначає рівень доступу до атрибутів та методів, що контролює, які частини системи можуть взаємодіяти з ними

- Публічний(+): Доступний з будь-якої частини системи.
- Приватний(-): Доступний тільки всередині самого класу.
- Захищений(#): Доступний в класі та його підкласах

Взаємозв’язки між класами: Відображають логічні відносини між об’єктами класів, як вони взаємодіють між собою

- Залежність: Один клас використовує інший, але не має прямого зв’язку ним
- Асоціація: Об’єкти двох класів пов’язані між собою, дозволяючи переходити від одного до іншого
- Агрегація: Один клас складається з інших класів, але вони можуть існувати окремо
- Композиція: Один клас складажться з інших класів, і вони не можуть існувати окремо

Діаграми класів застосовуються для візуалізації структури щоб розуміти з який частин складається система, документування щоб зафіксовувати архітектуру системи для майбутніх розробок та конструювання для створення коду системи, виходячи з діаграми.

Діаграма класів – це потужний інструмент для розробки програмного забезпечення, який допомагає зрозуміти структуру системи, документувати її та створити код, що відповідає цій структурі. Використання діаграм класів допомагає розробникам візуалізувати структуру системи, зрозуміти взаємозв’язки між класами та створити код, що відповідає цій структурі.

Діаграма класів програмного продукту зображена на рис. 2.7

З діаграми видно, яким саме чином побудовані класи, які відображають сутності, а також відношення між класами в програмному продукті

Діаграма класів для гри Гомоку – це візуальне представлення структури гри, що показує як різні класи взаємодіють, щоб створити функціональність гри. Наприклад BoardSearcher використовує GameBoard для виконання можливих ходів та Board Evaluator для оцінки їх результатів. BoardCanvas відображає стан GameBoard на екрані, а BoardFrame організує ці компоненти в єдине вікно

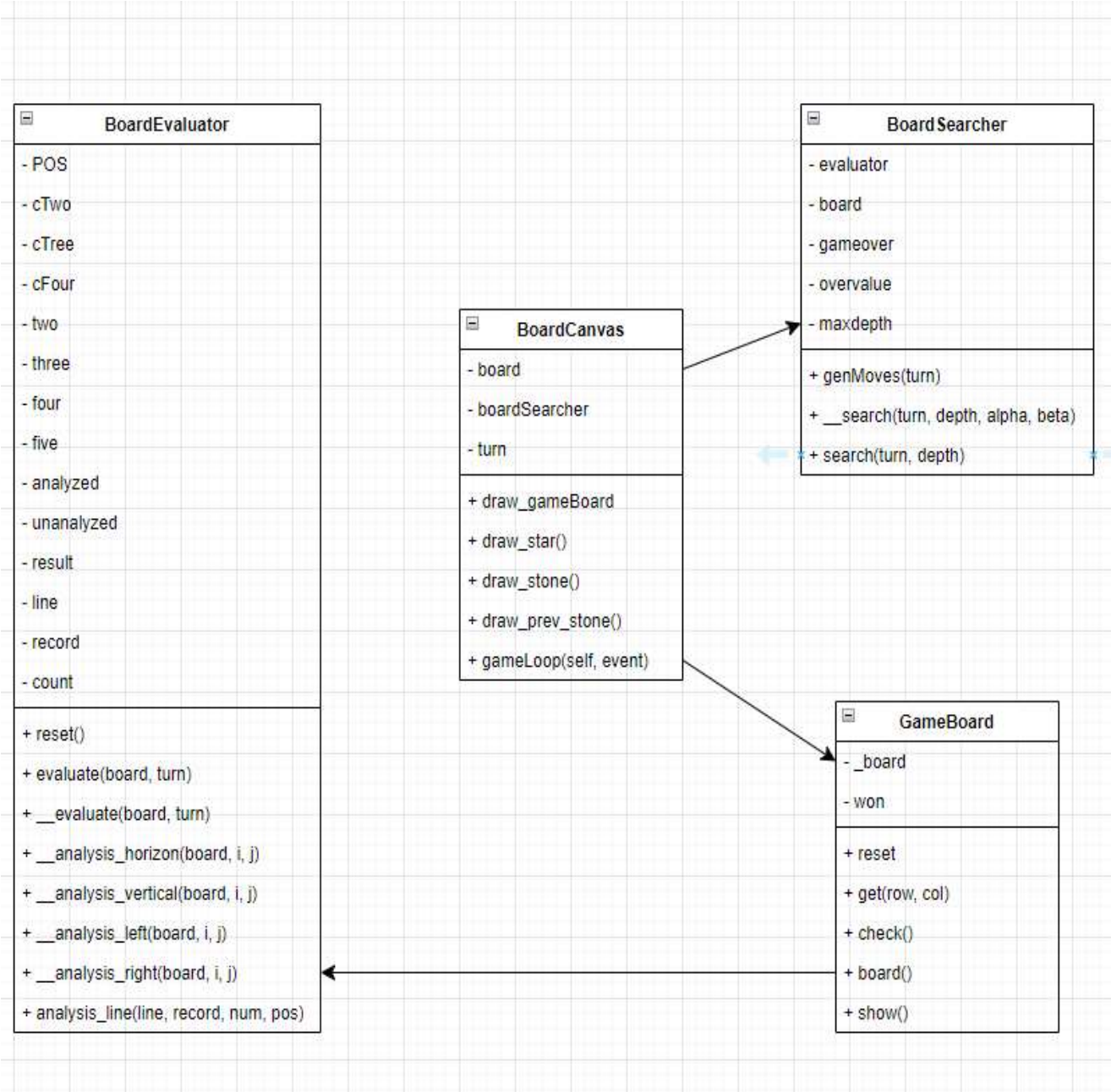


Рисунок 2.7 – UML-діаграма класів програмного модулю комп’ютерної гри «Гомоку»

Основний клас, GameBoard, представляє ігрове поле і контролює всі дії на ньому. Він містить двовимірний масив, що відображає статусу кожного перетину, та інформацію про поточного гравця. GameBoard також відповідає за перевірку допустимості ходів, встановлення каменів на поле та визначення переможця.

Атрибути:

- board: двовимірний масив, який представляє ігрове поле. Кожен елемент масиву містить інформацію про камінь на відповідному перетині(порожнє, чорне, біле).
- won: це логічне значення, що вказує чи була виграна гра.

Методи:

- reset(): скидає стан ігрового поля до початкового. Очищає поле від каменів і встановлює значення won в False.
- get(row, col): повертає значення каменю на перетині (row, col). Дозволяє отримати інформацію про статусу клітинки.
- check(): перевіряє, чи виграна гра. Визначає, чи один з гравців виграв за правилами гри.
- board(): повертає двовимірний масив, що представляє ігрове поле. Дозволяє отримати доступ до стану поля для зовнішніх класів.
- show(): виводить візуальне представлення ігрового поля на екран. Дозволяє переглянути статус гри користувачу.

BoardCanvas відображає статус ігрового поля на екрані за допомогою графіки. Він отримує інформацію від GameBoard про розміщення камені та малює їх на полі

Атрибути:

- board: об’єкт класу GameBoard, який представляє стан ігрового поля.
- boardSearcher: об’єкт класу BoardSearcher, який використовується для обчислення ходів програми.
- turn: поточний гравець (1 для чорних, 2 для білих).

Методи:

- draw_gameBoard(): Малює ігрове поле на дошці.
- draw_star(row, col): Малює зірку на перетині вказаного рядка та стовпця.

- draw_stone(row, col): Малює камінь на перетині вказаного рядка та стовпця.
- draw_prev_stone(row, col): Малює прозорий камінь на перетині вказаного рядка та стовпця.
- gameLoop(self, event): Основний цикл гри. Очікує на кліки користувача, обробляє їх і виконує відповідні дії.

BoardEvaluator використовується для оцінки стану ігрового поля та визначення найкращого ходу. Він використовує двовимірний масив ваг для оцінки кожного перетину і визначає, наскільки вигідним є цей хід для поточного гравця.

Атрибути:

- POS: двовимірний масив, що представляє вагу кожного перетину на ігровому полі. Вага зменшується від центру до країв ігрового поля.
- cTwo, cThree, cFour, two, three, four, five: різні ігрові комбінації(ваги) каменів одного кольору
- analyzed, unanalyzed: Значення, що вказує, чи був проаналізований перетин на ігровому полі.
- result: Масив, що зберігає результати аналізу для кожного перетину на ігровому полі.
- line: Масив, що використовується для зберігання рядка каменів на ігровому полі для аналізу.
- record: Двовимірний масив, що зберігає результати аналізу для кожного перетину на ігровому полі в чотирьох напрямках (горизонтальному, вертикальному, діагональному зліва направо та діагональному справа наліво).
- count: Масив, що зберігає кількість послідовностей каменів певної довжини для кожного кольору гравця.

Методи:

- reset(): Скидає атрибути класу до початкових значень.

- evaluate(board, turn): Оцінює стан ігрового поля для заданого гравця і повертає значення оцінки.
- _evaluate(board, turn): Внутрішній метод, що виконує фактичний аналіз і оцінку ігрового поля.
- _analysis_horizon(board, i, j), _analysis_vertical(board, i, j): Аналізує горизонтальний та вертикальний ряд на ігровому полі і зберігає результати в атрибуті record.
- _analysis_left(board, i, j): Аналізує діагональний ряд зліва направо на ігровому полі і зберігає результати в атрибуті record.
- _analysis_right(board, i, j): Аналізує діагональний ряд справа наліво на ігровому полі і зберігає результати в атрибуті record.
- _analysis_line(line, result, num, pos): Аналізує ряд каменів на ігровому полі і зберігає результати в атрибуті result.

BoardSearcher відповідає за пошук найкращого можливого ходу для комп’ютера. Він використовує алгоритм мінімакс для оцінки всіх можливих ходів до заданої глибини і вибирає найкращий варіант. BoardSearcher працює з Game Board та BoardEvaluator для виконання можливих ходів та оцінки їх результатів

Атрибути:

- evaluator: Об’єкт класу BoardEvaluator. Використовується для оцінки стану ігрового поля.
- board: Об’єкт класу GameBoard. Зберігає поточний стан ігрового поля.
- gameover: Логічне значення. Вказує, чи закінчена гра.
- Overvalue: Значення, що представляє результат гри, коли вона завершена.
- maxdepth: Максимальна глибина пошуку. Визначає, наскільки далеко в майбутнє алгоритм буде заглядати.

Методи:

- genMoves(turn): Генерує список усіх можливих ходів гравця

- _search(turn, depth, alpha, beta): Рекурсивна функція для пошуку найкращого ходу за допомогою алгоритму
- search(turn, depth): Ініціалізує та запускає пошук найкращого ходу. Викликає _search з відповідними параметрами.

BoardFrame організує всі ці компоненти в єдине вікно для користувача. Він містить BoardCanvas візуалізації ігрового поля, та дозволяє користувачеві взаємодіяти з грою.

Взаємозв’язки між цими класами дозволяють створити повну функціональність гри Гомоку. GameBoard контролює логіку гри, BoardCanas відображає статус, BoardEvaluator оцінює стан ігрового поля, BoardSearcher знаходить найкращі ходи а BoardFrame з’єднує всі ці компоненти в одному вікні

На етапі проектування були виділені такі об’єкти, що входять до складу програми:

- main.py головний модуль відповідає за запуск гри, ініціалізацію графічного інтерфейсу (GUI), створення об’єктів GameBoard та BoardSearcher, а також запуск циклу гри.
- board_gui.py візуалізує гру, обробляє натискання користувача на клітинки дошки, відображає результати гри. Також створює кнопки для рівнів та відображає зображення.
- game_board.py керує станом ігрової дошки, визначає перемогу, оновлює відображення дошки.
- board_searcher.py використовує BoardEvaluator для оцінки стану гри та алгоритм мінімакс для пошуку оптимального ходу, а також визначає глибину пошуку depth для кожного рівня складності.
- board_evaluator.py оцінює стан гри, щоб BoardSearcher міг вибрати найкращий хід. Налаштовується для різних видів складностей.

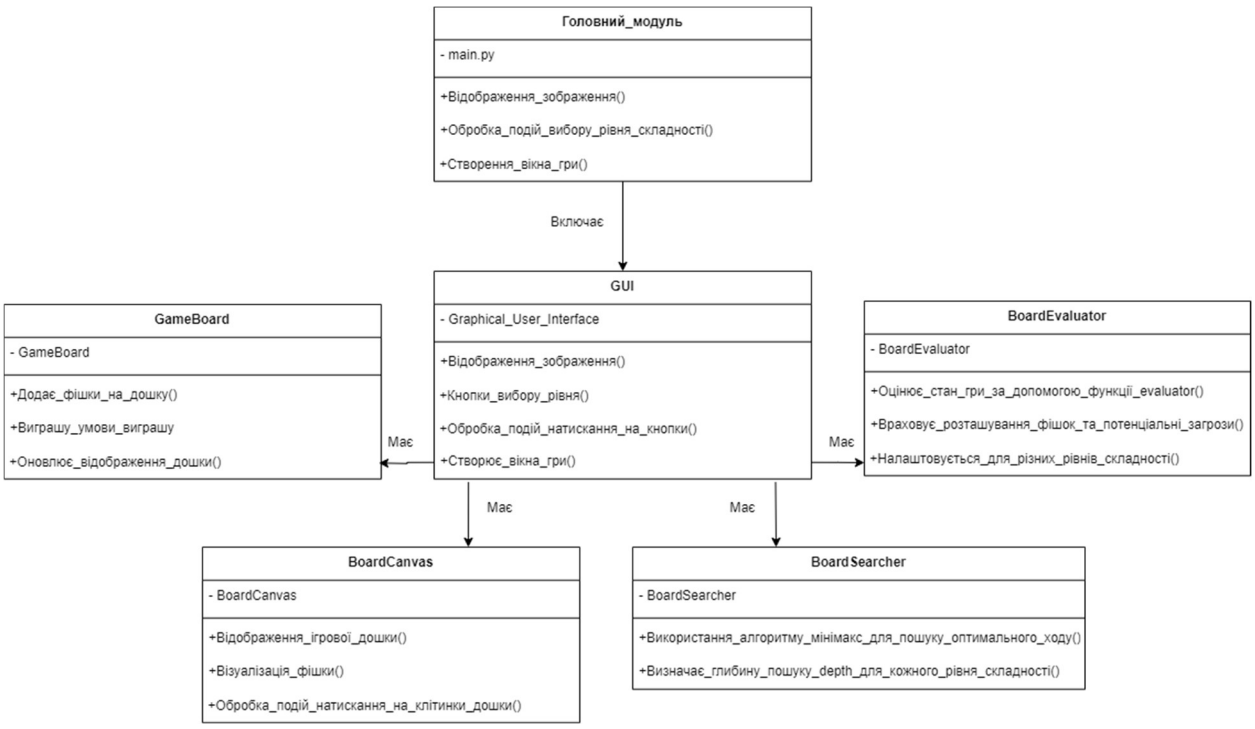


Рисунок 2.8 – Конструктивна схема основних компонентів архітектури програмного модулю комп’ютерної гри «Гомоку»

Всі ці модулі взаємодіють між собою для створення цілісної архітектури комп’ютерної гри «Гомоку».

2.4 Висновок

В другому розділі розроблений алгоритм для гри Гомоку. Наведені основні компоненти які знадобляться для реалізації програмного додатку, створена блок-схема алгоритму та описані принципи роботи.

Також створена UML діаграма класів, зроблений опис кожного атрибуту та методу в діаграмі

Розглянуті різні види комбінації які будуть списком занесені у програмний код на основі яких комп’ютер буде намагаться передбачати можливі варіанти та робити свій хід.

3 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ ГРИ ГОМОКУ

3.1 Обгрунтування вибору інструментів технічної реалізації

Python — це мова, на якій можна написати буквально все, починаючи від чайника, калькулятора, до розумних будинків. В інтернеті можна знайти безліч списків, але у цій статті ми згадаємо коротко про головні напрямки.

- Кібербезпека — захист від злому, створення тестів на проникнення в систему, аналіз систем безпеки, а також розробка програмного забезпечення.
- IoT — також відомий як Інтернет речей, розумні будинки. Ви можете придбати собі кілька девайсів і спробувати налаштувати їх в себе вдома або зайнятися цим професійно.
- Маркетинг — вилучення та аналіз інформації про користувачів із власних даних або використання API Facebook, Google та Twitter.
- Наука — обробка даних на математичному та статистичному рівні, вилучення інформативних частин з результатів лабораторних експериментів з різних областей.
- QA — Ви можете стати тестувальником програмного забезпечення, працюючи з автоматичними тестами.
- Машинне навчання — кажуть, що Python — це майбутнє машинного навчання, про що також радимо Вам дізнатися трохи більше на перспективу.

Python має безліч застосувань та велику потужність, доказом є те, що величезні компанії такі як Google, Dropbox, Spotify та Netflix, активно використовують його у своїх додатках.

Dropbox повністю написаний на Python, що робить його сумісним з будь-якою операційною системою. А на хвилиночку, ним користується близько 400 мільйонів користувачів. Spotify використовує Python для таких речей, як веб-API та для аналізу даних. Netflix використовує поєднання Java, Scala та Python, надаючи розробникам можливість самостійно вибирати мову, яка найкраще

відповідає проблемі, з якою вони стикаються. Також на пайтоні написані такі програми як: Facebook, Instagram, Yahoo, Quora, Pinterest, Disqus.[15]

Опис використаної технології. Tkinter - це пакет для Python, призначений для роботи з бібліотекою Tk. Бібліотека Tk містить компоненти графічного інтерфейсу користувача (graphical user interface - GUI). Ця бібліотека написана на мові програмування Tcl.[16]

Tcl — це динамічна інтерпретована мова програмування, як і Python. Хоча його можна використовувати самостійно як мову програмування загального призначення, найчастіше його вбудовують у додатки C як механізм сценаріїв або інтерфейс до набору інструментів Tk. Бібліотека Tcl має інтерфейс C для створення та керування одним або декількома екземплярами інтерпретатора Tcl, запуску команд і сценаріїв Tcl у цих екземплярах і додавання спеціальних команд, реалізованих у Tcl або C. Кожен інтерпретатор має чергу подій, а також є засоби для надсилання до нього подій та їх обробки. На відміну від Python, модель виконання Tcl розроблена навколо кооперативної багатозадачності, і Tkinter усуває цю різницю.

Під графічним інтерфейсом користувача (GUI) маються на увазі всі ті вікна, кнопки, текстові поля для введення, скролери, списки, радиокнопки, прапорці та ін., Які ви бачите на екрані, відкриваючи ту чи іншу програму. Через них ви взаємодієте з програмою і керуєте нею. Всі ці елементи інтерфейсу будемо називати віджетами (widgets).

В даний час майже всі програми, які створюються для кінцевого користувача, мають GUI. Рідкісні програми, які передбачають взаємодію з людиною, залишаються консольними. У попередніх двох курсах ми писали тільки консольні програми.[17]

Для розробки програмного додатку гра “Гомоку” було обрано середовище visual studi code. Цей вибір зумовлений такими факторами:

1. Безкоштовний, з відкритим вихідним кодом і крос-платформний, який працює на Windows Linux і MacOS.

2. Підтримує мову за промовчанням, яка відповідатиме файлу а також надає змогу змінити мовний режим.

3. VSCode поставляється з вбудованим налагоджувачем, який також є однією з його ключових функцій. Це допомагає прискорити цикл редагування, компіляції та налагодження.

4. Надає повну інтеграцію Git, що дозволяє миттєво бачити зміни, не виходячи із редактора.

5.Надає доступ до таких функцій як Go to Definition, Peek Definition, Find all References і rename Symbol.

6.Забезпечує екстремальне налаштування завдяки своєму гнучкому налаштуванню переваг і безліч розширень.VSC дає можливість змінити тему, змінити сполучення клавіш, створити фрагменти коду та багато іншого.

7.Замість системи проектів вона дозволяє користувачам відкривати один або кілька каталогів, які потім можна зберегти в робочих областях для подальшого використання.[18]

3.2 Програмна реалізація гри Гомоку

Створимо окремий файл для дошки ГО і реалізуємо його інтерфейс. Інтерфейс має складатись з 15 на 15 ліній, які перетинаються і утворюють сітку на перехрестях якої і проходить гра. Імпортуємо бібліотеки Tkinter та Math за допомогою якої буде визначатись вага кожного перехрестя. Чим більше фігур суперника навколо перехрестя тим важчим воно буде, тим більший шанс, що комп’ютер зробить хід конкретно в дане перехрестя. Створимо два класи. Перший буде відповідати за сітку поля. В даному класі можна задати розміри поля, формат сітки, ширину і довжину клітинок. Другий клас буде відповідати за вагу клітинок, він буде реалізовувати запис у список полів уже на яких є фігура, полів на яких немає фігури, також вагу поля при знаходженні фігур і комбінацій на ньому.

Реалізуємо клас для пошуку комбінацій і елементів на полі. Цей клас буде використовувати списки , за допомогою яких, буде проходити контроль елементів на всій дошці. Його аналіз буде проходити таким чином: Аналіз всіх елементів по горизонталі, по вертикалі, ліворуч та праворуч по горизонталі. Також він відповідає за аналіз закінчення гри, а іншими словами якщо на полі буде рядок з п’яти елементів одного кольору, то гру буде закінчено і об’явлено переможця.

Для створення ігрової дошки використовувався class Board Canvas, який завдяки методам self.create_line(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y) та self.draw_star малює 15 горизонтальних так вертикальних ліній по заданим кординатам, а також додає зірки на певні перетини (рис. 3.1).

```
def draw_gameBoard(self):

    # 15 horizontal lines
    for i in range(15):
        start_pixel_x = (i + 1) * 30
        start_pixel_y = (0 + 1) * 30
        end_pixel_x = (i + 1) * 30
        end_pixel_y = (14 + 1) * 30
        self.create_line(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y)

    # 15 vertical lines
    for j in range(15):
        start_pixel_x = (0 + 1) * 30
        start_pixel_y = (j + 1) * 30
        end_pixel_x = (14 + 1) * 30
        end_pixel_y = (j + 1) * 30
        self.create_line(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y)

    # place a "star" to particular intersections
    self.draw_star(3,3)
    self.draw_star(11,3)
    self.draw_star(7,7)
    self.draw_star(3,11)
    self.draw_star(11,11)
```

Рисунок 3.1 – Фрагмент коду типу «Створення ігрової дошки»

Камені на ігровій дошці створюються за допомогою методу draw_stone, який приймає як аргументи рядок і стовпець, у яких має бути розміщений камінь. Цей метод малює овал, який представляє камінь, за допомогою

create_oval. Колір заповнення овалу визначажться поточною чергою гравця, причому чорні камені мають чорне ядро, обведене білим, тоді як білі камені мають біле ядро, обведене чорним (рис. 3.2).

```
def draw_stone(self, row, col):

    inner_start_x = (row + 1) * 30 - 4
    inner_start_y = (col + 1) * 30 - 4
    inner_end_x = (row + 1) * 30 + 4
    inner_end_y = (col + 1) * 30 + 4

    outer_start_x = (row + 1) * 30 - 6
    outer_start_y = (col + 1) * 30 - 6
    outer_end_x = (row + 1) * 30 + 6
    outer_end_y = (col + 1) * 30 + 6

    start_pixel_x = (row + 1) * 30 - 10
    start_pixel_y = (col + 1) * 30 - 10
    end_pixel_x = (row + 1) * 30 + 10
    end_pixel_y = (col + 1) * 30 + 10

    if self.turn == 1:
        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y, fill='black')
        self.create_oval(outer_start_x, outer_start_y, outer_end_x, outer_end_y, fill='white')
        self.create_oval(inner_start_x, inner_start_y, inner_end_x, inner_end_y, fill='black')
    elif self.turn == 2:
        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y, fill='white')
        self.create_oval(outer_start_x, outer_start_y, outer_end_x, outer_end_y, fill='black')
        self.create_oval(inner_start_x, inner_start_y, inner_end_x, inner_end_y, fill='white')
```

Рисунок 3.2 – Фрагмент коду типу «Створення каменів на дошці»

Для Управління ігровим процесом використана функція gameLoop, яка при клацанні мишкою на дошці, визначає координати клітинки, на яку клацнув користувач, і перевіряє, чи хід правильний. Якщо хід правильний функція малює камінь на дошці, змінює чергу на комп’ютер, і перевіряє чи хід правильний. Якщо хід правильний функція малює камінь на дошці, змінює чергу на комп’ютер, перевіряє чи гра закінчена і запускає пошук оптимального ходу для комп’ютера за допомогою BoardSearcher. Після ходу комп’ютера, функція gameLoop знову очікує, поки поки користувач клацне мишею, і цикл повторюється. Якщо гра закінчена, gameLoop виводить

повідомлення про перемогу, а також блокує дошку, щоб користувач не міг робити далі ходи.

Щоб оцінювати стан ігрового поля створений клас BoardEvaluator який ініціалізує різні змінні та списки для оцінки дошки, такі як POS масив позиційних ваг, record який зберігає аналізовані лінії та напрямки на дошці, count який відстежує кількість каменів кожного типу в різних конфігураціях а також розроблені такі функції:

- Список ваг для перехресть на дошці. Додається вага 7 до центру, 6 до зовнішнього квадрату, потім 5, 4, 3, 2, 1, нарешті 0 до самого зовнішнього (рис. 3.3)

```
def __init__ (self):
    # self.POS is for adding weight to each intersetion
    # add weight of 7 to the center, 6 to the outer square, then
    # 5, 4, 3, 2, 1, at last 0 to the outermost square.
    self.POS = []
    for i in range(15):
        row = []
        for j in range(15):
            row.append( 7 - max(abs(i - 7), abs(j - 7)) )
        # row = [ (7 - max(abs(i - 7), abs(j - 7))) for j in range(15) ]
        self.POS.append(tuple(row))
```

Рисунок 3.3 – Фрагмент коду типу «Ваги для перехресть на дошці»

- функція reset() яка скидає всі змінні та списки аналізу дошки, підготовляючи їх до наступної оцінки
- Основна функція оцінки evaluate(board, turn) приймає поточний стан дошки board та поточний хід turn, оцінює стан дошки та повертає оцінку. Спочатку вона викликає _evaluate() для отримання основної оцінки. На основі кількості каменів кожного типу в різних конфігураціях регулює оцінку, додаючи або віднімаючи очки. Якщо гра наближається до кінця то оцінка відповідно коригується. _evaluate(board, turn) аналалізує дошку за допомогою приватних

функцій _analysis_horizon, _analysis_ vertical, _analysis_left та _analysis_right для оцінки ліній, вертикалей, діагоналей (рис. 3.4).

```
def __analysis_horizon (self, board, i, j):
    line = self.line
    result = self.result
    record = self.record
    unanalyzed = self.unanalyzed
    # add each intersection in a row to line
    for x in range(15):
        line[x] = board[i][x]
    self.analysis_line(line, result, 15, j)
    for x in range(15):
        if result[x] != unanalyzed:
            record[i][x][0] = result[x]
    return record[i][j][0]


# analyze vertically
def __analysis_vertical (self, board, i, j):
    line = self.line
    result = self.result
    record = self.record
    unanalyzed = self.unanalyzed
    for x in range(15):
        line[x] = board[x][j]
    self.analysis_line(line, result, 15, i)
    for x in range(15):
        if result[x] != unanalyzed:
            record[x][j][1] = result[x]
    return record[i][j][1]


# analyze left-hand diagonally
def __analysis_left (self, board, i, j):
    line = self.line
    result = self.result
```

Рисунок 3.4 – Фрагмент коду типу «Приватні функції аналізу»

- Функція analysis_line(line, result, num, pos) аналізує лінію каменів на дошці, представлені в списку line, і повертає в списку result значення, що відображають аналіз лінії. Аналіз визначає, чи існують будь-які п’ятірки, четвірки, трійки або інші конфігурації каменів, і відповідно маркує ці лінії. Num представляє довжину проаналізованої лінії, pos – позицію на дошці, де починається лінія.

За пошук найкращого ходу для комп’ютерного гравця в грі Гомоку відповідає клас BoardSearcher. Клас ініціалізує різні змінні, включаючи об’єкт BoardEvaluator для оцінки стану дошки, двовимірний масив board що

представляє ігрову дошку, змінні gameover, overvalue та maxdepth для керування ігровим процесом і пошуком. В цьому класі є такі функції:

- Функція genMoves(turn) генерує список можливих ходів для поточного гравця, використовуючи POSES з BoardEvaluator для оцінки вагових значень кожної клітинки на дошці (рис. 3.5).

```
def genMoves(self, turn):

    moves = []
    board = self.board
    POSES = self.evaluator.POS
    for i in range(15):
        for j in range(15):
            if board[i][j] == 0:
                score = POSES[i][j]
                moves.append((score, i, j))

    moves.sort(reverse=True)    # sort moves in reverse order, i.e., with decreasing scores
    return moves
```

Рисунок 3.5 – Фрагмент коду типу «Генерація списку можливих ходів»

- Функція _search(turn, depth, alpha, beta) є рекурсивною функцією для пошуку найкращого ходу. Вона використовує алгоритм мінімакс з обрізанням alpha-beta для оптимізації пошуку. Функція спочатку перевіряє базовий випадок. Коли глибина пошуку досягла 0, і якщо гра закінчена. Потім вона геренує список можливих ходів, оцінює кожну з них рекурсивно за допомогою _search(), і використовує обрізання alpha-beta для скорочення кількості перевірок (рис. 3.6).

```
def __search(self, turn, depth, alpha = -0x7fffffff, beta = 0x7fffffff):

    # base case: depth is 0
    # evaluate the board and return
    if depth <= 0:
        score = self.evaluator.evaluate(self.board, turn)
        return score

    # if game over, return immediately
    score = self.evaluator.evaluate(self.board, turn)
    if abs(score) >= 9999 and depth < self.maxdepth:
        return score
```

Рисунок 3.6 – Фрагмент коду типу «Пошук найкращого ходу»

- Функція search(turn, depth) запускає пошук найкращого ходу з заданою глибиною пошуку. Вона викликає _search() і повертає оцінку найкращого ходу і його координати

У створені основного вікна гри використовується клас BoardFrame у якому створюється об’єкт BoardCanvas з розміром 550x480 (рис. 3.7).

```
class BoardFrame(tk.Frame):

    def __init__(self, master = None):
        tk.Frame.__init__(self, master)
        self.create_widgets()


    def create_widgets(self):
        self.boardCanvas = BoardCanvas(height = 550, width = 480)
        self.boardCanvas.bind('<Button-1>', self.boardCanvas.gameLoop)
        self.boardCanvas.pack()
```

Рисунок 3.7 – Фрагмент коду типу «Створення ігрового вікна»

Складності гри реалізовані за допомогою зменшенням ваг на дошці та обмеженням глибини пошуку в функціях аналізу (рис. 3.8).

```
if difficulty == "легкий":
    self.three = 3
    self.four = 5
    search_depth = 2
elif difficulty == "середній":
    self.three = 5
    self.four = 6
    search_depth = 4
else:
    search_depth = 6
```

Рисунок 3.8 – Фрагмент коду типу «Реалізація складностей гри»

3.3 Тестування гри Гомоку

При тестуванні програмного додатку гри Гомоку необхідно перевірити правильність виконання графічного інтерфейсу та ігрових механік, що були спроектовані та розроблені під час виконання дипломної роботи.

Тестування розпочато з відкриття головного файлу гри main.py
При відкритті файлу з’являється вікно вибору складності гри(рис. 3.9)

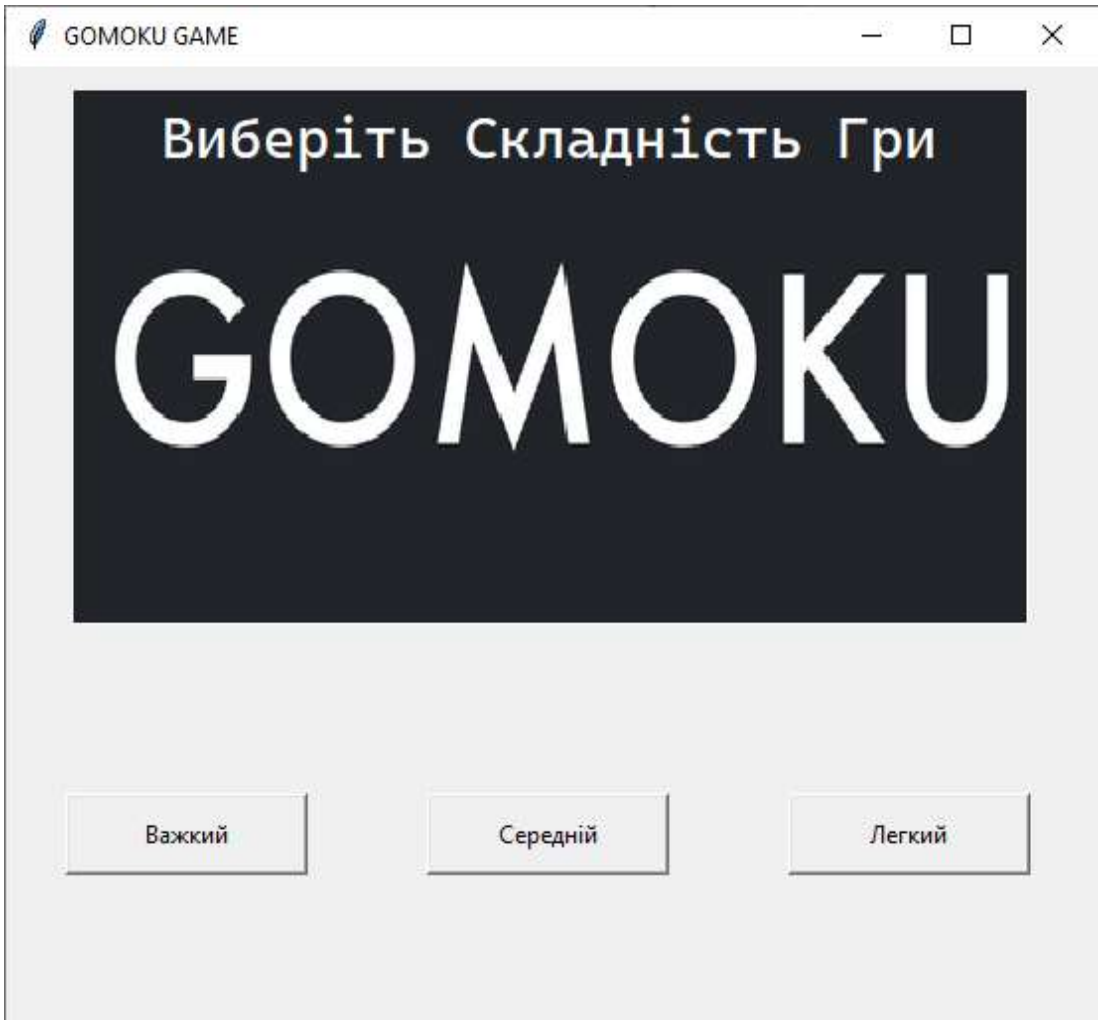


Рисунок 3.9 – Загальний вигляд інтерфейсного меню типу «Вибір складності гри»

Після вибору складності зявляється графічна дошка 15x15 на якій буде проходити ігровий процес(рис. 3.10)

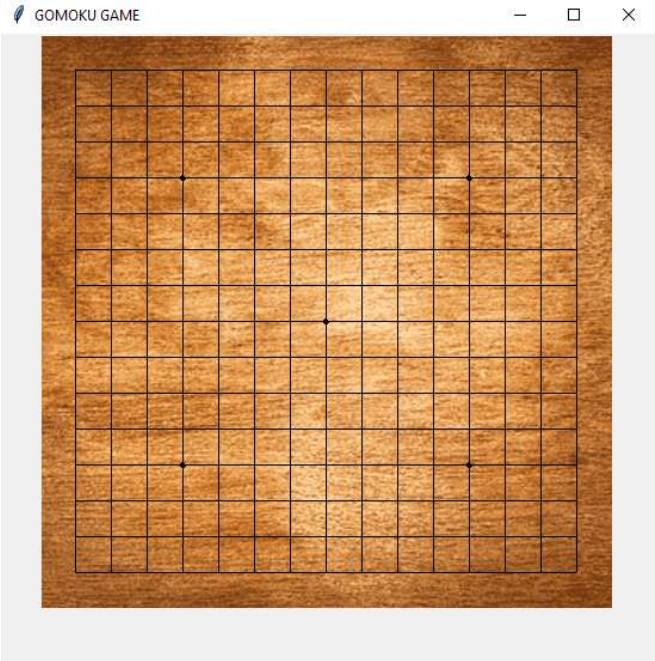


Рисунок 3.10 – Загальний вигляд інтерфейсного меню типу «інтерфейс комп’ютерної гри»

Для початку гри потрібно натисну на будь-яке перехрестя в межах дошки після чого комп’ютер повинен зробити свій хід(рис. 3.11)

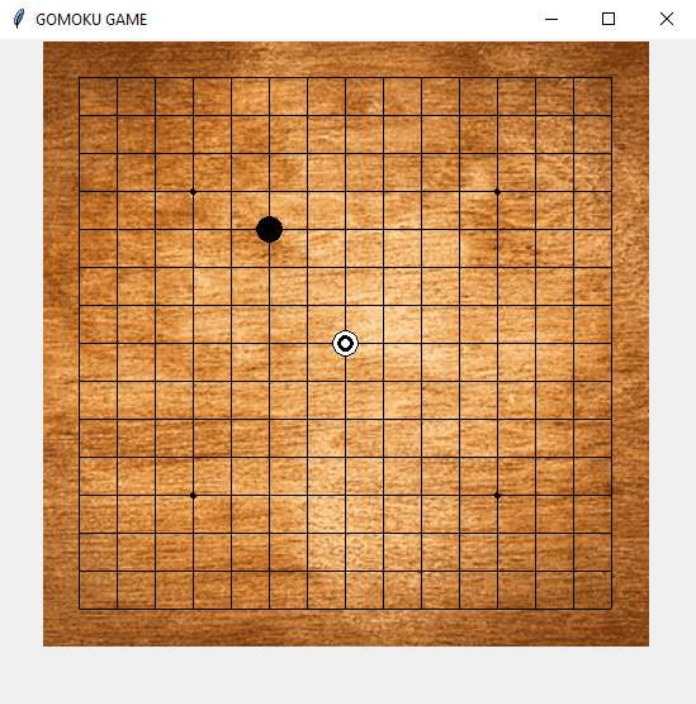


Рисунок 3.11 – Загальний вигляд інтерфейсного меню типу «Приклад першого ходу»

Після того як гравець поставив перший камінь комп’ютер обрав центральне перехрестя оскільки воно найвигідніше для проведення подальших комбінацій. Це демонстує те що алгоритм пошуку працює так як потрібно

Далі хід переходить до користувача і гра продовжується до тих пір поки гравець або комп’ютер не зберуть комбінацю з 5 каменів(рис. 3.12).

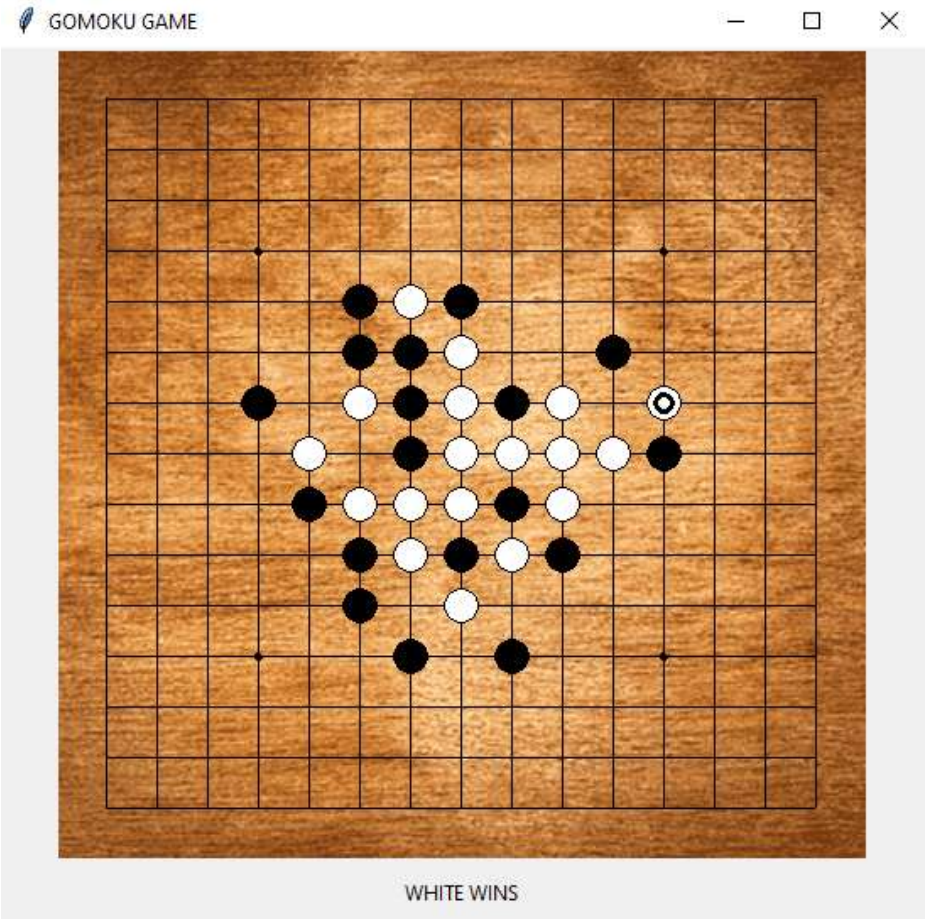


Рисунок 3.12 – Загальний вигляд інтерфейсного меню типу «Перемога білих»

Коли на дошці буде виставлено комбінацію з 5 каменів програма об’явить переможця нижче, а подальше використання гри для виставлення каменів буде заблоковано.

Згідно з проведеними тестами, комп’ютерна гра «Gomoku» виконує всі функції, що зазначалися в проектувані програмного додатку

3.4 Висновок

У даному розділі було обгрунтовано вибір технологій реалізації, а саме мови програмування та середовища розробки. Також було описано технологію реалізації графічного інтерфейсу. Розглянуто програмний код та його основні функції.

Обраними програмними засобами було реалізовано програмний додаток гру «Гомоку».

Було протестовано програмний додаток гру «Гомоку».

ВИСНОВКИ

Поставлені перед бакалаврською дипломною роботою задачі виконано в повному обсязі, а саме:

1) Проаналізовано предметну область та визначено актуальність дослідження.

2) Проведено аналіз існуючих рішень комп'ютерної гри «Гомоку».

3) Розроблено архітектуру програмного модуля комп'ютерної гри.

4) Розроблено алгоритм функціонування програмного модулю комп'ютерної гри.

5) Здійснено програмну реалізацію модулю комп'ютерної гри.

6) Протестувано програмний модуль комп'ютерної гри.

7) Розроблено інструкцію користувача програмного модулю комп'ютерної гри.

У першому розділі було досліджено основні теоретичні відомості у даній предметній області, були проаналізовані уже існуючі реалізації гри Гомоку та інших логічних ігор.

У другому розділі розроблений алгоритм для гри Гомоку. Наведені основні компоненти які знадобляться для реалізації програмного додатку, створена блок-схема алгоритму та UML-діаграма класів.

У третьому розділі було проаналізовано мови програмування, технологію реалізації графічного інтерфейсу та середовище розробки. Було виконано безпосереднє програмування та описано програмну реалізацію гри. Також було виконано тестування розробленої гри на коректність роботи. У результаті тестів виявлено що програма працює коректно.

Мета роботи - розширення функціональних можливостей та покращення графічного інтерфейсу – досягнута. Розроблена гра має потенціал для використання в освітніх та розважальних цілях, а також для розвитку когнітивних здібностей гравців.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Найцікавіші логічні ігри для майбутніх програмістів [Електронний ресурс] – Режим доступу: https://itfuture.online/uk/najczikavishi-logichni-igry-dlya-majbutnih-programistiv/
2. Millington, Roger. *Crossword Puzzles, Their History and Their Cult: Roger Millington*. T. Nelson, 1974
3. Найкращі головоломки 2024 року [Електронний ресурс] – Режим доступу: https://ahaslides.com/uk/blog/free-word-search-games/
4. Найкращі відеоігри головоломки [Електронний ресурс] – Режим доступу: https://techno.nv.ua/ukr/technoblogs/yak-trenuvati-mozok-10-naykrashchih-videoigor-golovolomok-50092612.html
5. Chen, Tsung Teng. "Chitetris: A Chinese proverb learning game utilizing Tetris game plot." *2010 Third IEEE International Conference on Digital Game and Intelligent Toy Enhanced Learning*. IEEE, 2010.
6. Alus, L. V., H. J. van den Herik, 1, and Matty PH Huntjens. "Go-Moku solved by new search techniques." *Computational Intelligence* 12.1 (1996): 7-23.
7. Wágner, János, and István Virág. "Solving renju." *ICGA journal* 24.1 (2001): 30-35.
8. Як грати у Гомоку [Електронний ресурс] – Режим доступу: https://uk.boardgamearena.com/gamepanel?game=gomoku
9. Ikonen, Jussi. "Organizer's Manual for Renju Team World Championships." (2008).
10. Allis, Louis Victor. "Searching for solutions in games and artificial intelligence." (1994).
11. Szőts, János, and István Harmati. "Development of an incremental pattern extraction based Gomoku agent." *Periodica Polytechnica Electrical Engineering and Computer Science* 62.4 (2018): 155-164.
12. Oda, Moriya, and Baxter F. Womack. "Special multimodality in renju game." *1970 IEEE Symposium on Adaptive Processes (9th) Decision and Control*. IEEE, 1970.

13. Bocanegra, Joseph, and Hubert Cecotti. "Tamentai: An abstract strategic board game in fully immersive virtual reality."
14. Vardi, Avi. "New minimax algorithm." *Journal of Optimization Theory and Applications* 75.3 (1992): 613-634.
15. Van Rossum, Guido. "Python Programming Language." *USENIX annual technical conference*. Vol. 41. No. 1. 2007.
16. Lundh, Fredrik. "An introduction to tkinter." *URL: www. pythonware. com/library/tkinter/introduction/index. htm* (1999).
17. Moore, Alan D. *Python GUI Programming with Tkinter: Design and build functional and user-friendly GUI applications*. Packt Publishing Ltd, 2021
18. Code, Visual Studio. "Visual studio code." *Recuperado el Octubre de* (2019).
19. Farooq, Aamir, and Vadim Zaytsev. "There is more than one way to zen your python." *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 2021.

ДОДАТКИ

ДОДАТОК А

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Комп'ютерна гра Гомоку з багатьма рівнями складності

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра комп'ютерних наук, ФІТА
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 85,4% Схожість 14,6%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку Озеранський В.С.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

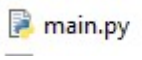
Автор роботи Парванчук Д.С.

Керівник роботи Малініч І.П.

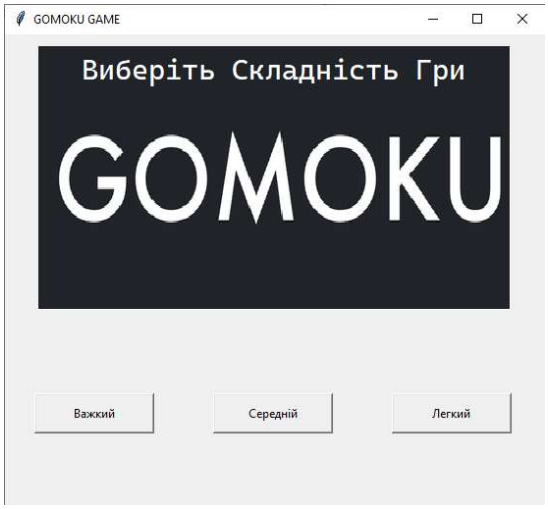
Додаток Б

ІНСТРУКЦІЯ КОРИСТУВАЧА

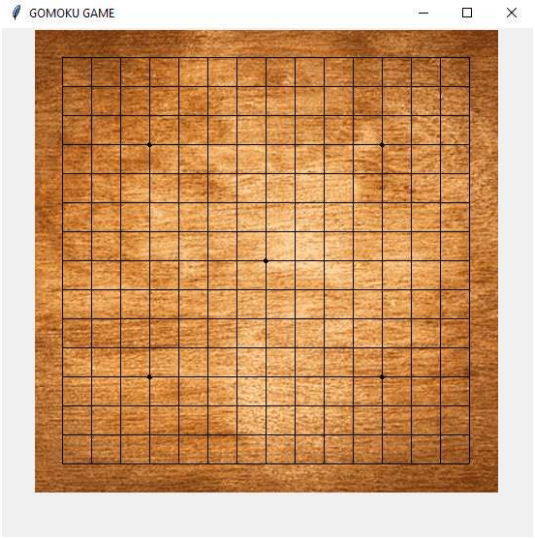
1. Запустити програмний додаток.



2. Перед користувачем постане вікно з вибором складності гри



3. Після вибору складності гри з’явиться дошка ГО на якій потрібно грати.



4. Щоб зробити хід потрібно нажати лівою кнопкою миші на перехресті будь-яких ліній.
5. Після цього хід переходить до комп'ютера, комп'ютер робить хід блокуючи чи атакуючи суперника.
6. Хід повертається до гравця і це повторюється до тих пір поки або гравець, або комп'ютер складе комбінацію з п'яти елементів у ряд.
7. Хто скаладає ряд з 5 елементів свого кольору, той стає переможцем.

Додаток В

ЛІСТИНГ ПРОГРАМНОГО КОДУ

main.py:

```
import tkinter as tk
from board_gui import BoardFrame

def main():
	window = tk.Tk()
	window.wm_title("GOMOKU GAME")
	gui_board = BoardFrame(window)
	gui_board.pack()
	window.mainloop()


if __name__ == "__main__":
	main()
```

board_gui.py:

```
import tkinter as tk
import math
from game_board import GameBoard
from board_searcher import BoardSearcher


class BoardCanvas(tk.Canvas):
	"""Apply the tkinter Canvas Widget to plot the game board and stones."""

	def __init__(self, master=None, height=0, width=0):

		tk.Canvas.__init__(self, master, height=height, width=width)
		self.draw_gameBoard()
		self.gameBoard = GameBoard()
		self.boardSearcher = BoardSearcher()
		self.boardSearcher.board = self.gameBoard.board()
		self.turn = 2
		self.undo = False
		self.depth = 2
		self.prev_exist = False
		self.prev_row = 0
		self.prev_col = 0


	def draw_gameBoard(self):
		"""Plot the game board."""

		# 15 horizontal lines
		for i in range(15):
			start_pixel_x = (i + 1) * 30
			start_pixel_y = (0 + 1) * 30
			end_pixel_x = (i + 1) * 30
			end_pixel_y = (14 + 1) * 30
			self.create_line(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y)

		# 15 vertical lines
		for j in range(15):
			start_pixel_x = (0 + 1) * 30
			start_pixel_y = (j + 1) * 30
			end_pixel_x = (14 + 1) * 30
			end_pixel_y = (j + 1) * 30
			self.create_line(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y)
```

```
                # place a "star" to particular intersections
                self.draw_star(3,3)
                self.draw_star(11,3)
                self.draw_star(7,7)
                self.draw_star(3,11)
                self.draw_star(11,11)


        def draw_star(self, row, col):
                """Draw a "star" on a given intersection

                Args:
                        row, col (i.e. coord of an intersection)
                """
                start_pixel_x = (row + 1) * 30 - 2
                start_pixel_y = (col + 1) * 30 - 2
                end_pixel_x = (row + 1) * 30 + 2
                end_pixel_y = (col + 1) * 30 + 2

                self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y, fill =
'black')


        def draw_stone(self, row, col):
                """Draw a stone (with a circle on it to denote latest move) on a given intersection.

                Specify the color of the stone depending on the turn.

                Args:
                        row, col (i.e. coord of an intersection)
                """

                inner_start_x = (row + 1) * 30 - 4
                inner_start_y = (col + 1) * 30 - 4
                inner_end_x = (row + 1) * 30 + 4
                inner_end_y = (col + 1) * 30 + 4

                outer_start_x = (row + 1) * 30 - 6
                outer_start_y = (col + 1) * 30 - 6
                outer_end_x = (row + 1) * 30 + 6
                outer_end_y = (col + 1) * 30 + 6

                start_pixel_x = (row + 1) * 30 - 10
                start_pixel_y = (col + 1) * 30 - 10
                end_pixel_x = (row + 1) * 30 + 10
                end_pixel_y = (col + 1) * 30 + 10

                if self.turn == 1:
                        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y,
fill='black')
                        self.create_oval(outer_start_x, outer_start_y, outer_end_x, outer_end_y,
fill='white')
                        self.create_oval(inner_start_x, inner_start_y, inner_end_x, inner_end_y,
fill='black')
                elif self.turn == 2:
                        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y,
fill='white')
                        self.create_oval(outer_start_x, outer_start_y, outer_end_x, outer_end_y,
fill='black')
                        self.create_oval(inner_start_x, inner_start_y, inner_end_x, inner_end_y,
fill='white')
```

```
def draw_prev_stone(self, row, col):
    """Draw the previous stone with single color.

    Specify the color of the stone depending on the turn.

    Args:
        row, col (i.e. coord of an intersection)
    """

    start_pixel_x = (row + 1) * 30 - 10
    start_pixel_y = (col + 1) * 30 - 10
    end_pixel_x = (row + 1) * 30 + 10
    end_pixel_y = (col + 1) * 30 + 10

    if self.turn == 1:
        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y,
fill='white')
    elif self.turn == 2:
        self.create_oval(start_pixel_x, start_pixel_y, end_pixel_x, end_pixel_y,
fill='black')


def gameLoop(self, event):
    """The main loop of the game.
    Note: The game is played on a tkinter window. However, there is some quite useful
information
        printed onto the terminal such as the simple visualizaiton of the board
after each turn,
        messages indicating which step the user reaches at, and the game over
message. The user
        does not need to look at what shows up on the terminal.

    self.gameBoard.board()[row][col] == 1(black stone) / 2(white stone)
    self.gameBoard.check() == 1(black wins) / 2(white wins)

    Args:
        event (the position the user clicks on using a mouse)
    """

    while True:
        # User's turn. Place a black stone.
        print('Your turn now...\n')
        self.turn = 1
        invalid_pos = True
        # since a user might not click exactly on an intersection, place the stone
onto
        # the intersection closest to where the user clicks
        for i in range(15):
            for j in range(15):
                pixel_x = (i + 1) * 30
                pixel_y = (j + 1) * 30
                square_x = math.pow((event.x - pixel_x), 2)
                square_y = math.pow((event.y - pixel_y), 2)
                distance =  math.sqrt(square_x + square_y)

                # since there is noly one intersection such that the
distance between it
                # and where the user clicks is less than 15, it is not
necessary to find
                # the actual least distance
```

```
                                        if (distance < 15) and (self.gameBoard.board()[i][j] ==
0):
                                            invalid_pos = False
                                            row, col = i, j
                                            self.draw_stone(i,j)
if self.prev_exist == False:
                                                self.prev_exist = True
                                            else:
                                                self.draw_prev_stone(self.prev_row,
self.prev_col)
                                            self.prev_row, self.prev_col = i, j
                                            # unbind to ensure the user cannot click
anywhere until the program
                                            # has placed a white stone already
                                            self.unbind('<Button-1>')
                                            break    # break the inner for loop
                                    else:
                                        continue# executed if the inner for loop ended
normally(no break)
                                    break                # executed if 'continue'
skipped(break)
                                                    # break the outer for loop

                            # break the inner while loop
                            if invalid_pos:
                                print('Invalid position.\n')
                                break
                            else:
                                break

                    # Place a black stone after determining the position
                    self.gameBoard.board()[row][col] = 1

                    # If the user wins the game, end the game and unbind.
                    if self.gameBoard.check() == 1:
                        print('BLACK WINS !!')
                        self.create_text(240, 500, text = 'BLACK WINS !!')
                        self.unbind('<Button-1>')
                        return 0

                    # Change the turn to the program now
                    self.turn = 2
                    print('Program is thinking now...')

                    # Determine the position the program will place a white stone on.
                    # Place a white stone after determining the position.
                    score, row, col = self.boardSearcher.search(self.turn, self.depth)
                    coord = '%s%s'%(chr(ord('A') + row), chr(ord('A') + col))
                    print('Program has moved to {}\n'.format(coord))
                    self.gameBoard.board()[row][col] = 2
                    self.draw_stone(row,col)
                    if self.prev_exist == False:
                        self.prev_exist = True
                    else:
                        self.draw_prev_stone(self.prev_row, self.prev_col)
                    self.prev_row, self.prev_col = row, col
                    self.gameBoard.show()
                    print('\n')

                    # bind after the program makes its move so that the user can continue to play
                    self.bind('<Button-1>', self.gameLoop)
```

```
                # If the program wins the game, end the game and unbind.
                if self.gameBoard.check() == 2:
                        print('WHITE WINS.')
                        self.create_text(240, 500, text = 'WHITE WINS')
                        self.unbind('<Button-1>')
                        return 0


class BoardFrame(tk.Frame):
        """The Frame Widget is mainly used as a geometry master for other widgets, or to
        provide padding between other widgets.
        """

        def __init__(self, master = None):
                tk.Frame.__init__(self, master)
                self.create_widgets()


        def create_widgets(self):
                self.boardCanvas = BoardCanvas(height = 550, width = 480)
                self.boardCanvas.bind('<Button-1>', self.boardCanvas.gameLoop)
                self.boardCanvas.pack()
```

board_searcher.py:

```
from board_evaluator import BoardEvaluator


class BoardSearcher(object):
        """Board searcher for best next move."""

        def __init__ (self):
                self.evaluator = BoardEvaluator()
                self.board = [ [ 0 for n in range(15) ] for i in range(15) ]
                self.gameover = 0
                self.overvalue = 0
                self.maxdepth = 3               # set the max depth to 3 so that the running time
                                                        # for each move is not too long
                                                        # depth: 1 - <1 sec, 2 - a few sec, 3 -
up to 4 min


        def genMoves(self, turn):
                """Generate all legal moves for the current board.

                store the score and position of each move in a list in format of (score, i, j)
                """
                moves = []
                board = self.board
                POSES = self.evaluator.POS
                for i in range(15):
                        for j in range(15):
                                if board[i][j] == 0:
                                        score = POSES[i][j]
                                        moves.append((score, i, j))

                moves.sort(reverse=True) # sort moves in reverse order, i.e., with decreasing
scores
                return moves


        def __search(self, turn, depth, alpha = -0x7fffffff, beta = 0x7fffffff):
                """Recursive search, return the best score.
```

```
                Minimax algorithm with alpha-beta pruning.
                0x7fffffff == (2^31)-1, indicating a large value
                """

                # base case: depth is 0
                # evaluate the board and return
                if depth <= 0:
                        score = self.evaluator.evaluate(self.board, turn)
                        return score

                # if game over, return immediately
                score = self.evaluator.evaluate(self.board, turn)
                if abs(score) >= 9999 and depth < self.maxdepth:
                        return score

                # generate new moves
                moves = self.genMoves(turn)
                bestmove = None

                # for all current moves
                # len(moves) == num of empty intersections on current board
                # worst case O(m^n) or O( m!/(m-n)! ), m = num of empty spots,
                #                       n = depth(num of further steps this program predicts)
                for score, row, col in moves:

                        # label current move to board
                        self.board[row][col] = turn

                        # calculate next turn
                        if turn == 1:
                                nturn = 2
                        elif turn == 2:
                                nturn = 1

                        # DFS, return score and position of move
                        score = - self.__search(nturn, depth - 1, -beta, -alpha)

                        # clear current move on board
                        self.board[row][col] = 0

                        # calculate the move with best score
                        # alpha beta pruning: removes nodes that are evaluated by the minimax
algorithm
                        #                       in the search tree, eliminates branches
that cannot posibbly
                        #                       influence the final decision.
                        if score > alpha:
                                alpha = score
                                bestmove = (row, col)
                                if alpha >= beta:
                                        break

                # if depth is max depth, record the best move
                if depth == self.maxdepth and bestmove:
                        self.bestmove = bestmove

                # return current best score and its correponding move
                return alpha

        # specific search
        # args: turn: 1(black)/2(white), depth
```

```
                    def search(self, turn, depth=3):
                        self.maxdepth = depth
                        self.bestmove = None
                        score = self.__search(turn, depth)
                        if abs(score) > 8000:
                            self.maxdepth = depth
                            score = self.__search(turn, 1)
                        row, col = self.bestmove
                        return score, row, col
```

game_board.py:

```
class GameBoard(object):
    """Game board."""

    def __init__(self):
        # board is a 15*15 array: each posision is initially set to be 0
        self.__board = [[0 for _ in range(15)] for _ in range(15)]

        # store positions of 5 stones in a line
        self.won = {}


    def reset(self):
        """Clear the board (set all position to 0)."""
        self.__board = [[0 for _ in range(15)] for _ in range(15)]


    def get(self, row, col):
        """Get the value at a coord."""

        if row < 0 or row >= 15 or col < 0 or col >= 15:
            return 0
        return self.__board[row][col]


    def check(self):
        """Check if there is a winner.

        Returns:
            0-no winner, 1-black wins, 2-white wins
        """
        board = self.__board
        # check in 4 directions
        # a coordinate stands for a specific direction, imagine the direction of a coordinate
        # relative to the origin on xy-axis
        dirs = ((1, -1), (1, 0), (1, 1), (0, 1))
        for i in range(15):
            for j in range(15):
                # if no stone is on the position, don't need to consider this position
                if board[i][j] == 0:
                    continue
                # value-value at a coord, i-row, j-col
                value = board[i][j]
                # check if there exist 5 in a line
                for d in dirs:
                    x, y = i, j
                    count = 0
                    for _ in range(5):
                        if self.get(x, y) != value:
                            break
                        x += d[0]
                        y += d[1]
                        count += 1
```

```
                # if 5 in a line, store positions of all stones, return value
                if count == 5:
                    self.won = {}
                    r, c = i, j
                    for _ in range(5):
                        self.won[(r, c)] = 1
                        r += d[0]
                        c += d[1]
                    return value
        return 0


    def board(self):
        """Return the board array."""
        return self.__board


    def show(self):
        """Output current board on terminal."""
        print('  A B C D E F G H I J K L M N O')
        self.check()
        for col in range(15):
            print(chr(ord('A') + col), end=" ")
            for row in range(15):
                ch = self.__board[row][col]
                if ch == 0:
                    print('.', end=" ")
                elif ch == 1:
                    print('X', end=" ")
                elif ch == 2:
                    print('O', end=" ")
            print()
```

board_evaluator.py:

```
'''
                        Define an BoardEvaluator class.
                        Note: I wrote more than 100 lines if-elif-else statements to discuss every single
                                possible scenarios that could occur. This portion of code looks ugly but it is
                                quite necessary for the evaluation process.
'''
class BoardEvaluator(object):

                        def __init__ (self):
                                # self.POS is for adding weight to each intersetion
                                # add weight of 7 to the center, 6 to the outer square, then
                                # 5, 4, 3, 2, 1, at last 0 to the outermost square.
                                self.POS = []
                                for i in range(15):
                                        row = []
                                        for j in range(15):
                                                row.append( 7 - max(abs(i - 7), abs(j - 7)) )
                                        # row = [ (7 - max(abs(i - 7), abs(j - 7))) for j in range(15) ]
                                        self.POS.append(tuple(row))



                                # different types of situations below
                                self.cTwo = 1           # chong'er      2 stones in a row, 1 move to make a
chongsan
                                self.cThree = 2         # chongsan      3 stones in a row, 1 move to make a
chongsi
```

```
                self.cFour = 3          # chongsi       4 stones in a row, 1 move(1 possible position) to make a 5
                self.two = 4            # huo'er 2 stones in a row, 1 move to make a huosan
                self.three = 5          # huosan        3 stones in a row, 1 move to make a huosi
                self.four = 6           # huosi         4 stones in a row, 1 move(2 possible positions) to make a 5
                self.five = 7           # huowu         5 stones in a row
                self.analyzed = 8       # has benn analyzed
                self.unanalyzed = 0                     # has not been analyzed
                self.result = [ 0 for i in range(30) ]          # save current reslut of analyzation in a line
                self.line = [ 0 for i in range(30) ]            # current data in a line
                self.record = []                # result of analysis of whole board
                                                                # format of each item in list is record[row][col][dir]
                for i in range(15):
                        self.record.append([])
                        self.record[i] = []
                        for j in range(15):
                                self.record[i].append([ 0, 0, 0, 0])
                self.count = []                 # count of each situation: count[black/white][situation]
                for i in range(3):
                        data = [ 0 for i in range(10) ]
                        self.count.append(data)
                self.reset()


        # reset data
        def reset(self):
                unanalyzed = self.unanalyzed
                count = self.count
                for i in range(15):
                        line = self.record[i]
                        for j in range(15):
                                line[j][0] = unanalyzed
                                line[j][1] = unanalyzed
                                line[j][2] = unanalyzed
                                line[j][3] = unanalyzed
                for i in range(10):
                        count[0][i] = 0
                        count[1][i] = 0
                        count[2][i] = 0
                return 0


        # analyze & evaluate board
        # return score based on analysis result
        def evaluate (self, board, turn):
                score = self.__evaluate(board, turn)
                count = self.count
                if score < -9000:
                        if turn == 1:
                                stone = 2
                        elif turn == 2:
                                stone = 1
                        # print('evaluate: stone = ', stone)
                        for i in range(10):
                                if count[stone][i] > 0:
                                        score -= i
                elif score > 9000:
```

```
                if turn == 1:
                        stone = 2
                elif turn == 2:
                        stone = 1
                # print('evaluate: stone = ', stone)
                for i in range(10):
                        if count[turn][i] > 0:
                                score += i
        return score


# analyze & evaluate board
# in 4 directinos: horizontal, vertical, diagonal(left-hand or right-hand)
# return score difference between players based on analysis result
def __evaluate (self, board, turn):
        record = self.record
        count = self.count
        unanalyzed = self.unanalyzed
        analyzed = self.analyzed
        self.reset()
        # analysis in 4 directions
        for i in range(15):
                boardrow = board[i]
                recordrow = record[i]
                for j in range(15):
                        if boardrow[j] != 0:
                                # has not analyzed horizontally
                                if recordrow[j][0] == unanalyzed:
                                        self.__analysis_horizon(board, i, j)
                                # has not analyzed vertically
                                if recordrow[j][1] == unanalyzed:
                                        self.__analysis_vertical(board, i, j)
                                # has not analyzed left-hand diagonally
                                if recordrow[j][2] == unanalyzed:
                                        self.__analysis_left(board, i, j)
                                # has not analyzed right-hand diagonally
                                if recordrow[j][3] == unanalyzed:
                                        self.__analysis_right(board, i, j)

        five = self.five
        four = self.four
        three =  self.three
        two = self.two
        cFour = self.cFour
        cThree = self.cThree
        cTwo = self.cTwo

        check = {}

        # for either white or black, calculated the number of occurences of different
        # situations (i.e., five, four, cFour, three, cThree, two, cTwo)
        for c in (five, four, cFour, three, cThree, two, cTwo):
                check[c] = 1
        # for each stone on the board
        for i in range(15):
                for j in range(15):
                        stone = board[i][j]
                        if stone != 0:
                                # for 4 directions
                                for k in range(4):
                                        ch = record[i][j][k]
                                        if ch in check:
```

```
                                                    count[stone][ch] += 1

# return score if there is a five
black = 1
white = 2
# current turn is white
if turn == white:
        if count[black][five]:
                return -9999
        elif count[white][five]:
                return 9999
# current turn is black
else:
        if count[white][five]:
                return -9999
        elif count[black][five]:
                return 9999

# if there exist 2 chongsi, it's equivalent to 1 huosi
if count[white][cFour] >= 2:
        count[white][four] += 1
if count[black][cFour] >= 2:
        count[black][four] += 1

# return score for specific situations
wvalue = 0
bvalue = 0
w
```

ДОДАТОК Г

ІЛЮСТРАТИВНА ЧАСТИНА

«КОМП'ЮТЕРНА ГРА ГОМОКУ З БАГАТЬМА РІВНЯМИ СКЛАДНОСТІ»

Виконав: студент 4 курсу, групи 2КН-20б
Спеціальності 122 – Комп'ютерні науки
(шифр і назва напрямку підготовки, спеціальності)
Парванчук Д.С.
(прізвище та ініціали)
Керівник: асистент каф. КН
Малініч І. П.
(прізвище та ініціали)

«07» 06 2024 р.

Вінниця ВНТУ – 2024 рік

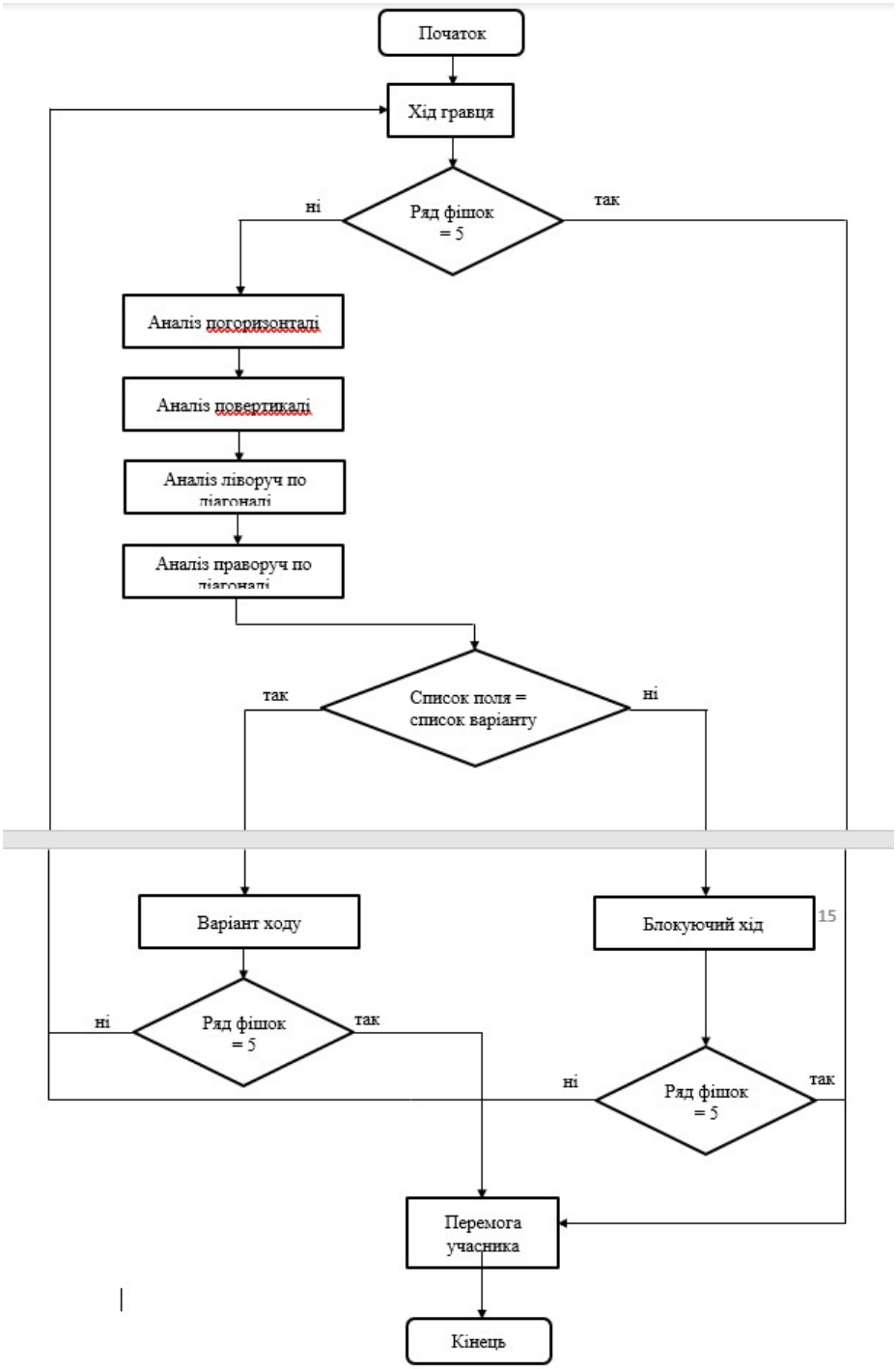


Рисунок Г.1 – Блок-схема алгоритму гри «Гомоку»

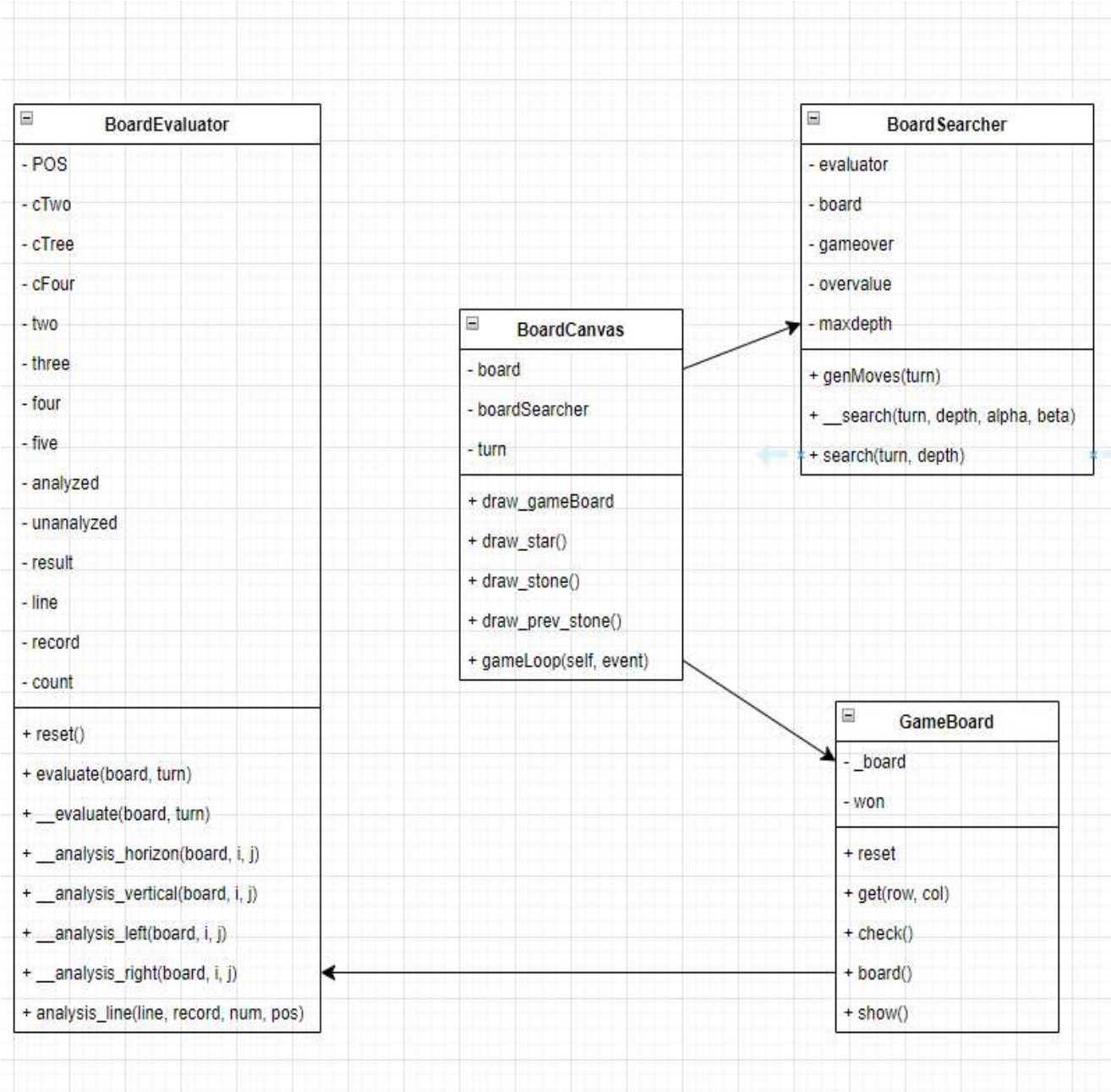


Рисунок Г.2 – Діаграма класів

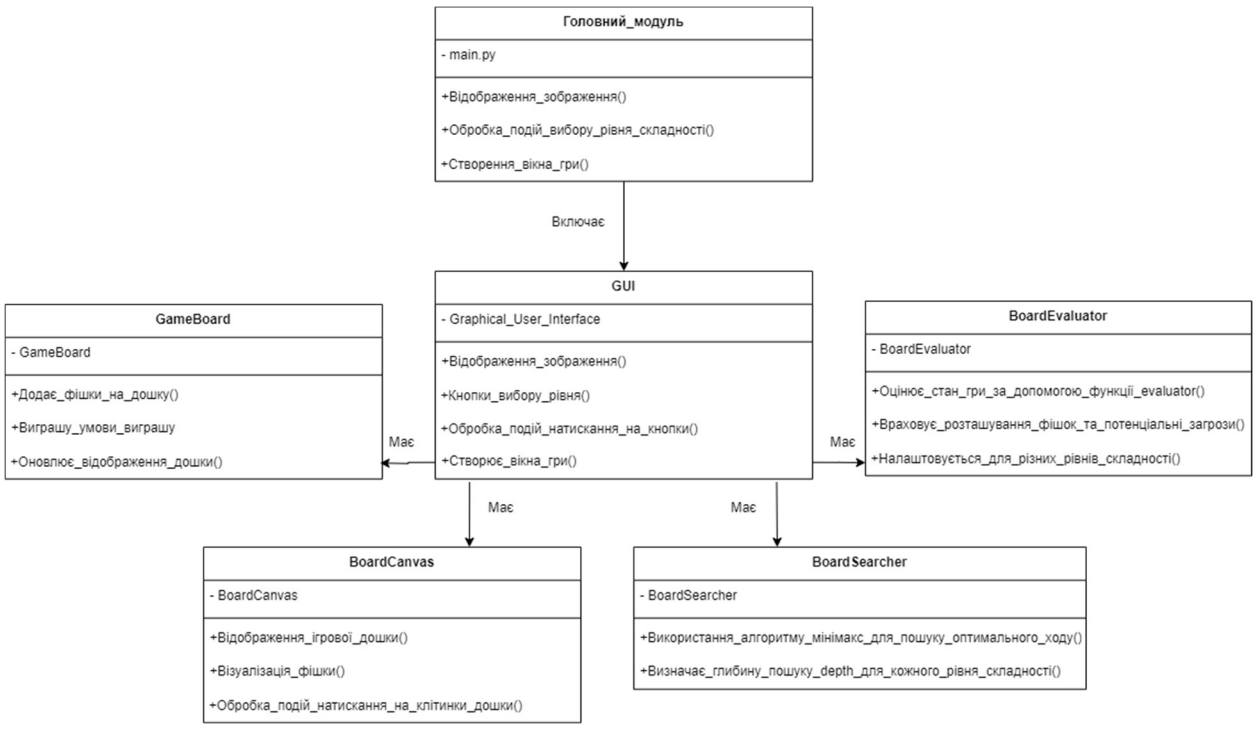


Рисунок Г.3 – Основна структурна схема програми

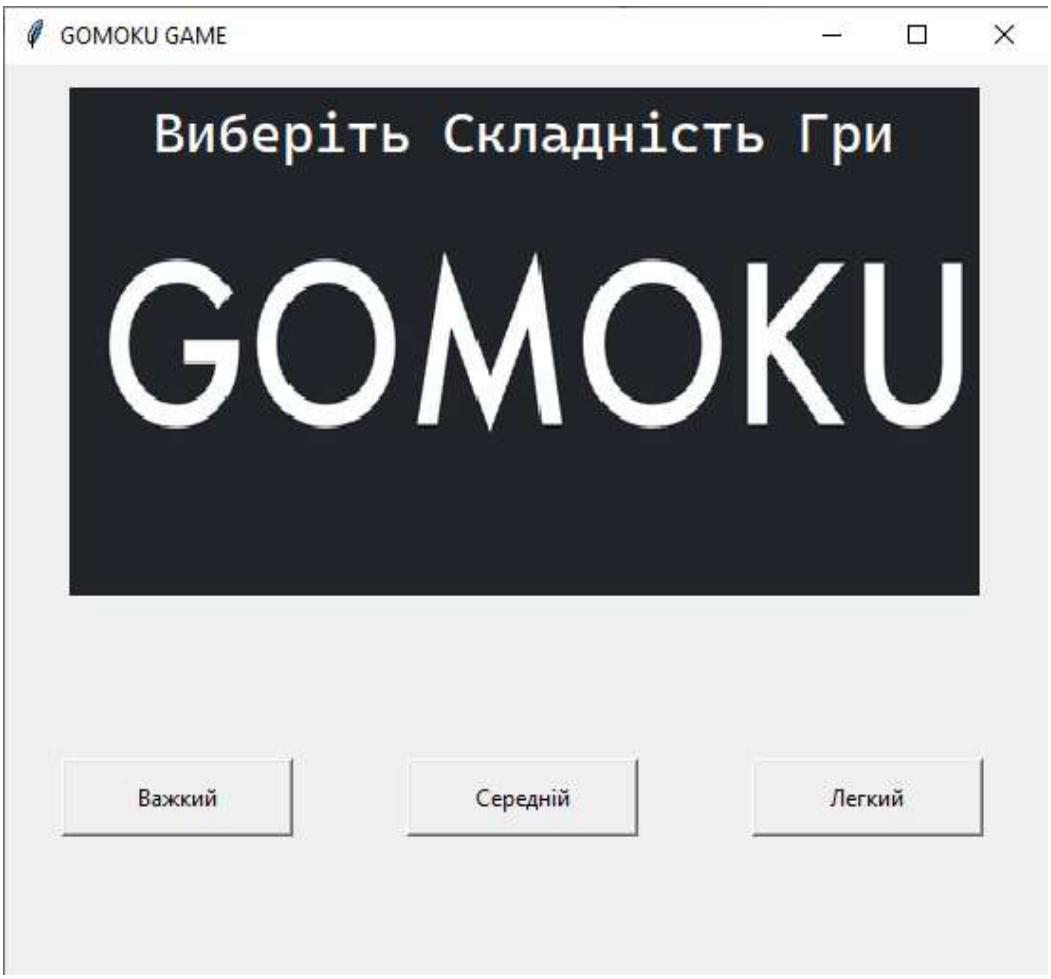


Рисунок Г.4 – Вибір складності гри

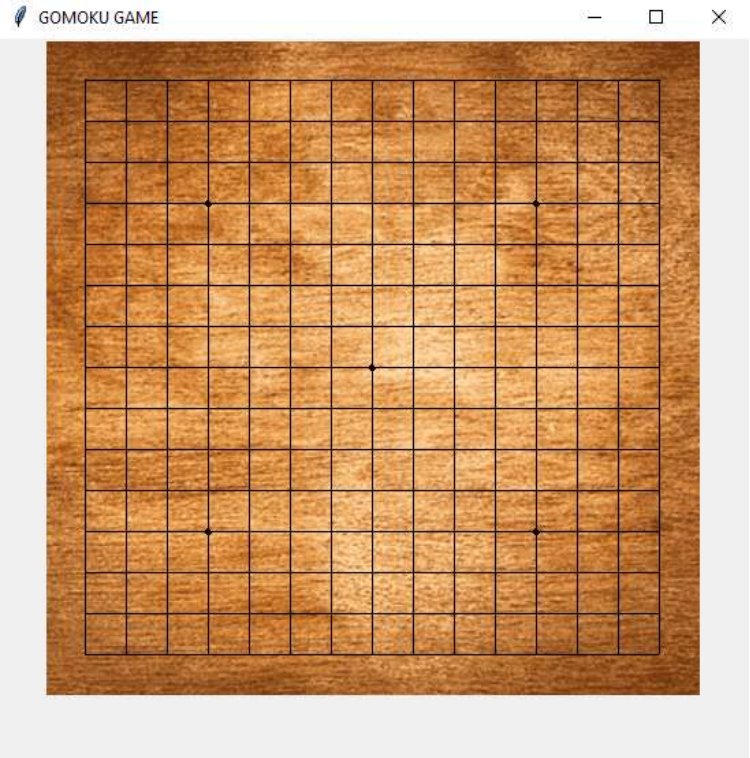


Рисунок Г.5 – Графічний інтерфейс програми

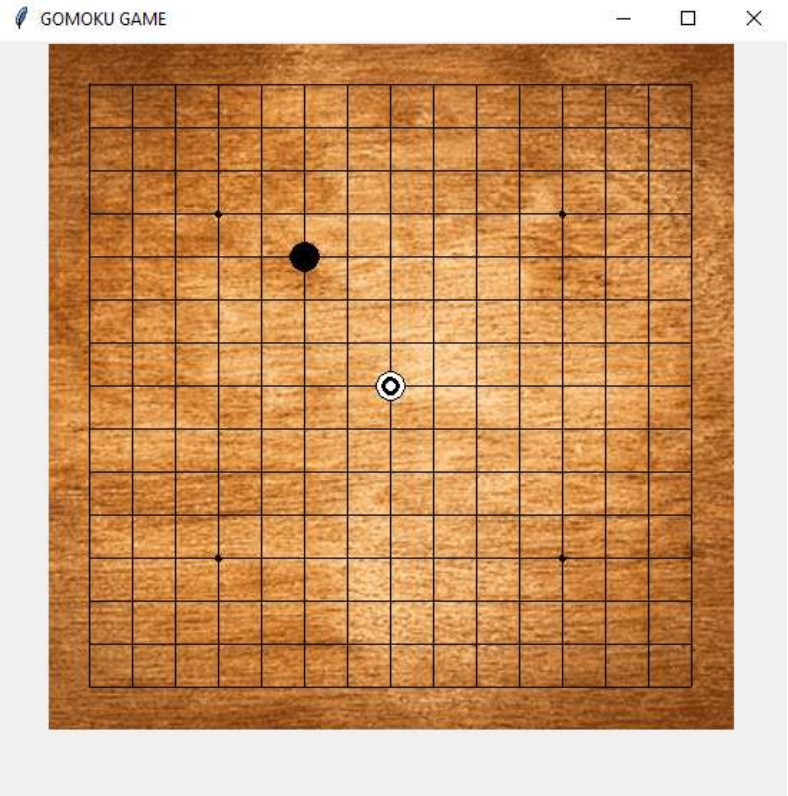


Рисунок Г.6 – Приклад першого ходу

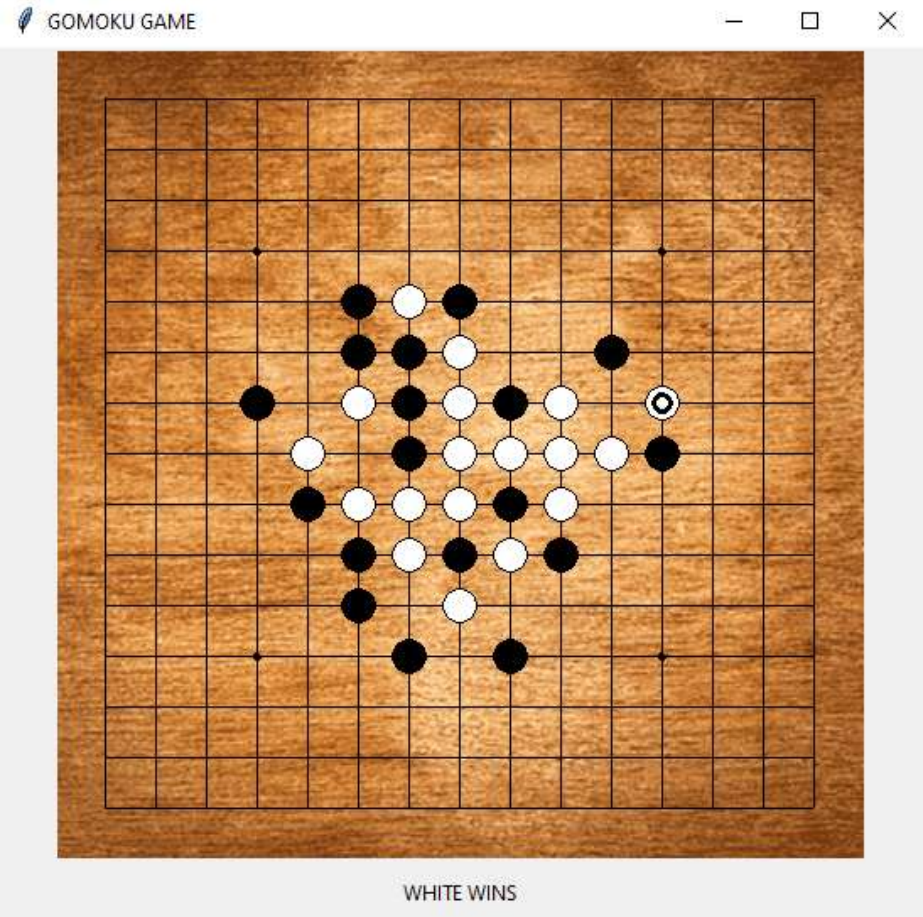


Рисунок Г.7 – Перемога біли