

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота

з дисципліни «Технології паралельних обчислень»

Тема: Розробка мультипотокового Алгоритму Minimax та порівняння з його базовою версією

|  |  |
| --- | --- |
| **Керівник**:  Дифучина О.Ю.  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2023 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Терешкович М. О.  студент групи ІТ–02  залікова книжка № ІТ–0223  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «23» травня 2023 р.  Інна СТЕЦЕНКО  Олександра ДИФУЧИНА |

**Київ – 2023**

**Міністерство освіти та науки України**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

Кафедра Інформатики та програмної інженерії

Дисципліна Технології паралельних обчислень

Спеціальність Інженерія програмного забезпечення

Курс ІII Група IT-02 Семестр 6

**ЗАВДАННЯ**

**на курсову роботу студента**

Терешковича Максима Олександровича

(прізвище, ім’я, по батькові)

**1.** **Тема роботи** Розробка мультипотокового Алгоритму Minimax та порівняння з його базовою версією реалізований мовою С#.

**2.** **Термін здачі студентом закінченої роботи** « 23 » травня 2023 р.

**3. Зміст розрахунково-пояснювальної записки** (перелік питань, які підлягають розробці): зміст, реферат, ключові слова, вступ, розділ 1 (опис алгоритму та його відомих паралельних реалізацій), розділ 2 (розробка послідовного алгоритму та аналіз його швидкодії), розділ 3 (вибір програмного забезпечення для розробки паралельних обчислень та його короткий опис), розділ 4 (розробка паралельних обчислень алгоритму з використанням обраного програмного забезпечення: проектування, реалізація, тестування), розділ 5 (дослідження ефективності паралельних обчислень алгоритму (порівняльний аналіз швидкості обчислень)), висновки, список використаних джерел, додатки (результати тестування послідовного алгоритму, повний лістинг програми, результати тестування паралельного алгоритму).

**4. Перелік графічного матеріалу** (з точним зазначенням обов’язкових креслень)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**РЕФЕРАТ**

В рамках курсової роботи був спроектований та реалізований базовий алгоритм Мінімакс. Базуючись на базовому алгоритмі була розроблена модифікація даного алгоритму використовуючи механізм мультипотокового програмування.

Було виконане тестування паралельного алгоритму щоб довести коректність результатів обчислень. Було виконане дослідження швидкодії алгоритму при зростанні кількості даних для обчислень. А також в рамках роботи виконане експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень.

Для реалізації алгоритму була обрана мова С#. Вона є потужною інструментальною платформою, яка надає розробникам доступ до багатьох інструментів та бібліотек для створення високоякісних додатків. C# є однією з найбільш популярних мов програмування для розробки ігор, тому вона може бути хорошим вибором для реалізації алгоритму Мінімакс для ігор.

Переваги мови C# при реалізації алгоритму Мінімакс можуть включати:

1. Об'єктно-орієнтований підхід: C# є мовою, що підтримує об'єктно-орієнтований підхід, що дозволяє більш ефективно працювати зі складними об'єктами, які використовуються в алгоритмі Мінімакс.
2. Бібліотеки: C# має вбудовані бібліотеки, такі як System.Collections.Generic, які дозволяють працювати зі списками, стеками та іншими колекціями даних, що можуть бути корисними для реалізації алгоритму Мінімакс.
3. Простота: C# має простий синтаксис, який дозволяє швидко розробляти програми. Це може бути особливо корисним, коли потрібно розробити алгоритм Мінімакс для невеликих ігор.

Курсова робото проводилася на ноутбуці Asus TUF Gaming A17 FA706IU-H7023 (90NR03K1-M02580) Fortress Gray. Основні характеристики:

* Процесор - Восьмиядерний AMD Ryzen 9 4900H (3.3 - 4.4 ГГц)
* Об'єм оперативної пам'яті - 16 ГБ
* Тип оперативної пам'яті - DDR4
* Характеристики оперативної пам'яті - DDR4-3200 МГц

Ключові слова: C#, MiniMax, parallel algorithms, прискорення.

**ЗМІСТ**

[**ВСТУП** 5](#_Toc134368408)

[**1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ** 7](#_Toc134368409)

[**1.1 Цілі та загальна характеристика** 7](#_Toc134368410)

[**1.2** **Розрахунок характеристик** 9](#_Toc134368411)

[**1.3 Паралельна версія алгоритму(загальна характеристика)** 9](#_Toc134368412)

[**2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ** 12](#_Toc134368413)

[**2.1 Базовий алгоритм МініМаксу** 12](#_Toc134368414)

[**2.2 Додатковий код** 13](#_Toc134368415)

[**2.3 Результати роботи та вимірювання** 15](#_Toc134368416)

[**3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС** 17](#_Toc134368417)

[**3.1 Аналіз існуючих програмних рішень.** 17](#_Toc134368418)

[**4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ** 18](#_Toc134368419)

[**4.1 Проектування** 18](#_Toc134368420)

[**4.2 Реалізація** 18](#_Toc134368421)

[**4.3 Результати роботи та вимірювання** 20](#_Toc134368422)

[**5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ** 22](#_Toc134368423)

[**ВИСНОВКИ** 26](#_Toc134368424)

[**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** 27](#_Toc134368425)

[**ДОДАТКИ** 28](#_Toc134368426)

[**Додаток А. Лістинг коду** 28](#_Toc134368427)

[**Файл Program.cs:** 28](#_Toc134368428)

[**Файл Tree.cs** 30](#_Toc134368429)

[**Файл Node.cs:** 31](#_Toc134368430)

# **ВСТУП**

Розробка ефективних алгоритмів для вирішення складних завдань є однією з ключових задач сучасної інформаційної технології. Алгоритм МініМакс - це один з найбільш популярних алгоритмів, який використовується для прийняття рішень в іграх, таких як шахи, го, хрестики нулики та інші. Обчислення даної курсової роботи будуть проводитись на восьми-ядерному процесорі.

Однак, якщо обчислювати всі можливі ходи та їх наслідки в грі, це може зайняти дуже багато часу, тому розробка ефективного алгоритму для прийняття рішень є важливим завданням для інформаційних технологій.

В цьому рефераті розглядається розробка та порівняння базового та мультипотокового алгоритму Мінімакс на мові програмування С#. Базовий алгоритм Мінімакс буде реалізовано без використання мультипотоковості, а мультипотоковий алгоритм буде розроблено з використанням паралельного програмування.

Паралельні обчислення виконуються за наступним принципом:

* задача розбивається на підзадачі для виконання одночасно
* кожна підзадача в свою чергу розбивається на послідовність інструкцій;
* інструкції також виконуються одночасно на різних процесорах;

Розглянемо різні підходи до реалізації алгоритму МініМакс та порівняємо їх ефективність. Основним критерієм оцінки роботи алгоритма та успішності виконаної роботи буде «Прискорення». Мультипотокова версія повинна бути мінімум в 1.2 швидша за базову але ,як відомо, при невеликому об’ємі даних базовий алгоритм завжди буде швидшим – тому потрібно буде визначити основні параметри та значення при яких дана умова буде виконуватися успішно.

Метою цього дослідження є розробка більш ефективного алгоритму МініМакс з використанням мультипотоковості та порівняння його з базовим алгоритмом без мультипотоковості для визначення його переваг та недоліків

Основні завдання, які слід виконати в курсовій роботі для розробки та порівняння базового та мультипотокового алгоритму МініМакс:

1. Розробити базовий алгоритм МініМакс на мові С#.
2. Розробити мультипотоковий алгоритм МініМакс на мові С# з використанням паралельного програмування.
3. Провести тестування розроблених алгоритмів на різних даних для порівняння їх ефективності.
4. Дати висновки про переваги та недоліки кожного алгоритму та порівняти їх ефективність.

# **1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ**

## **Цілі та загальна характеристика**

Основною метою послідовного алгоритму МініМакс є знаходження найкращого ходу у грі з двома гравцями, де кожен гравець старається максимізувати свій виграш, або мінімізувати виграш супротивника.

Загальні характеристики послідовного алгоритму МініМакс можуть бути наступними:

1. Послідовний алгоритм МініМакс вимагає перебору всіх можливих ходів у грі, що може зайняти значну кількість часу.
2. Для кожного можливого ходу у грі, алгоритм розраховує очікуваний виграш для гравця, який веде гру, за умови, що обидва гравці грають оптимально.
3. Алгоритм обчислює дерево гри, де кожен вузол представляє можливу позицію у грі, а кожна дуга виходить з вузла і представляє можливий хід гравця.
4. Після розрахунку очікуваного виграшу для кожного можливого ходу, алгоритм вибирає хід, що максимізує очікуваний виграш для гравця, який веде гру.
5. За умови, що гра має скінченну кількість ходів та ігри на дошці не має неперервних стратегій, послідовний алгоритм МініМакс завжди знайде оптимальний хід для гравця, який веде гру.

Псевдокод для базового МініМакс:

A picture containing text, screenshot, font

Description automatically generatedРис. 1.1: Псевдокод базового МініМаксу.

Цей псевдокод приймає на вхід поточний стан гри, параметр глибини та параметр гравця. Вона повертає найкраще значення, яке можна отримати з поточного стану для даного гравця.

Функція починає роботу з перевірки того, чи досягнуто максимальної глибини, або чи завершено гру. У будь-якому з цих випадків функція повертає оцінку поточного стану.

Якщо поточний гравець максимізує, функція ініціалізує найкраще значення від'ємною нескінченністю, а потім ітерує всі дочірні стани поточного стану. Для кожного дочірнього стану рекурсивно викликається функція МініМаксу з глибиною, зменшеною на 1, і іншим гравцем як поточним гравцем. Найкраще значення оновлюється як максимальне значення між поточним найкращим значенням і значенням, що повертається рекурсивним викликом.

Якщо поточний гравець мінімізує, функція ініціалізує найкраще значення додатною нескінченністю, а потім ітераційно проходить через усі дочірні стани поточного стану. Для кожного дочірнього стану рекурсивно викликається функція мінімаксу з глибиною, зменшеною на 1, і іншим гравцем як поточним гравцем. Найкраще значення оновлюється як мінімальне значення між поточним найкращим значенням і значенням, що повертається рекурсивним викликом.

## **Розрахунок характеристик**

Для того, щоб Мінімакс міг приймати рішення про хід у грі, необхідно розрахувати певні характеристики (або оцінки) кожного стану гри. Наприклад, в шахах можна використовувати кількість фігур на дошці, позиційні переваги, розташування короля, тощо. У кожній грі характеристики можуть бути різними.

Розрахунок характеристик зазвичай відбувається на основі евристичних правил, тобто на основі досвіду та інтуїції програміста або експерта в грі. Наприклад, у грі в хрестики-нулики можна використовувати такі правила:

* Кожен вільний кут на дошці збільшує шанс перемоги гравця Х.
* Кожен рядок, колонка або діагональ з двома фігурами гравця Х і вільною клітинкою збільшує його шанс на перемогу.
* Кожен рядок, колонка або діагональ з двома фігурами гравця О і вільною клітинкою зменшує шанси гравця Х на перемогу.

Зазвичай оцінка кожного стану гри залежить від рівня глибини дерева, яке будується МініМаксом. Чим глибше дерево, тим точніша оцінка стану гри. Проте, підвищення глибини дерева також призводить до збільшення часу виконання алгоритму.

## **1.3 Паралельна версія алгоритму(загальна характеристика)**

Паралельна версія алгоритму МініМакс заснована на розділенні дерева гри між декількома процесами або потоками, кожен з яких оброблює окрему гілку дерева. Після того, як кожен процес або потік закінчив обробку своєї гілки дерева, результати об'єднуються, і знаходиться оптимальний хід для гравця, який веде гру. Звісно це завжди не оптимальний варіанти, тому що кожен проект, гра або штучний інтелект потребує власних правок та доопрацювань – то завжди треба зважати на ці аспекти при роботі з багатопоточністю.

Основний метод паралельного алгоритму МініМакс може бути записана наступним чином:

де:

* node - поточний вузол дерева гри
* depth - глибина дерева гри
* is\_maximizing\_player - параметр, який вказує, чи максимізує гравець, який веде гру, свій виграш

У паралельній версії алгоритму МініМакс кожен процес або потік обробляє свою гілку дерева гри та повертає результат, який містить оптимальний хід для даної гілки. Після того, як всі результати були отримані, об'єднуються, і оптимальний хід для всього дерева гри визначається на основі результатів кожної гілки. Якщо говорити простими словами – то всі ноди корінної ноди запускаються паралельно, з яких потім обирається оптимальний результат.

Псевдокод для паралельного МініМакс:  
A picture containing text, screenshot, font, display

Description automatically generated

Рис. 1.2: Псевдокод паралельного мінімаксу.

Цей псевдокод приймає майже все те й саме що і базова версія, проте замість звичайного мінімакса – викликається паралельний та обраховуються всі дочерні ноди оригінального вузла.

# **2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ**

## **2.1 Базовий алгоритм МініМаксу**

**Text

Description automatically generated**В даному розділі наведений алгоритм МініМаксу реалізований мовою С#. Для реалізації алгоритму МініМакс (надалі може вживатись як просто “алгоритм”) був написаний метод Minimax, який приймає на вхід глибину дерева, ноду та булевий параметр (який відповідає за те чи максимізує/мінімізує гравець) показано на рисунку 2.1.

Рис. 2.1 – код базової версії МініМакс

Про метод GetResult буде інформація у 4 розділі.

Даний алгоритм працює за таким принципом:

1. Отримуємо на вхід дані про дерево – це як я вже зазначав: глибина дерева, саму ноду і булеве значення, яке відповідає за максимізацію/мінімізацію на даному кроці.
2. Якщо глибина пошуку дорівнює нулю або вузол не має дочірніх вузлів, то повертається значення вузла (це оцінка позиції для гравця).
3. Інакше, ініціалізується змінна result, яка буде зберігати краще значення для поточного гравця. Якщо гравець максимізує свою оцінку, то result приймає мінімальне можливе значення (int.MinValue), а якщо мінімізує - максимальне (int.MaxValue).
4. Для кожного дочірнього вузла рекурсивно викликається функція Minimax з протилежним значенням isMaximizingPlayer і глибиною, зменшеною на одиницю. Таким чином, симулюється хід противника.
5. За допомогою функції GetResult порівнюється значення result із значенням дочірнього вузла (childValue) і обирається краще для поточного гравця. Якщо гравець максимізує свою оцінку, то обирається більше значення, а якщо мінімізує - менше.
6. Повертається краще значення для поточного гравця (result).

## **2.2 Додатковий код**

Text

Description automatically generatedТепер поговоримо про саме дерево, ноди так як вони влаштовані. В програмі присутні два класи це: Node та Tree дивіться рисунок 2.2 і 2.3. Для того щоб показати коректність роботи алгоритму та не створювати для цього якусь гру або додаткову програму – було створено бінарне дерево. На початку програми вас просять задати глибину дерева та кількість нод на вузлі. Далі дерево генерується з вказаними параметрами, а значення на вузлах обирається рандомно від 1 до 100.

Рис. 2.2 – клас Node

Даний код цього класу працює таким чином:

1. Визначається властивість Value, яка зберігає значення вузла. Вона має модифікатор доступу private set, що означає, що її можна змінювати тільки всередині класу.
2. Визначається властивість Children, яка зберігає список дочірніх вузлів. Вона також має модифікатор доступу private set.
3. Визначається конструктор класу Node, який приймає параметр Value.

Text

Description automatically generatedТаким чином, цей код створює клас Node, який можна використовувати для побудови дерева що буде продемонстровано в подальшому.

Рис. 2.3 – клас Tree та його методи.

Основна задача даного класу – це створити дерево та інформацію для нього.

Послідовність дій така:

1. Визначається властивість Root, яка зберігає кореневий вузол дерева.
2. Визначається конструктор класу Tree, який приймає параметри depth і numChildren. Конструктор створює новий об’єкт класу Random і викликає приватний метод GenerateTree з параметрами depth, numChildren і random. Результатом цього методу є кореневий вузол дерева, який присвоюється властивості Root.
3. Визначається приватний метод GenerateTree, який приймає параметри depth, numChildren і random. Метод робить наступне:

* Генерує випадкове число від 1 до 100 і присвоює його змінній value.
* Створює новий об’єкт класу Node з параметром value і присвоює його змінній node.
* Якщо глибина дерева більша за нуль, то виконується цикл for numChildren разів. На кожній ітерації циклу рекурсивно викликається метод GenerateTree з параметрами depth - 1, numChildren і random. Результатом цього методу є дочірній вузол, який додається до списку Children вузла node.
* Повертається вузол node.

## **2.3 Результати роботи та вимірювання**

Було проведено декілька експериментів з різним набором даних, з різною кількістю нод та глибиною дерева. Результати були записані у таблицю та лінійний графік (дивіться рисунок (2.4 та 2.5). *Час витрачений на дослідження вимірювався у мілісекундах.*

Table, calendar

Description automatically generatedРис. 2.4 – Базовий МініМакс та витрачений час

Рис. 2.5 – Лінійний графік показує затрачений час на певну нод і глибину

Хотілось би зауважити, що як Ви бачите, не всі клітинки в таблиці були заповнені по тій причині, що об’єм даних був надто великим і критично тяжким для мого ноутбука. Наприклад вже при глибині 15 та з 3 нодами на вузлі = маємо понад 21 млн нод, а саме 21 523 360, або при глибині 25 і 2 нодами на вузлі = маємо 67 108 863. Тому в подальших дослідженнях більшого об’єму не бачу сенсу, так як в кінці звіту будуть підбиті підсумки і продемонстровано, що код має прискорення мінімум в 1.2 рази. Також головним аспектом є ще те, що програма дуже довго генерує само дерево, тому при величезних об’ємах даних просто не доходить до самих алгоритмів.

# **3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС**

## **3.1 Аналіз існуючих програмних рішень.**

Для реалізації курсової роботи була вибрана мова програмування C# її детальні переваги у цьому проекті були наведені на сторінці 3. Але основним критерієм вибору було те, що я більше за все розбираюся в даній мові програмування. Також я обрав Microsoft Visual Studio, яка є хорошим варіантом, оскільки це інтегроване середовище розробки, спеціально призначене для роботи з C# та .NET.

Visual Studio має вбудовані засоби для підтримки паралельного програмування, зокрема, можливість створення мультипотокових програм з використанням бібліотеки Parallel Extensions[[4]](#_СПИСОК_ВИКОРИСТАНИХ_ДЖЕРЕЛ). Ця бібліотека містить в собі набір інструментів для розпаралелювання обчислень на рівні коду, що дозволяє покращити продуктивність програми.

Крім того, Visual Studio має можливості для профілювання програми, що дозволяє виявляти та оптимізувати проблеми або бутленеки в коді, що може бути корисним при розробці паралельних обчислень.

# **4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ**

## **4.1 Проектування**

Для того, щоб створити паралельну реалізацію алгоритму МініМакс, як i будь-якого іншого алгоритму, спочатку необхідно визначитися, як правильно розбивати задачу на підзадачі, так щоб виконувати ці підзадачі паралельно. Так як у нашого батьківського вузла є список з дочірніх вузлів, можна розбити нашу задачу на підзадачі, де метою кожної з них буде обчислення найбільш неоптимального значення вузла та вже по результатам всіх підзадач вибрати найбільш оптимальний варіант.

## **4.2 Реалізація**

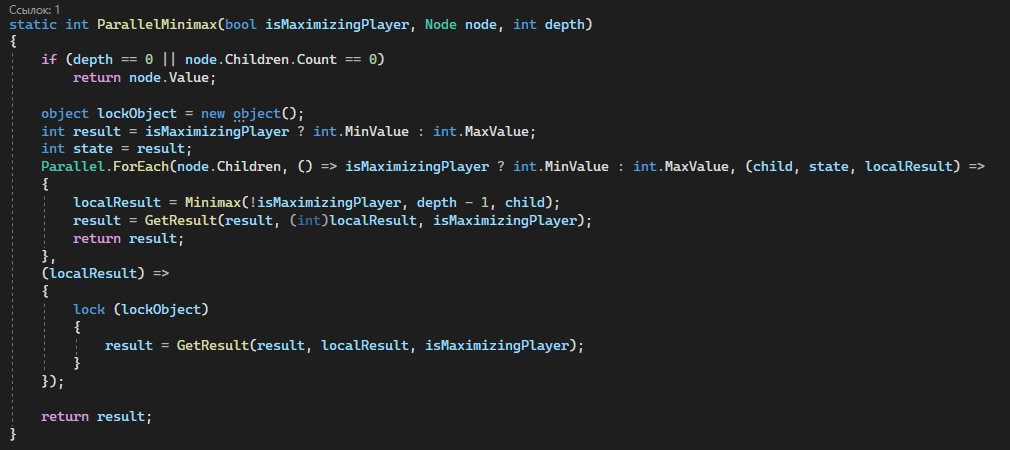
Після того як ми визначилися, що потрібно для створення паралельної версії алгоритму – можна переходити до реалізації. (Рис. 4.1) та (Рис. 4.2)

Рис. 4.1 – Паралельна реалізація алгоритму МініМакс

Graphical user interface, text

Description automatically generatedРис. 4.2 – Реалізація методу GetResult

Для того, щоб паралельно виконати підзадачі було вирішено використати клас Parallel з бібліотеки TPL[[1]](#_СПИСОК_ВИКОРИСТАНИХ_ДЖЕРЕЛ), а саме метод ForEach[[2]](#_СПИСОК_ВИКОРИСТАНИХ_ДЖЕРЕЛ), який здійснює ітерацію по колекції, подібно до циклу foreach, тільки він, на відміну від звичайного foreach, здійснює паралельне виконання перебору. Якщо говорити про повторне використання потоків - це механізм, при якому потоки повторно використовуються для обробки декількох завдань замість створення нового потоку для кожного завдання. У данному методі для кожного дочірнього вузла створюється новий потік, тому немає жодних підстав вважати, що потоки можуть бути повторно використані для обробки декількох вузлів, 1 вузл = 1 потік,не більше не менше. Щодо крадіжки завдань - це механізм, за допомогою якого зайнятий потік може "вкрасти" завдання у простоюючого потоку, щоб бути зайнятим. У наведеній реалізації методу ParallelMinimax немає крадіжки завдань, оскільки метод Parallel.ForEach використовується з опцією балансування навантаження за замовчуванням PartitionerOption.NoBuffering. Це означає, що робота статично розподіляється між доступними потоками, і кожен потік обробляє призначену йому частину роботи, не намагаючись вкрасти роботу у інших потоків. Таким чином, у цій реалізації відсутня крадіжка завдань. Першим параметром методу ForEach є набір елементів по якому буде виконаний перебір. Другим параметром є функція яка ініціалізує значення для локальної змінної, яка буде виконуватися один раз для кожного потоку. Третій параметр це функція яка викликається циклом на кожній ітерації. Четвертий параметр це також функція, яка виконується один раз для кожного потоку, враховуючи кінцевий результат, отриманий у цьому потоці.

Для кожного дочірнього вузла виконується паралельне обчислення, як тільки якийсь з потоків закінчив виконувати свою підзадачу, тоді результат виконання цієї підзадачі переходить до методу GetResult, де ми порівнюємо отримане значення з базовим та в залежності від цього присвоюємо значення для змінної result. Також перед тим, як буде виконаний метод GetResult ми синхронізуємо доступ до нього за допомогою конструкції lock, щоб запобігти одночасному виконанню, що може призвести до неочікуваного результату паралельного алгоритму. Після того як ForEach завершив своє виконання, повертаємо отриманий результат.

## **4.3 Результати роботи та вимірювання**

Нижче наведені результати роботи мультипоточної версії алгоритму МініМакс. *Всі вимірювання відбувалися в мілісекундах.* Дивіться рисунок 4.3 та 4.4.

A screenshot of a computer

Description automatically generated with medium confidenceРис. 4.3 – Мультпотоковий Мінімакс та витрачений час

Рис. 4.4 – Лінійний графік показує затрачений час на певну кількість потоків і глибину

# **5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ**

Після демонстрації роботи кожного з алгоритмів – час підпити підсумки та підрахувати прискорення кожного з експерементів. Курсова робото проводилася на ноутбуці Asus TUF Gaming A17 FA706IU-H7023 (90NR03K1-M02580) Fortress Gray. Основні характеристики:

* Процесор - Восьмиядерний AMD Ryzen 9 4900H (3.3 - 4.4 ГГц)
* Об'єм оперативної пам'яті - 16 ГБ
* Тип оперативної пам'яті - DDR4
* Характеристики оперативної пам'яті - DDR4-3200 МГц

Table

Description automatically generatedТак, як у нас найбільша частина даних при 2 нодах на вузл, зробимо порівняння спочатку для них. Дивіться Рис. 5.1 і 5.2.

Рис. 5.1 – Порівняння часу обох алгоритмів для певної глибини та 2 нодів на вузл

Рис. 5.2 – Графік залежності часу від глибини та 2 нод на вузл

Тепер поглянемо на загальне прискорення для всіх експериментів. Результати наведені на Рис. 5.3 і 5.4.

Рис. 5.3 – Результат ділення часу для базового алгоритму на мультипоточний – «Прискорення»

Table

Description automatically generated

Рис. 5.4 – Таблиця «Прискорення»

Text

Description automatically generated

Text

Description automatically generated

Рисунок 5.3. – Результати виконання програми у консолі

Проаналізувавши всі графіки та таблиці можна прийти до декількох висновків. Швидкодія базового алгоритму Мінімакс та мультипотокової версії залежить від декількох факторів. У базової версії Мінімакс, час виконання залежить від глибини дерева та кількості нащадків кожного вузла. Якщо глибина дерева та кількість нащадків збільшується, час виконання також збільшується. Тому, якщо ми маємо дерево з великою глибиною та багатьма нащадками, то базова версія Мінімакс може працювати дуже повільно. У мультипотокової версії Мінімакс, час виконання залежить від кількості доступних потоків та ефективності розподілення роботи між ними. Якщо кількість доступних потоків збільшується, то можна очікувати покращення швидкодії мультипотокової версії МініМакс. Керування потоками зазвичай здійснюється використовуючи бібліотеки мов програмування, такі як .NET Framework, яка була використана в даному прикладі. Ці бібліотеки надають різні інструменти для керування потоками, такі як створення, запуск, призупинення та закінчення потоків. Розподіл роботи між потоками може здійснюватися різними способами, наприклад, за допомогою паралельних конструкцій, таких як Parallel.ForEach, яка була використана в даному прикладі. Ця конструкція автоматично розподіляє роботу між доступними потоками, дозволяючи ефективніше використовувати ресурси системи. Однак, якщо розподілення роботи між потоками не ефективне, то можна очікувати незначне покращення швидкодії або навіть погіршення її.

В нашому випадку ми зрозуміли що базова версія МініМаксу працює швидше лише при відносно не великій глибині дерева та кількості нодів. Оптимальним значення при якому мультипотокова версія алгоритму починає працювати швидше – це глибина 20 та 2 нащадки (приблизно 2 млн нод), глибина 15 і 3 нащадки (приблизно 21 млн нод) або глибина 10 та 5 нащадків (приблизно 12 млн нод).

# **ВИСНОВКИ**

В даній курсовій роботі було реалізовано базовий та паралельний алгоритм МініМаксу. Основним критерієм розпаралелення було те, що всі ноди корінної ноди запускаються паралельно, з яких потім обирається оптимальний результат. Паралельна реалізація була зроблена шляхом використання Parallel.ForEach та локуванням об’єктів.

Були проведені експеременти, підраховано час та зроблено порівняння двох алгоритмів. МініМакс це алгоритмом штучного інтелекту, що використовується для прийняття рішень у гральних задачах, таких як шахи, го, покер тощо.

Базовий алгоритм МініМакс працює за принципом рекурсивного спуску в дереві гри, де кожен вузол представляє можливу гру до кінця. Алгоритм Мінімакс виконує повний перебір всіх можливих ходів і прогнозує виграш або програш для кожного з них. Потім знаходиться оптимальний хід, що максимізує виграш гравця, якщо він максимізатор, або мінімізує його втрати, якщо він мінімізатор.

Мультипоточна версія алгоритму МініМакс може працювати набагато швидше, ніж базовий алгоритм, але потребує додаткових зусиль для розподілу обчислень між потоками та управління конкуренцією за ресурси. Шляхом експериментів було виявлено, те що мультипотокова версія починає працювати швидше при глибині дерева 20 і 2 нодах, 15 і 3 нодах, 10 та 5 нодах. Максимальне отримане прискорення складає в 3.4 рази в порівнянні з базовою версією алгоритму.

Отже, базовий алгоритм МініМакс простий та надійний, але може бути повільним при великих об’ємах даних. Мультипоточна версія алгоритму МініМакс може прискорити час виконання, але для цього потрібно ретельно розібрати алгоритм та дізнатися які аспекти впливають на його роботу, такі як глибина, кількість нод евристики і тд. Вибір між базовим та мультипоточним алгоритмом МініМакс залежить від конкретної гри та обмежень обчислювальних ресурсів.

# **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>
2. <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library>
3. <http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/rocki_ppam09.pdf>
4. <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/>

# **ДОДАТКИ**

## **Додаток А. Лістинг коду**

Посилання на проект: <https://github.com/mcmcmax437/CourseWork---MiniMax>

## **Файл Program.cs:**

using MiniMax\_MulthiThread\_vs\_Basic.Class;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Threading.Tasks;

namespace MiniMax\_MulthiThread\_vs\_Basic

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter tree depth:");

int depth = int.Parse(Console.ReadLine());

Console.WriteLine("Enter number of children per node:");

int numChildren = int.Parse(Console.ReadLine());

Tree tree = new Tree(depth, numChildren);

int numNodes = CountNodes(tree.Root);

Console.WriteLine($"Number of nodes in the tree: {numNodes}");

Console.WriteLine("Starting basic minimax...");

Stopwatch basicStopwatch = Stopwatch.StartNew();

int basicResult = Minimax(true, depth, tree.Root);

basicStopwatch.Stop();

Console.WriteLine($"Basic minimax result: {basicResult}");

Console.WriteLine($"Time taken: {basicStopwatch.ElapsedMilliseconds}ms");

Console.WriteLine("Starting parallel/multithreaded minimax...");

Stopwatch parallelStopwatch = Stopwatch.StartNew();

int parallelResult = ParallelMinimax(true, tree.Root, depth);

parallelStopwatch.Stop();

Console.WriteLine($"Parallel/multithreaded minimax result: {parallelResult}");

Console.WriteLine($"Time taken: {parallelStopwatch.ElapsedMilliseconds}ms");

float parallTIME = parallelStopwatch.ElapsedMilliseconds;

float basicTIME = basicStopwatch.ElapsedMilliseconds;

float res = basicTIME / parallTIME;

Console.WriteLine($"Speed Up = {basicStopwatch.ElapsedMilliseconds} / {parallelStopwatch.ElapsedMilliseconds} = {res}");

Console.ReadLine();

}

static int ParallelMinimax(bool isMaximizingPlayer, Node node, int depth)

{

if (depth == 0 || node.Children.Count == 0)

return node.Value;

object lockObject = new object();

int result = isMaximizingPlayer ? int.MinValue : int.MaxValue;

int state = result;

Parallel.ForEach(node.Children, () => isMaximizingPlayer ? int.MinValue : int.MaxValue, (child, state, localResult) =>

{

localResult = Minimax(!isMaximizingPlayer, depth - 1, child);

result = GetResult(result, (int)localResult, isMaximizingPlayer);

return result;

},

(localResult) =>

{

lock (lockObject)

{

result = GetResult(result, localResult, isMaximizingPlayer);

}

});

return result;

}

static int Minimax(bool isMaximizingPlayer, int depth, Node node)

{

if (depth == 0 || node.Children.Count == 0)

return node.Value;

int result = isMaximizingPlayer ? int.MinValue : int.MaxValue;

foreach (Node child in node.Children)

{

int childValue = Minimax(!isMaximizingPlayer, depth - 1, child);

result = GetResult(result, childValue, isMaximizingPlayer);

}

return result;

}

static int GetResult(int oldVa1ue, int newValue, bool isMaximizingPlayer)

{

return (isMaximizingPlayer && newValue > oldVa1ue) || (!isMaximizingPlayer && newValue < oldVa1ue)

? newValue

: oldVa1ue;

}

static int CountNodes(Node node)

{

int count = 1;

foreach (Node child in node.Children)

{

count += CountNodes(child);

}

return count;

}

}

}

## **Файл Tree.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

namespace MiniMax\_MulthiThread\_vs\_Basic.Class

{

class Tree

{

public Node Root { get; private set; }

public Tree(int depth, int numChildren)

{

Random random = new Random();

Root = GenerateTree(depth, numChildren, random);

}

private Node GenerateTree(int depth, int numChildren, Random random)

{

int value = random.Next(1, 100);

Node node = new Node(value);

if (depth > 0)

{

for (int i = 0; i < numChildren; i++)

{

node.Children.Add(GenerateTree(depth - 1, numChildren, random));

}

}

return node;

}

}

}

## **Файл Node.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

namespace MiniMax\_MulthiThread\_vs\_Basic.Class

{

class Tree

{

public Node Root { get; private set; }

public Tree(int depth, int numChildren)

{

Random random = new Random();

Root = GenerateTree(depth, numChildren, random);

}

private Node GenerateTree(int depth, int numChildren, Random random)

{

int value = random.Next(1, 100);

Node node = new Node(value);

if (depth > 0)

{

for (int i = 0; i < numChildren; i++)

{

node.Children.Add(GenerateTree(depth - 1, numChildren, random));

}

}

return node;

}

}

}