**ЗМІСТ**

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 3

ВСТУП 4

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ СИСТЕМ ПОШУКУ У СТРУКТУРАХ ДАНИХ ГРИ ГО 7

1.1. Огляд існуючих програмних продуктів для пошуку 7

1.2. Огляд існуючих алгоритмів пошуку у структурах даних гри Го 12

2. ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ 26

2.2. Вибір платформи 26

2.2. Вибір мови програмування 37

3. РОЗРОБКА СИСТЕМИ ПАРАЛЕЛЬНОГО ПОШУКУ У СТРУКТУРАХ ДАНИХ ГРИ ГО 42

3.1. Загальний огляд системи 43

3.2. Трансформація у внутрішнє дерево 45

3.3. Модуль політики 50

3.4. Альфа-бета відсічення 52

3.5. Порівняння з шаблоном 55

3.6. Хешування Зобріста 59

3.7. Метод Монте-Карло пошуку в дереві 60

3.8. Метод верхньої оцінки значущості для дерева 63

4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА МЕТОДИКА ТЕСТУВАННЯ 65

4.1. Порівняння розміру початкового коду систем пошуку у структурах даних гри Го 65

4.2. Порівняння платформ систем пошуку інформації у структурах даних гри Го 66

ВИСНОВКИ 76

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ 78

ДОДАТКИ 81

# СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

SGF (Simple Game Format) – формат файлів для збереження партій настільних ігор.

GTP (Go Text Protocol) – протокол, який використовують більшість програм, що грають в Го для комунікації між собою.

MCTS (Monte-Carlo Tree Search) – метод Монте-Карло для пошуку в дереві. Евристичний алгоритм для пошуку та прийняття рішень. Його часто використовують програмах, які грають в настільні ігри

UCT (Upper Confidence bounds applied to Trees) – метод верхньої оцінки значущості для дерев. Один з методів Монте-Карло пошуку в дереві.

# ВСТУП

Го [1] – стародавня китайська стратегічна гра на двох гравців. Вона є антагоністичною [2] грою з повною інформацією. Те, що Го антагоністична гра означає, що двоє гравців мають протилежні умови виграшу, вони є суперниками. Ігри ж з повною інформацією, це ті ігри, у яких кожен з гравців знає все про стан гри у кожен момент часу, тобто такі ігри як шахи, шишки, В Го грають два гравці – “Чорні” та “Білі”. Вони по черзі розміщують на дошці що складається з перетину 19 на 19 ліній камені свого кольору. Після встановлення на дошку камені більше не рухаються. Однак є можливість зняти камінь з дошки. Кожен камінь або група каменів мають довкола себе так звані степені свободи, або даме (дихання). Кожна група або камінь, що більше не мають вільних степенів свободи коло себе – знімаються з дошки. Го територіальна гра, тобто гравець, що під кінець гри має більшу територію виграє. Контрольована територія визначається найчастіше візуально – дуже рідко коли гра ведеться до останнього ходу.

Як і для багатьох інших подібних ігор, були спроби створити комп'ютерні програми, що грають в Го, але це виявилося справжнім викликом для програмістів. Складність обчислення партій в Го на кілька порядків більша за шахи. На кожному кроці можливі близько 200-300 ходів. Також сама гра ведеться протягом близько 250 ходів у середньому.

Оцінка життя груп каменів – дуже важка вправа для людей, адже для розрахунку життя кожної групи треба розраховувати та моделювати гру на десятки альтернативних ходів вперед, кожен з яких може впливати на життя інших груп. Для комп’ютера така оцінка фактично неможлива. Єдиним поганим ходом можна цілком зіпсувати всю гру, навіть коли решта ходів були близькі до оптимальних. Тому програми для гри в Го не використовують таких алгоритмів, як шахові програми (тобто повний перебір ходів у глибину), а замість цього зазвичай мають кілька десятків модулів для оцінки різних аспектів гри і під час гри намагаються аналізувати гру ти використовувати ті ж самі поняття, що й люди. Попри це вони і далі грають дуже слабко та програють навіть не дуже сильним аматорам.

Усі програми, що грають в Го повинні оперувати деякими структурами даних (наприклад такими, що репрезентують поточний стан дошки або гри). Більшість з таких програм засновані на моделюванні великої кількості різних партій та подальшому їх аналізі. Саме тому питання пошуку у подібних структурах даних дуже актуальне, адже його оптимізація корінним чином вплине на роботу та якість програм, що грають в Го.

На даний момент найперспективнішим методом для моделювання партій з Го є метод пошуку Монте-Карло пошуку у деревах. Він і є основою для пошуку у структурах даних у даній дипломній роботі. В основі моделювання методом Монте-Карло лежить ідея, що якщо взяти велику кількість подій з предметної області і функцію оцінки результату цих подій, то в результаті можна отримати гарну модель оптимальної поведінки.

Говорячи про програми, що грають в Го, слід зазначити, що більшість з них розроблюються однією людиною і ще не було спроб створення модульної платформи для моделювання та аналізу ігри з Го. Найчастіше такі програми розробляють ентузіасти-одинаки, котрі не планують поширювати свій продукт і розробкою займаються на самоті. Модульна ж система для аналізу ігор, розрахунку наступного оптимального ходу дозволить більшій кількості розробників і ентузіастів займатися цією предметною областю, адже в них буде потужний інструмент без необхідності розроблювати увесь продукт самим.

Темою даної дипломної роботи є розроблення модульної, тобто гнучкої, системи для роботи з партіями з Го. Для цього потрібно не тільки мати можливість відкривати файли, у яких зберігаються ігри з Го, але і моделювати потенційні ходи у цій партії, знаходити найоптимальніший з них та виводити результуючий хід на монітор. Така система буде гнучкою, адже модулі, однакові за характером, можна буде міняти місцями між собою – можлива буде зміна алгоритмів пошуку між собою, зміна політик, що генерують набір потенційних ходів і заміна між собою різних модулів, що відповідають за ввід та вивід даних.

# АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ СИСТЕМ ПОШУКУ У СТРУКТУРАХ ДАНИХ ГРИ ГО

## 1.1. Огляд існуючих програмних продуктів для пошуку

Кожна програма, що грає в Го повинна вміти ефективно шукати ходи/партії у своїй внутрішній базі даних. Однак партії професійних гравців у Го мають важливе навчальне значення самі по собі. Тому існують програми, що єдиною своєю метою ставлять роботу з партіями Го. Вони об’єднують багато партій в одну базу даних і маніпулюють цією базою.

***1.1.1. Kombilo***

Kombilo [3] – програма, що працює з базою партій гри Го. Основне завдання такої програми – пошук деяких підпослідовностей в колекції SGF-партій (наприклад пошук усіх партій з деяким початком). Також ця програма дозволяє шукати по деяким властивостям партій (таким, як гравці, події, дати).

Особливості програми:

* Можливість пошуку як по повному ігровому полю, так і по позиціями у куті або на стороні дошки. Пошук ведеться також враховуючи симетрію та поворот. Існує можливість інвертувати пошук по кольору гравців, тобто поміняти їх місцями.
* Kombilo також комплектується повним редактором SGF: тобто існує можливість редагувати файли, коментувати їх та інше. Також редактор дозволяє бачити дерево варіантів партії. Він дозволяє повертати/віддзеркалювати партії.
* Kombilo також має у собі механізм по відображенню посилань. Цей механізм додає підказки до ігор, що впізнає. На поточний момент база цих посилань налічує 2000 записів.
* Можливо використовувати складні запити, використовуючи напряму SQL-базу, що зберігає розібрані партії.
* Можливо застосувати будь-яку комбінацію пошукових запитів, та у будь-який момент отримати список партій, що відповідають заданому пошуковому запиту.

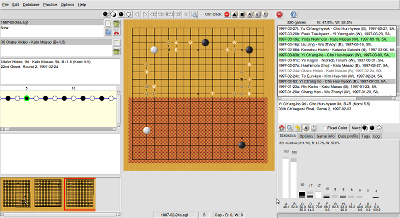


Рис. 1.1. Вигляд роботи з програмою Kombilo

Говорячи про реалізацію програми, можна відзначити що вона написана на Python, тому може вважатися багатоплатформною. Основне ж ядро, що присвячене пошуку, також написане на С++. Компілюючи його у модуль та підключаючи до основної програми ми отримаємо збільшення швидкодіє більше, ніж у два рази. Також це відкрита програма, і програмний код доступний для читання та аналізу.

***1.1.2. Master Go***

Це комерційна програма, що працює з базами даних гри го і призначена для пошуку поширених початків та розвитків ігор (вивченні фусекі і джосекі). База даних містить 53059 професійних ігор у власному форматі. Оновлення доступні для всіх зареєстрованих користувачів. Можливо додавати ігри в базу, що постачається з Master Go [4] шляхом придбання цих ігор в Японії, Китаї і Кореї і копіюванням їх в базу. Також Master Go працює тільки на операційній системі MS Windows.



Рис. 1.2. Вигляд роботи з програмою Master Go

Якщо порівнювати цю програму з Kombilo, та можна помітити декілька речей:

1. Master Go, комерційна та платна програма.
2. Вона використовує внутрішній формат даних, що також є не дуже зручним.

***1.1.3. BiGo Assistant***

BiGo Assistant [5] – програма для роботи з базами фусекі та джосекі, створена для гравців, що хочуть підняти свій рівень гри в Го.

Основні особливості:

* Можливість не тільки вивчати ігри го в цілому, але й переглядати ігри професіоналів у декількох режимах.
* Можливість вивчати початковий розвиток партії в го (фусекі) за прикладом бази даних професійних ігор.
* Також присутня база даних джосекі.
* Можливість аналізувати власні початки різних партій.
* Динамічний інтерфейс – можливість створення будь-якої кількості вікон з різними партіями.
* Можливість роздруковувати партії.
* Можливість застосувати до дошки всі 8 можливих перетворень (симетрія та поворот).
* Програма відслідковує дублікати ігор.
* Можливість збирати статистику по списку ігор у вигляді статистики по варіаціям можливих рухів або оціночну статистику позицій у грі обох гравців.
* Пошук по заданій позиції (по одній або декільком частинам дошки).



Рис. 1.3. Приклад роботи з програмою BiGo Assistant

Програма має багато можливостей, однак через те, що цікавим представляється тільки метод пошуку, ця програма має одну важливу відмінність від усіх інших: вона дозволяє шукати по декільком частинам дошки.

Говорячи про реалізацію програми, можна відзначити що вона написана на Java, тому може вважатися багатоплатформною. Основне ж ядро, що присвячене пошуку, також написане на С++. Його можна компілювати та підключати його у модуль та підключаючи до основної програми ми отримаємо збільшення швидкодіє більше, ніж у два рази.

1.1.4. AlphaGo

У 2016 році працівниками компанії Google була створена програма нового типу, що грала в Го. Вона використовувала деревовидну структуру даних, для збереження стану партії, метод Монте-Карло для оптимального пошуку у цьому дереві і дві різні нейронні мережі. Перша використовувалася для генерування потенційних ходів, тобто являла собою модуль політики. Друга нейронна мережа являла собою оціночну функцію і використовувалася для того, щоб не розраховувати гру до кінця і була можливість зупинитися на довільній глибині і розрахувати гравця, для якого поточна позиція є виграшною.

AlphaGo досконаліша за інші програми для гри в Го. У 500 іграх проти інших програм, включаючи Crazy Stone та Zen, AlphaGo перемогла 499 разів. У жовтні 2015 року, AlphaGo перемогла триразового чемпіона Європи Фан Хуі з рахунком 5:0. Це перший випадок, коли комп'ютерна програма перемогла професійного гравця в Го на класичній дошці в грі на рівних. Оприлюднення цієї новини було відкладено до 27 січня 2016 року, коли вийшов номер журналу Nature, в якому описуються застосовані алгоритми.

На березень 2016 року заплановано гру з Седолем Лі, південнокорейським професійним гравцем, що має 9 дан.



Рис. 1.4. Схема взаємодії нейронних мереж AlphaGo

AlphaGo єднає метод Монте-Карло пошуку у дереві та нейронні мережі. У дереві гри кожна вершина зберігає значення функції оцінки, кількість відвідувань та розраховану ймовірність. При роботі алгоритму виконується проходження по цьому дереву починаючи з його кореня. На кожному кроці головною задачею є розрахування нових значень політики на результату роботи оцінювальної функції. Ці обчислювання об’єднуються для створення нового рівня нейронних мереж та нової версії дерева гри.

## 1.2. Огляд існуючих алгоритмів пошуку у структурах даних гри Го

Структури даних і алгоритми – завжди є основою для моделювання будь-чого на комп’ютері. Решта частина дипломної роботи буде присвячена алгоритмізації, тому у першій її частині зконцентруємо увагу саме на структурах даних.

При моделюванні ігор постає задача збереження поточного стану ігри (при грі Го поточний стан задається позицією каменів на дошці), а також утримання в пам’яті потенційних рухів. Кожен з таких рухів створює новий стан дошки, що у свою чергу створює новий рекурсивний рівень потенційних рухів. Саме через таку рекурсивність для збереження партій у пам’яті найчастіше використовують деревовидну структуру даних. Також для такої структури даних існує багато алгоритмів, однак через великий розмір дошки для Го (від 9 на 9, до 19 на 19 і 31 на 31 у деяких випадках) розгалудженість та глибина подібного дерева були б дуже великими. Саме тому існує велика кількість різних алгоритмів, що спрощують роботу з великими деревами.



Рис. 1.5. Приклад вигляду закінченої партії в Го

Також обрана структура даних, будь це дерево або якась інша, повинна надавати можливості для швидкого повернення стану дошки і для збереження і роботу не тільки з окремими каміннями у контексті усієї дошки, але і з групами каменів, адже у грі Го декілька каменів, що знаходяться один до одного мають спільні степені свободи. Структури даних і алгоритми – завжди є основою для моделювання будь-чого на комп’ютері. Решта частина дипломної роботи буде присвячена алгоритмізації, тому у першій її частині зконцентруємо увагу саме на структурах даних.

Також важливим для роботи алгоритму є правильне програмування деяких досить незвичайних правил гри Го, таких як правило Ко. Правило Ко означає що для поточного гравця можуть існувати деякі заборонені позиції, а саме заборонено ходити так, щоб ситуація на дошці повторилася.

Єдиний вибір, що повинна зробити програма, що грає в Го, це куди поставити наступний камінь. Однак, цей вибір ускладняється тим, що навіть один камінь може дуже сильно впливати на ситуацію на дошці в цілому. Важливий фактор грає час – деякі ходу можуть бути актуальними лише на початку партії, в її середині (тюбан) або ж наприкінці (есє). У той же час не можна забувати про об'єднання каменів – їх групи, та про взаємодію цих груп між собою. Важливими чинниками для груп є їх розташування на дошці по відношенню до країв дошки і до інших груп і їх форма. Про форму груп для го існує багато статей та навіть книжок але у цій дипломній роботі буде лише розглянуто алгоритм для пошуку груп на дошці і ефективного їх збереження у відповідну структуру даних. Для вирішення цих проблем використовуються різні підходи. Розглянемо деякі з них.

Всі ігри з повною інформацією мають деяку оптимальну оціночну функцію, котра вирішує результат гри для будь-якого стану гри за ідеальних умовах для кожного з гравців. Подібні ігри можуть бути змодельовані рекурсивно вираховуючи оптимальне значення оціночної функціі у дереві пошуку. У таких іграх як Го, подібне дерево є дуже великим і на даний момент не існує машини з потрібною обчислювальною спроможністю, однак існує два принципи, котрі можуть допомогти спростити подібний пошук. По-перше, глибина пошуку може бути зменшена використовуючи оцінку позиції: не моделюючи гру до самого кінця і замінюючи оцінку фінальних позицій на оцінку поточної позиції, на якій було вирішено закінчити пошук. Подібний метод був використаний в шахах і надав можливості комп’ютеру перемогти професійного гравця. Однак подібний метод, як вважається, не може бути ефективно використаний для гри Го, зважаючи на складність останньої. По-друге розгалудженість дерева також може бути зменшена, якщо профільтрувати множину усіх можливих ходів через спеціальну функцію політики, що має вигляд функції ймовірності в залежності від конкретної клітинки. Наприклад, метод пошуку у глибину Монте-Карло може працювати майже без розширення дерева, якщо він буде використовувати подібну політику. Використовуючи подібний метод для повного пошуку у глибину, можна створити програму яка буде непогано грати в Го, на рівні аматорів-людей.

***1.2.1. Мінімаксний пошук по дереву варіантів***

В основі мінімаксного пошуку [6] лежить моделювання великої кількості різних партій. Загалом, алгоритм простий: спочатку алгоритм по черзі грає усі варіанти ходів до деякого моменту, будуючи дерево варіантів. Потім використовується оціночна функція (функція, що оцінює поточну позицію на дошці), для вираховування того, наскільки позиція гарна для кожного гравця. Далі, на основі цього зваженого дерева рухів, робиться розрахунок оптимальної стратегії для одного з гравців – вибираються ті ходи, що дають найбільше переваги цьому гравцю. Це робиться “згортанням” дерева з глибини і до кореня дерева. Мінімаксним цей метод називається тому, що на кожному рівні дерева, в залежності від того, чий наразі хід, першого гравца, або другого, шукається варіант руху з максимальним (тобто позитивним) результатом, або ж з мінімальним (тобто негативним результатом), адже ми враховуємо, що кожен з гравців буде грати максимально ефективно. Мінімаксний пошук гарно підходить для задачі Го, адже він дозволяю, базуючись на стані дошки, тобто враховуючи правила, скорочувати необхідні обчислення. Однак зрозуміло, що цей метод, як і всі інші розглянуті, не є абсолютно точним. У той час, як ми скорочуємо обчислення, ми втрачаємо деяку долю точності, в порівнянні з повним перебором.

Важливий фактор грає час – деякі ходу можуть бути актуальними лише на початку партії, в її середині (тюбан) або ж наприкінці (есє). У той же час не можна забувати про об'єднання каменів – їх групи, та про взаємодію цих груп між собою. Важливими чинниками для груп є їх розташування на дошці по відношенню до країв дошки і до інших груп і їх форма. Про форму груп для го існує багато статей та навіть книжок але у цій дипломній роботі буде лише розглянуто алгоритм для пошуку груп на дошці і ефективного їх збереження у відповідну структуру даних. Для вирішення цих проблем використовуються різні підходи. Розглянемо деякі з них.

Хоча цей метод є досить ефективним для шахів, він не дуже підходить для Го. По-перше, через те, що досі не було створено відповідної функції оцінки для позиції в Го. Гра Го все ще не формалізована математично, тому поточні функції оцінки партії запрограмовані робити висновки як люди. Найчастіше для цього використовуються або експертні системи (що є окремим джерелом для вивчення) або нейронні мережі. Однак навіть нейронну мережу треба якось навчати, а враховуючи те, що правильного алгоритму не існує, приклади для навчання повинні генерувати самі люди, що знов знижує точність. По-друге, через те, що для партії в Го притаманний великий фактор розгалуження – у кожний момент гри для кожного з гравців існує дуже багато коректних ходів. Також самі партії в Го довші, ніж у шахах. Саме тому такі методи дуже обчислювально коштовні. На даний момент програми, що використовують подібні алгоритми можуть грати тільки на дошках менших ніж 9x9.

***1.2.2. Порівняння з шаблоном***

Аналіз результатів ігор між різними комп’ютерними програмами, що використовують різні методи, дозволило дійти висновку, що кращий спосіб гри в Го – поєднання методів порівняння з шаблоном із методами швидкого локалізованого тактичного пошуку.

Методи, що використовують порівняння з шаблоном маніпулюють послідовністю ходів, що є прийнятною для обох гравців. Це відомі маленькі шматочки з яких найчастіше складаються локальні ситуації. На початку гри такі шаблони дуже гарно відомі і називаються фусекі і людина гравець витрачає велику частку часу саме їх вивченню. Такі послідовності добре вивчені і обґрунтовані, тому достатньо їх правильно використовувати всередині гри. Пошук цих зразків є дуже важливим як для гравців-людей, так я для програм, що грають в Го. Розглянемо один із можливих алгоритмів пошуку таких зразків в грі Го.

Виберемо деяку зону на дошці, яку будемо вважати зразком. Наприклад квадрат 5x5. Зрозуміло, що велике значення має саме вибір подібних частин дошки, але зараз ми не будемо зосереджувати на цьому увагу. Потім потрібно обчислити хеш цього квадрату, тобто представити у вигляді int64 числа. Однак перед цим потрібно привести його до деякого базового вигляду. Адже навіть однакові зразки з точністю до повороту або симетрії будуть виглядати різними на дошці, бо не будуть співпадати в усіх клітинках. Всього вісім різних позицій будуть однаковими. Чотири повороти (на 0, 90, 180, 270 градусів відповідно) і 4 повороти віддзеркаленого зразка. Нехай базовий вигляд буде вигляд, у якого хеш найменше число. Тоді порахувавши 8 хешів і обравши менший, ми зведемо всі подібні зразки до одного.

Також алгоритм хешування повинен обиратися таким, щоб він давав змогу знаходити неточні входження одного шаблону в інший, адже нас цікавить пошук незакінчених шаблонів. Далі, якщо зберігати багато подібних зразків у базу, то ця база може бути використана для пошуку у ній часток поточної гри. З великою вірогідністю, декілька початкових каменів дадуть змогу знайти відповідний гарний шаблон, який і треба буде далі відіграти програмі. Однак подібний метод все ж не “розуміє” важливих концепцій з гри Го, таких як вплив та жертвування. Для використання таких методів потрібно об’єднувати різні рішення.



Рис. 1.6. Пошук за однаковим шаблоном у двох різних кутах дошки

Після вибору такої зони дошці, яку будемо вважати зразком, зрозуміло, що велике значення має саме вибір подібних частин дошки, але зараз ми не будемо зосереджувати на цьому увагу. Потім потрібно обчислити хеш цього квадрату, тобто представити у вигляді цілого числа числа. Однак перед цим потрібно привести його до деякого базового вигляду. Адже навіть однакові зразки з точністю до повороту або симетрії будуть виглядати різними на дошці, бо не будуть співпадати в усіх клітинках. Всього вісім різних позицій будуть однаковими. Ми маємо наш шаблон і повертаючи його у чотори різні сторони і маючи на увазі явище симетрії при повороті вкупі ми отримаємо вісім унікальних перестановок, кожна з яких потребую своєї перевірки.

***1.2.3. Методи, засновані на ймовірності***

Базуючись на попередньому методі, можна отримати базу зразків партій досвідчених гравців. Використовуючи цю базу, можна отримати розподіл ймовірностей ходів для професійних ігор, який можна використовувати для відтворення цих ходів у окремій програмі. Цей розподіл можна використовувати не тільки для програми, що грає в Го, але й в якості навчального посібника для гравців у Го.

Цей метод має дві основні складові:

* Схема вилучення шаблону з експертних партій гри (реалізовано у попередньому пункті).
* Байесовський алгоритм навчання, який навчається розподілу аналізуючи ходи у локальному місці дошки.

Цей метод відрізняється від попереднього тим, що він не намагається отримати “повну” інформацію від гри і саме тому він не є настільки витратним по пам’яті.

***1.2.4. Методи, засновані на базі знань***

Якщо використовувати все ті ж самі шаблони, що згенерував метод порівняння зі зразком, та додати до них інтелект програміста, то вийде досить сильна програма для гри в Го. Під інтелектом програміста, мається на увазі можливість вирішувати локальну позицію у шаблоні використовуючи деякий набор евристик. Програмісту достатньо тільки перевести ці правила в комп’ютерний код та використати пошук за зразком, щоб знаходити ситуації, де ці правила доречні. Основний недолік – складність цих правил, а точніше можливість програмування їх, залежить насамперед від здатності та навику гри в Го самого програміста. Зважаючи на те, що математичного апарату для подібної роботи нема, кожен програміст намагається навчити свою програму грати, як він. Тому найчастіше програми мають більше сотні модулів, що вираховують найкращий хід кожен окремо для своєї ситуації. Однак такі методи страждають від проблем, аналогічних попереднім – нерозуміння глобальної ситуації. Це призводить до того, що вони роблять помилки у стратегічному плані. Відомо, що дуже легко програти гру, якщо у вирішальний момент обрати неправильний хід.

***1.2.5. Нейронні мережі***

Однією з головних альтернатив використанню жорстко запрограмованих методів пошуку – використовувати методи Монте-Карло. Існує ціле сімейство методів Монте-Карло, що об’єднує спільна ідея. Метод Монте-Карло пошуку у дереві збільшує свою точність, якщо збільшується кількість змодельованих рівнів, однак подібний метод подібний і потребує дуже великої кількості комп’ютерних ресурсів.

Програмісту достатньо тільки перевести ці правила в комп’ютерний код та використати пошук за зразком, щоб знаходити ситуації, де ці правила доречні. Основний недолік – складність цих правил, а точніше можливість програмування їх, залежить насамперед від здатності та навику гри в Го самого програміста. Зважаючи на те, що математичного апарату для подібної роботи нема, кожен програміст намагається навчити свою програму грати, як він. Тому найчастіше програми мають більше сотні модулів, що вираховують найкращий хід кожен окремо для своєї ситуації.

Говорячи про політику, що може використовуватися у методах Монте-Карло, то вона також змінюється протягом часу, адже вона часто реалізовується використовуючи нейронні мережі, котрі можуть самонавчатися протягом використання. Додаючи у режимі реального часу значення з поточної гри у нейронну мережу, можна отримати подібний результат. Асимптотичну подібна політика наближається до деякого ідеального та оптимального значення. Також оціночна функція, якщо реалізована подібним чином, має схожі властивості. Отже маючи необхідні потужності можливо розробити програму, що при навчанні буде наближуватися до оптимальної гри. На даний момент найсильніші програми для гри в Го використовуються саме таки підхід, що дозволяє їм тренуватися та вираховувати ходи, що обрали б професійні гравці.

Нещодавно, глибокі конволюційні нейронні мережі досягли дуже непередбачуваних результатів, загалом продуктивності, у задачах, що зв’язані з графікою, а саме: класифікація зображень, знаходження обличь та гра в ігри Atari. Такі нейронні мережі використовують багато рівнів нейронів, що перекривають один одного. Подібну архітектуру нейронних мереж використовують для гри Го. Дошку розглядають як картинку 19 на 19, що має 3 можливі кольори (білий камінь, чорний камінь, та пуста клітинка).

На початку нейронні мережі можна навчати на збережених іграх, тобто на рухах, що були зробленими гравцями-людьми. Це дає змогу дуже швидко навчити першу нейронну мережу, щоб у подальшому використовувати її, для отримання результату та для тренування наступних її версій.

Подальша робота над політикою та функцією оцінки відбувається при грі програми вже самої з собою. Цікавий варіант тренування нейронної мережі, що відповідає за політику використовуючи одну з випадкових попередніх версій цієї мережі. Це робиться для того, щоб звести нанівець можливість політики зациклитися на деяких окремих частинах дошки, ігноруючи усі інші. Хоча здається, що подібна велика кількість значень для розрахунку, сучасні комп’ютери здатні виконувати до 100,000 подібних циклів на кожен крок гри. Розробнику достатньо тільки перевести ці правила в комп’ютерний код та використати пошук за зразком, щоб знаходити ситуації, де ці правила доречні. Основний недолік – складність цих правил, а точніше можливість програмування їх, залежить насамперед від здатності та навику гри в Го самого програміста. Зважаючи на те, що математичного апарату для подібної роботи нема, кожен програміст намагається навчити свою програму грати, як він. Тому найчастіше програми мають більше сотні модулів, що вираховують найкращий хід кожен окремо для своєї ситуації.

1.2.6. Методи Монте-Карло

Якщо говорити про Го, то метод, що використовується, називається методом Монте-Карло пошуку у деревах і він полягає в наступному: згенеруємо список потенційних ходів, які ми хочемо перевірити; в ідеалі це будуть всі ходи, але це занадто витратно по ресурсах, тому використовують два варіанти:

* Рухи вибираються випадковим чином.
* Рухи генеруються окремим модулем, найчастіше у цій ролі виступає нейронна мережа.

Потім для кожного такого ходу зіграємо тисячу випадкових партій (тобто наступних випадкових ходів до остаточного кінця гри – стану на дошці, при якому можливо точно визначити переможця). Також, зважаючи на обчислювальні можливості, що доступні, можна грати партії не до кінця, а тільки до деякої глибини. Далі виберемо з дерева ходів той, при якому найбільша кількість з випадкових партій виграна програмою. Цей метод дуже схожий на метод Мегамакс, але він відрізняється від нього тим, що ставка ставиться саме на велику кількість зроблених ходів.

Перевага цього методу полягає в тому, що він потребує дуже малих знань про предметну область, у якій працює. Єдине, що потрібно “пояснювати” цьому алгоритму – це що називається ходом, тобто на кожному конкретному кроці які ходи є дозволеними. Також необхідно надати йому оцінюючу функцію. Змінюючи ці два параметри можна досягнути кардинально різних результатів, не змінюючи сам алгоритм.

Недоліком подібних алгоритмів є те, що вони використовують більше пам'яті та процесорного часу, і що через те, що ходи генеруються випадковим чином, ми можемо неправильно оцінити якість ходу. Наприклад, якщо у відповідь на якийсь хід буде згенеровано 100 ходів-відповідей, у більшості з який перший гравець виграє, ми будемо оцінювати цей хід як дуже сильний.



Рис. 1.8. Приклад підрахунку результату 12 змодельованих ігор

Однак існує можливість того, що є дуже добра відповідь на наш хід, яка зведе нашу перевагу нанівець, однак, через те, що ми не згенерували цей хід, ми про це не дізнаємося. В результаті цього, програма буде сильна в загальному стратегічному сенсі, однак буде дуже слаба тактично. Цю проблему можна вирішити, якщо додати деякі проблемно-орієнтовні знання в генерацію ходів або підвищити глибину пошуку. При першому варіанті алгоритм буде вибирати випадковим чином не зі всіх дозволених варіантів на кожному кроці, а лише з деякої їх множини, що була вибрана на попередньому кроці модулем політики. Другий варіант підходить коли є можливість використовувати багато програмних ресурсів.



Рис. 1.8. Приклад роботи модуля політики

На рисунку 1.8 показані наступні етапи роботи модуля політики:

1. Поле значень оціночної функції.
2. Дерево ходів, отриманих з поля значень.
3. Дерево ходів, отриманих з методом Монте-Карло.
4. Поле значень політики.
5. Відсоток виграшних симуляцій.
6. Головна варіація.

У 2006 році був створений новий пошуковий алгоритм, що був використаний для гри на дошках розміру 9x9 та зарекомендував себе дуже гарно. Він називається Upper Confidence Bounds algorithm і базується на методі Монте-Карло. Алгоритм UCT [7] змінює правила за якими визначається важливість ходів у дереві пошуку. Він вибирає те піддерево, у якому вірогідність перемоги більше 50%. Якщо ж не існує такого піддерева, то вибір робиться навмання. Зважаючи на це зрозуміло, що кардинально нових алгоритмів за останній час не було знайдено і прорив може відбутися лише у ресурсному плані, або комбінуючи вже описані алгоритми оптимальним чином.

# 2. ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

Темою даної дипломної роботи є розробка системи для паралельного пошуку у структурах даних гри Го використовуючи метод Монте-Карло пошуку у деревах. Реалізована бібліотека повинна дозволяти шукати у структурах даних гри Го методом Монте-Карло пошуку у деревах. Ця бібліотека може бути використана програмами, що грають в Го, аналізують партії в Го, збирають статистику з партій або ж шукають деякі закономірності у партіях. Щоб ця бібліотека була корисною і використовувалася, важливо вибрати правильну платформу та мову програмування для неї. Також дуже важливо організувати проект так, щоб не виникало труднощів при спробах модифікувати його. Корисно розбити проект на модулі, кожен з яких буде відповідати за свою частину роботи системи. Подібний структурний розподіл обов’язків дозволяє одночасну модифікацію різних компонент системи, що спрощує розробку групою програмістів.

Основними властивостями розроблювальної бібліотеки повинні бути:

1. Платформо незалежність – це зробить бібліотеку більш поширеною та зручною.
2. Можливість використання з багатьма популярними мовами програмування.
3. Простота реалізації *–* внутрішні структури даних змодельовані використовуючи вбудовані у мову структури даних.
4. Структурна незалежність компонент ­*–* спрощує паралельну розробку.

## 2.2. Вибір платформи

Вибір платформи для бібліотеки – дуже важливе рішення, адже платформа одразу не тільки поставить обмеження на доступні мови програмування але і додає свої плюси та мінуси, до розроблюваного програмного забезпечення.

Три основні варіанти реалізації програмних продуктів, що розглядалися:

* Без використання платформи.
* На основі платформи Java.
* На основі платформи Microsoft .NET.



Рис. 2.1. Порівняння структури платформ .NET та Java

Розглянемо кожен з цих варіантів окремо.

***2.1.1. Без використання платформи***

Найочевиднішим вибором буде не використовувати ніяку платформу. Хоча програмування, використовуючи платформу, таку як Java Platform або Microsoft .NET, має свої значні переваги, при програмування бібліотеки методів пошуку не обов'язково використовувати якусь з них, тож розглянемо і цей варіант.

Основними перевагами написання програми без використання будь-якої платформи є:

* відсутність прошарків між програмою та ОС (найчастіше);
* відсутність прив’язаності до будь-яких інструментів, бібліотек, утиліт;
* можливість написати простий та компактний standalone-додаток.

Основними мовами, що розглядалися були:

* C та C++;
* Python;
* Common Lisp.

Мови С та С++ є не тільки мовами системного програмування. Їх можливо з успіхом використовувати для розробки різноманітних додатків. Хоча вони також мають деякі фреймворки та бібліотеки, що можна використовувати для спрощення розробки та створення кросс-платформенних програм. Основною перевагою цих мов є швидкодія отриманої програми, що є досить важливим фактором для програм, що грають в Го. Недоліком є складність та витратність розробки програмного забезпечення. Також не дуже зручно програмувати багатозадачні системи якщо використовувати такі мови. Багатозадачність ж є однією з ключових особливостей розроблювальної системи. Загалом хоча ці мови і підходять для даного типу задач, я вважаю, що їх не доцільно використовувати у даній дипломній роботі, адже вони додають чимало складності системі.

Python – широко використовувана, скриптова, інтерпретована, високорівнева мова програмування. Вона доступна для більшості платформ. Вона підтримує концепти функціонального програмування, зокрема у ній функції є об'єктами першого класу (тобто можуть передаватися як змінна та бути збереженими у змінну). Також Python має багато вбудованих типів, що гарно підходять для даної задачі. На відміну від попередньо-розглянутих мов, Python має REPL (Read-Eval-Print-Loop) – цикл читання-обробки-відображення, що дуже спрощує розробку, робить її більш динамічною і надає можливість тестувати роботу окремих компонент системи, не чіпаючи усі інші.

Серед мінусів слід зазначити, що ця мова інтерпретована. Існування інтерпретатора накладає свої мінуси, серед яких зменшення швидкодії та залежність від додаткових програмних засобів. Хоча, незважаючи на це, існує багато прикладів розроблення і розповсюдження додатків написаних використовуючи Python і що мали необхідну портативність.

Недоліком є складність та витратність розробки програмного забезпечення. Також не дуже зручно програмувати багатозадачні системи якщо використовувати такі мови. Багатозадачність ж є однією з ключових особливостей розроблювальної системи. Загалом хоча ці мови і підходять для даного типу задач, я вважаю, що їх не доцільно використовувати у даній дипломній роботі, адже вони додають чимало складності системі.

Common Lisp – це діалект Lisp, що набув значного розповсюдження у програмному світі. Для нього існує реалізації під більшість платформ. Мова високорівнева, підтримує багато парадигм програмування, компільована. Основні переваги для роботи з деревами ця мова має через те, що вона є Lisp-мовою, тобто вона створена для маніпулювання такими структурами даних, як списки. У програмуванні дерево найчастіше подається у вигляді списку списків, тому більшість алгоритмів для роботи з деревами гарно програмуються на Lisp. Використовуючи таке стандартне подання дерев ми також отримаємо такі плюси, як можливість використовувати велику кількість як вбудованих функцій, так і функцій, написаних іншими людьми. Common Lisp також має менеджер пакетів Quicklisp, що також дуже спрощує як розробку, так і подальше розповсюдження програми. Серед мінусів слід зазначити невелику популярність (якщо порівнювати з іншими не Lisp), та невелику кількість бібліотек.

Недоліком є складність та витратність розробки програмного забезпечення. Також не дуже зручно програмувати багатозадачні системи якщо використовувати такі мови. Багатозадачність ж є однією з ключових особливостей розроблювальної системи. Загалом хоча ці мови і підходять для даного типу задач, я вважаю, що їх не доцільно використовувати у даній дипломній роботі, адже вони додають чимало складності системі.

Загалом, серед розглянутого, Common Lisp був би найкращим вибором, якщо якось подолати його мінуси.

***2.1.2. На основі платформи Java***

Віртуальна машина Java – набір комп'ютерних програм та структур даних, що використовують модель віртуальної машини для виконання інших комп'ютерних програм чи скриптів. JVM використовує байт-код Java, який генерується з вихідних кодів мови програмування Java. Віртуальну машину також застосовують для виконання коду, згенерованого з інших мов програмування. JVM доступна для всіх основних сучасних операційних систем, тому про програми, що скомпільовані у Java байткод теоретично можна сказати “Написано один раз, працює скрізь”.

Основними мовами для розглядання були:

* Groovy;
* Scala;
* Clojure.

Groovy – об'єктно-орієнтована динамічна мова програмування, що працює в середовищі JRE. Вона розробляється під ліцензією Apache Commons. Мова Groovy запозичила деякі корисні якості Ruby, Haskell і Python, але створена для роботи всередині віртуальної машини Java (JVM) і підтримує тісну інтеграцію з Java програмами. Groovy має досить багато функціональних можливостей, що значно спрощує швидку розробку додатків. З недоліків слід зазначити те, що ця мова скриптова, тобто неможливість створення компактної портативної програми, залежність від JVM.

Ключові особливості мови:

* Безшовна інтеграція з Java.
* Висока швидкість розробки – Groovy є більш високорівневою мовою програмування порівняно з Java, а отже розробка на ньому зазвичай відбувається швидше. Цьому сприяють перш за все динамічна природа мови, а по-друге, наявні елементи функціональнального програмування, зокрема замикання.
* Функціональна спрямованість – саме цьому аспекту мови розробники надають один з найбільших пріоритетів. Нові можливості з'являються досить регулярно.
* Режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

Scala – мова програмування, що підтримує багато парадигм програмування та поєднує властивості об'єктно-орієнтованого та функціонального програмування. З точки зору функціонального програмування – Scala має дуже потужну систему типів, вона має навіть типи вищого порядку. Подібний функціональний підхід дозволяє дуже швидко прототипувати, а потім і розроблювати великі програмні комплекси. Прив’язка до Java ж забезпечує необхідну кількість бібліотек.

За умовчанням рівень доступу до членів класу у Scala є загальнодоступним (public). Відкриття доступу підкласам (protected), так само як у C++ і C# відкриває доступ лише підкласам, на відміну від Java, в якій protected також отримують доступ усі класи того самого пакету. Члени з видимістю protected доступні у інших екземплярах цього ж класу, але нащадки не мають доступу до protected членів інших екземплярів своїх базових класів (так само як і в Java).

Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [8], що не дозволяє Scala.

У Scala protected і private можуть мати кваліфікатор доступу у квадратних дужках: protected. Кваліфікатор доступу члена може бути іменем зовнішнього класу чи пакету, тоді вказаний клас або всі класи, що знаходяться у вказаному пакеті отримують доступ до члена. Також кваліфікатор доступу може бути this, тоді член доступний лише із свого екземпляру, але не доступний через інші екземпляри класу. Супутні об'єкти (companion objects) мають спільний рівень доступу із своїми супутніми класами.

Clojure – сучасний діалект мови програмування Lisp. Це мова загального призначення, що підтримує інтерактивну розробку, зорієнтовану на функціональне програмування, спрощує програмування декількох потоків у одній програмі та містить риси сучасних скриптових мов. Clojure працює на Java Virtual Machine і Common Language Runtime.

Підхід Clojure до паралельності характеризується концепцією тотожностей, що представляють серію незмінних станів протягом часу. Оскільки стани є незмінними значеннями, будь-яка кількість обробників може паралельно обробляти їх і конкуренція зводиться до питання керування змінами від одного стану до іншого. З цією метою, Clojure надає декілька типів змінюваних посилань, кожен з яких має добре визначену семантику переходу між станами. Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [9], що не дозволяє Scala.

Як і інші Lisp-подібні мови, синтаксис Clojure побудовано на *S*-виразах, які в процесі синтаксичного розбору спершу перетворюються на структури даних за допомогою функції-читача, перш ніж компілюються. Clojure's reader підтримує літеральний синтаксис для хеш-таблиць, множин та векторів на додаток до списків, і вони передаються компілятору як є. Іншими словами, компілятор Clojure компілює не лише спискові структури даних, але й безпосередньо підтримує всі названі вище типи. Clojure — Lisp-1, і не є сумісним за кодом з іншими діалектами мови Lisp.

Система макросів Clojure дуже схожа на використовувану в Common Lisp, з тією відмінністю, що версія синтаксичного цитування Clojure (з використанням знаку `) доповнює символи їхніми просторами імен. Це допомагає запобігти ненавмисному перехопленню імен, оскільки прив'язка до імен, доповнених простором імен, заборонена. Є можливість форсувати таке захоплення імен, але це має бути зроблено явно. Clojure також не дозволяє переприв'язку глобальних імен з інших просторів імен, які були імпортовані в поточний простір.

***2.1.3. На основі платформи Microsoft .NET***

Microsoft .NET – платформа від фірми Microsoft для створення як звичайних програм, так і веб-застосунків. Багато в чому є продовженням ідей та принципів покладених в технологію Java Platform. Одною з ідей Microsoft .NET є сумісність служб написаних різними мовами. Хоча ця можливість рекламується як перевага Microsoft .NET, платформа Java має таку саму можливість. Функціональній спрямованості мови розробники надають один з найбільших пріоритетів. Нові можливості з'являються досить регулярно. Існує режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

Окрім повної версії .Net, компанією Microsoft також випускається так званий «.Net Compact Framework». .Net Compact Framework є обрізаною версією повного фреймворка і несумісний з ним на рівні виконання (програми, написані для Compact Framework не можуть виконуватись виконавчим середовищем від повної версії фреймворка, для їх виконання необхідно встановити виконавче середовище саме від Compact Framework). Внутрішньо Compact Framework працює дещо інакше, ніж повний фреймворк, наприклад “збирач сміття” працює значно більш агресивно, не розділяючи об'єкти на покоління. Відмінності здебільшого обумовлені особливостями роботи компактних пристроїв: меншими розрахунковими можливостями, значно вищими вимогами до низьких енергозатрат, обмеженими графічними можливостями.

Основні мови, що розглядалися:

* C#;
* C++/CLI;
* F#.

C# – об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи Microsoft .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research. Станом на сьогодні C# визначено флагманською мовою корпорації Microsoft, бо вона найповніше використовує нові можливості .NET. Решта мов програмування, хоч і підтримуються, але визнані такими, що мають спадкові прогалини щодо використання .NET.

C# розроблялась як мова програмування прикладного рівня для CLR і тому вона залежить, перш за все, від можливостей самої CLR. Це стосується, перш за все, системи типів C#. Присутність або відсутність тих або інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльована у відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C#; подібної взаємодії слід чекати і надалі. (Проте ця закономірність буде порушена з виходом C# 3.0, що є розширеннями мови, що не спираються на розширення платформи .NET.) CLR надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, збірка сміття не реалізована в самому C#, а проводиться CLR для програм, написаних на C# точно так, як і це робиться для програм на VB.NET, J# тощо. Функціональній спрямованості мови розробники надають один з найбільших пріоритетів. Нові можливості з'являються досить регулярно. Існує режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

C++/CLI – прив'язка мови програмування С++ до платформи .NET фірми Microsoft. Ця мова підтримує стандарт C++ ISO. Також до його підтримки додається Об'єднана система типів (Unified Type System, UTS), що розглядається як частина Загальної мовної інфраструктури (Common Language Infrastructure, CLI). Вона підтримує і рівень програмного коду, і функціональну сумісність виконуваних файлів, скомпільованих із керованого C++. C++/CLI являє собою еволюцію С++.

Відстеження посилань в C ++ / CLI відбувається автоматично, а не вручну. Концект схожий на використання “\* &” (посилання на покажчик) в стандарті C++. При оголошенні функції йому відповідає ключове слово “REF, що застосовується до типів в C #, або “ByRef” в Visual Basic .NET. C++/CLI використовує синтаксис “^%”для вказівки посилання відстеженн, що не відбувається автоматично.

F# – багатопарадигмова мова програмування розроблена в підрозділі Microsoft Research і призначена для виконання на платформі .NET. Вона поєднує в собі виразність функціональних мов, таких як OCaml і Haskell, з можливостями і об'єктною моделлю Microsoft .NET. Функціональна мова максимально адаптована до використання на платформі Microsoft .NET, відповідно, вона не заперечує і імперативного підходу. Функційна мова максимально адаптована до використання в .NET Framework, відповідно, вона не заперечує і імперативного підходу.

Функціональній спрямованості мови розробники надають один з найбільших пріоритетів. Нові можливості з'являються досить регулярно. Існує режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

Протягом тривалого часу F# існував як дослідницький проект, основним завданням якого було збагатити імперативну мову C# можливостями, традиційно доступними лише функціональним мовам. Безліччю нововведень C# 3.0 з VS 2008 завдячує саме йому. Сам по собі F # не створений з чистого аркуша в Microsoft, в його основу покладено досить популярний OCaml, який, у свою чергу, походить від одної з перших типізованих функціональних мов ML. Попри те, що синтаксично F# і OCaml досить близькі, вони не еквівалентні: грубо кажучи, перший становить собою підмножину другого, доповнену доступом до властивостей .NET Framework. Однак деякі програми на OCaml можуть бути практично без модифікацій скомпільовані F#, зворотна компіляція також справедлива, зрозуміло, за відсутності звернень до класів .NET Framework.

***2.1.4. Висновки***

Написання бібліотеки без платформи хоч і має свої плюси, однак мінуси у вигляді втрати доступу до великої кількості розробників, що все використовують платформу, важать більше, ніж плюси. До того ж більшість розглянутих мов без платформи не дуже підходить до задачі. Виняток складає мова Common Lisp, що гарно підходить для програмування вибраних методів, однак її популярність ще менша.

Платформа Microsoft .NET сама по собі досить актуальна, але набір мов, що доступні для розробки на цій платформі, також не дуже підходить для розробки системи пошуку у структурах даних гри Го.

У свою чергу платформа Java демонструє багато позитивних сторін, що знадобляться для реалізації методів пошуку. Вона популярна, багатоплатформна, має значний набір мов програмування, найкраще з яких – Clojure – гарний вибір для розробки бібліотеки.

Розглянемо мови програмування платформи Java більш детально.

## 2.2. Вибір мови програмування

***2.2.1. Groovy***

Groovy [10] є більш високорівневою мовою програмування порівняно з Java, а отже розробка на ньому зазвичай відбувається швидше. Цьому сприяють перш за все динамічна природа мови, а по друге існуючі елементи функціонального програмування, зокрема замикання.

Функціональній спрямованості мови розробники надають один з найбільших пріоритетів. Нові можливості з'являються досить регулярно. Існує режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

Серед переваг Groovy слід зазначити те, що вона:

1. Скриптова мова, за компактністю схожа на Python.
2. Значно швидша: звичайні регулярні вирази і рядкові функції виконуються разів в 5 швидше.
3. Розуміє мову і класи Java.
4. Як наслідок має велику кількість готових високоякісних бібліотек.
5. Є мовою Gradle, що спрощує будування багатоплатформових проектів.

***2.2.2. Scala***

На Scala [11] вплинуло багато мов. Однорідна об'єктна модель вперше з'явилася у Smalltalk і згодом у Ruby. Універсальність вкладеності присутня у Algol, Simula, Beta. Принцип однорідного доступу для виклику методу і звернення до поля походить з мови Eiffel. Підхід до функціонального програмування подібний до підходу родини мов ML, таких як OCaml і F#. Багато функцій вищого порядку у стандартній бібліотеці Scala також наявні у ML або Haskell. Неявні параметри у Scala аналогічні класам типів Haskell. Заснована на акторах бібліотека багатозадачності подібна до Erlang. Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [12], що не дозволяє Scala.

Згідно опитування ZeroTurnAround 2014 року 47% Java-розробників обрали би Scala наступною мовою програмування для JVM [13]. Автор мови Groovy Джеймс Стракен розглядає Scala як заміну Java у довгостроковій перспективі, та стверджує, що якби він свого часу познайомився зі Скалою, то імовірно не створив би Groovy [14]. Основний розробник JRuby Чарльз Наттер вважає Scala “наступником трону Java” [15]. Брюс Еккель, автор відомих книжок “Філософія C++” і “Філософія Java”, стверджує, що Scala, зберігаючи переваги і статичної типізації, часто відчувається навіть більш лаконічною та ясною ніж мови із динамічною типізацією, такі як Python. Кей Хорстман, відомий як автор книги “Основи Java”, вважає Scala найпривабливішим вибором для тих, хто хоче просунутися за межі Java чи С++. Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [16], що не дозволяє Scala.

Згідно опитування O'Reilly у 2015 році 10% дослідників даних використовують Scala. Значною мірою це пов’язано із популярністю Apache Spark, яким користуються 17% дослідників, із яких 46% використовують одночасно Spark і Scala. Знання цих двох технологій виявилося найзначущим коефіцієнтом пов’язаним із розміром платні, яка в середньому вища на $15000 ніж у аналогічних дослідників, що не володіють ними [17].

Це не окремий тип колекцій, особливо враховуючи, що їм не обов'язково бути саме списками. Доступні також такі вбудовані структури даних, як вектор, хеш-словник, набір та інші. При спробі отримати з порожньої колекції послідовність її елементів (викликом seq) повертається nil. При спробі отримати з послідовності (на її останньому елементі) залишок (rest) буде повернуто іншу логічну послідовність. Це дозволяє послідовностям та протоколу послідовностей бути лінивими.

***2.2.3. Clojure***

Clojure [18] компільована мова, вона геренує байткод для JVM. Вона має значну інтеграцію з Java: відкомпільовані в байткод JVM, програми на Clojure можуть пакуватися та запускатися на JVM-серверах без додаткових ускладнень. Мова також надає макроси, які полегшують використання існуючих Java API. Всі структури даних Clojure реалізують стандартні інтерфейси Java, що робить простим запуск з Java коду, розробленого на Clojure. Подібна політика використання Java байт-коду де можливо дуже спрощує не тільки розробку на Clojure, але і робить її більш швидкою. Зважаючи на те, що більшість даних будуть зберігатися у стандартних колекціях і на те, що при програмуванні на Clojure найчастіше намагаються використовувати не тільки примітивні колекції, але й примітивні типи. Це зменшує навантаження на віртуальну машину Java, адже примітивні типи у колекціях не потребують процедури обгортання у об’єкти (boxing & unboxing). Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [19], що не дозволяє Scala.

Це не окремий тип колекцій, особливо враховуючи, що їм не обов'язково бути саме списками. Доступні також такі вбудовані структури даних, як вектор, хеш-словник, набір та інші. При спробі отримати з порожньої колекції послідовність її елементів (викликом seq) повертається nil. При спробі отримати з послідовності (на її останньому елементі) залишок (rest) буде повернуто іншу логічну послідовність. Це дозволяє послідовностям та протоколу послідовностей бути лінивими.

Також Clojure має мультиметоди (аналог перевантажень функцій), що підтримують динамічний вибір метода за типами та значеннями довільного набору аргументів. Подібна реалізація відрізняється від простої і стандартної реалізації, що є присутньої у мові Java – у ній перевантаження функцій може відбуватися лише за сигнатурою методу.



Рис. 2.2. Модель виконання коду на платформі Clojure

Найбільша відмінність Clojure – послідовності. Це не окремий тип колекцій, особливо враховуючи, що їм не обов'язково бути саме списками. Доступні також такі вбудовані структури даних, як вектор, хеш-словник, набір та інші. При спробі отримати з порожньої колекції послідовність її елементів (викликом seq) повертається nil. При спробі отримати з послідовності (на її останньому елементі) залишок (rest) буде повернуто іншу логічну послідовність. Це дозволяє послідовностям та протоколу послідовностей бути лінивими.

***2.2.4. Висновки***

Серед розглянутих мов найбільше до вирішення даної задачі підходить мова Clojure. Вона має прив'язку до Java, тому це гарний варіант для розроблювання бібліотеки. Також ця мова – діалект Lisp, тобто вміє добре працювати з списками та деревами. Не слід забувати про гарну підтримку паралельного програмування, яка знадобиться при розроблюванні даної програмної системи.

# 3. РОЗРОБКА СИСТЕМИ ПАРАЛЕЛЬНОГО ПОШУКУ У СТРУКТУРАХ ДАНИХ ГРИ ГО

Як було показано в першому розділі існує немало програм для пошуку у структурах даних гри Го, але практично всі вони є комерційними. Некомерційною є програма Kombilo, але вона має такі недоліки:

* Складність роботи з системою. Якщо хтось захоче додати підтримку Го у свою систему, йому доведеться розбиратися з великою кількістю коду на С++.
* Графічний інтерфейс на Python. Хоча Python багатоплатформна мова, але для графічного інтерфейсу краще використовувати щось більш стандарте, що не залежить від додаткових програмних модулів.

Інша некомерційна програма, для гри в Го – Fuego [20]. Вона також написана на C++, але якщо аналізувати структуру її модулів, то вона краще організована та простіша у використанні. Однак у цій системі бракує методів для пошуку у структурах даних гри Го.

Розроблювальна система має схожу структуру модулів, вона зображена на кресленні “Схема взаємодії̈ програмних модулів”. Подібна структура бібліотеки дає змогу модулям бути самостійними одиницями і використовуватися окремо.

Після огляду існуючих розробок можна зробити висновок про актуальність даної програмної розробки. Саме тому є доцільним розробити систему пошуку для гри Го, яка буде написана, використовуючи можливості платформи Java та мови Clojure. Така система повинна бути модульною та зручною для розробників у використанні. Це дасть змогу великій кількості людей використовувати цю систему, додавати до неї нові модулі та робити дослідження у області штучного інтелекту для гри в Го

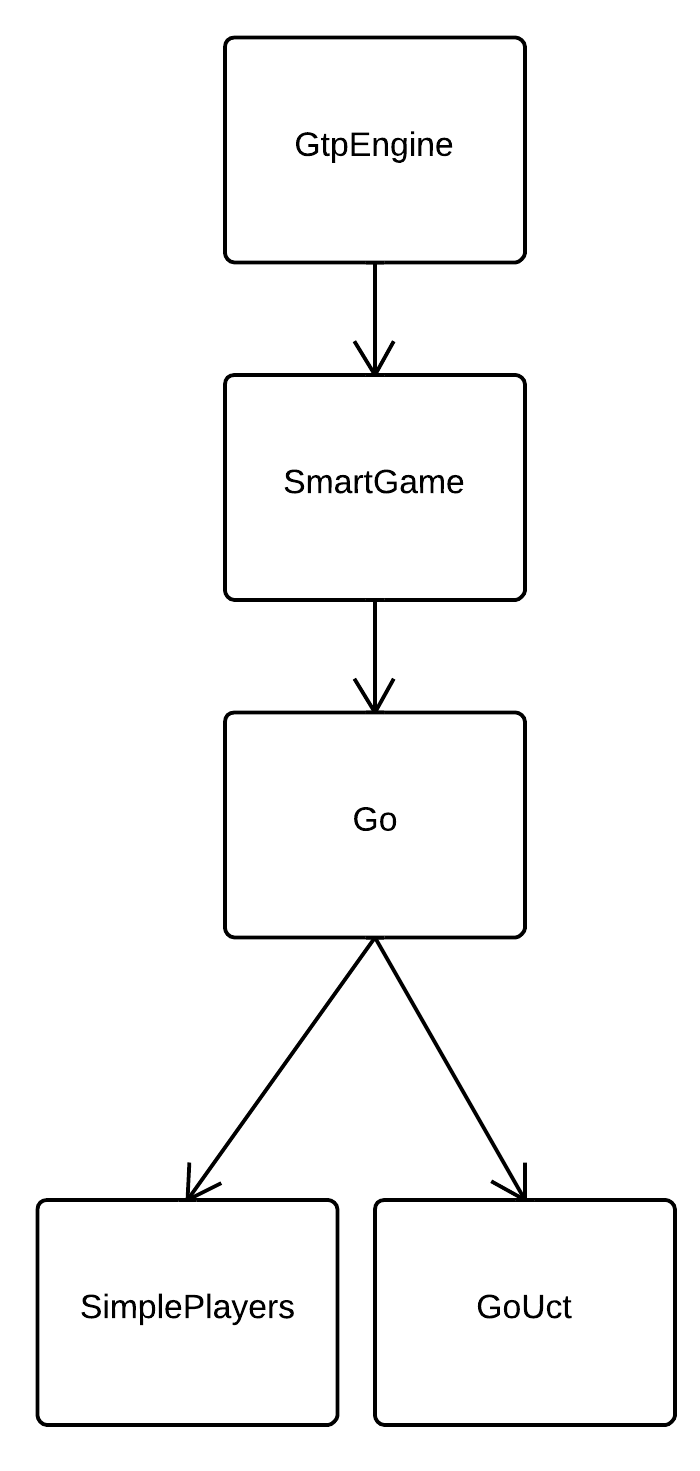


Рис. 3.1. Структура модулів Fuego

## 3.1. Загальний огляд системи

Розроблювана система пошуку у структурах даних гри Го складається з чотирьох модулів:

* модуль трансформації SGF-файлів у внутрішнє дерево;
* модуль методу Монте-Карло пошуку в дереві;
* модуль політики.

Перший модуль – модуль трансформації у внутрішнє дерево. Він отримує на вході SGF-файл та оброблює його використовуючи парсер, що перетворює цей файл на послідовність лексем та лексер, що перетворює послідовність лексем на внутрішнє дерево. У процесі розробки даного модуля була створена система функцій, що дозволяє ліниво читати файл блоками по 512 байт. Даний підхід гарно корелює з функціональною моделлю структур даних у мові Clojure, адже дозволяє зменшуючи обчислювальні витрати та витрати пам’яті, маніпулювати тим самим списком символів з файлу. Члени з видимістю private доступні лише у екземплярах цього ж класу. У Java private обмежує доступ на рівні класу найвищого рівня вкладеності, тобто зовнішній клас може одержати доступ до private члена внутрішнього вкладеного класу [22], що не дозволяє Scala.

Функція (lazy-read file len) отримує шлях до файлу, що потрібно читати та розмір блоків для читання. Було знайдено, що значення 512 байт для параметру len дає гарні результати швидкості читання файлу.

Лістинг 3.1. Функція lazy-read

(defn lazy-read [file len]

(let [rdr (clojure.java.io/reader file)

is-closed? (atom false)

array (char-array len)]

(fn next-char []

(if @is-closed?

nil

(let [result (.read rdr array)]

(if (= -1 result)

(do (.close rdr)

(swap! is-closed? complement)

nil)

array))))))

Ця функція повертає замикання на функцію, що читає файл масивами по len байт. Але лексеру необхідно читати файл по одному символу інколи повертаючи прочитаний символ назад у файл. Використовуючи java.io.reader такої поведінки досягнути неможливо. Тому були створені допоміжні функції (read-next-char lazy-read-fn stack) та (unread-char chr stack). Вони використовують функцію, отриману від lazy-read та стек, в який зберігають чергу символів для читання.

Також цей метод потребує модуль правил гри Го, що складається з модуля генерування допустимих ходів та модуля оцінки поточної ситуації на дошці. Модуль генерування допустимих ходів повинен повертати список ходів, що не є забороненими в даній ситуації. Він повинен враховувати правила зняття з дошки каменів та правило Ко. Модуль оцінки поточної ситуації на дошки найчастіше реалізовується у вигляді евристичної функції оцінки ситуації. У даній дипломній роботі розглядається розробка модуля політики, який базується на простій багаторівневій нейронній мережі.

Третій модуль реалізує один із методів Монте-Карло пошуку в дереві, а саме – метод верхньої оцінки значущості для дерев. Цей модуль також потребує модуль правил гри Го. Цей модуль генерує велику кількість випадкових ігор на основі початкової позиції та використовує отримане дерево партій для пошуку найкращого варіанту ходу. Модуль генерування допустимих ходів повинен повертати список ходів, що не є забороненими в даній ситуації. Він повинен враховувати правила зняття з дошки каменів та правило Ко. Модуль оцінки поточної ситуації на дошки найчастіше реалізовується у вигляді евристичної функції оцінки ситуації. У даній дипломній роботі розглядається розробка модуля політики, який базується на простій багаторівневій нейронній мережі.

Розглянемо ці модулі більш детально.

## 3.2. Трансформація у внутрішнє дерево

Smart Game Format (SGF) – комп’ютерний формат даних, що зберігає партії ігор, таких як Го, шахи, шашки та реверсі. Це простий текстовий формат, що зберігає партії у вигляді дерев.

Були розроблені функції, що дозволяють маніпулювати файлами у цьому форматі: читати його, перетворювати у внутрішній деревовидний формат, спрощувати дерево, перетворювати у формат, що відображає положення дошки та візуалізовувати отриману дошку.

Модуль генерування допустимих ходів повинен повертати список ходів, що не є забороненими в даній ситуації. Він повинен враховувати правила зняття з дошки каменів та правило Ко. Модуль оцінки поточної ситуації на дошки найчастіше реалізовується у вигляді евристичної функції оцінки ситуації. У даній дипломній роботі розглядається розробка модуля політики, який базується на простій багаторівневій нейронній мережі.

Читання та перетворення SGF-файлу запрограмовано у вигляді лексеру та парсеру. Лексер приймає на вході послідовність символів з файлу, що читаються по мірі необхідності, а генерує послідовність лексем, тобто ключових слів, що притаманні даному формату. Лексер працює як словник, що перетворює вхідну послідовність символів на лексеми та значення властивостей. До значень властивостей відносяться чісла та рядки. До списку реалізованих лексем належать такі лексеми:

* :treestart;
* :treeend;
* :nodestart;
* :propident;
* :propvalue;
* :propvalueend.

Далі використовується парсер – функція, що парсить список лексем, генеруючи дерево властивостей. Саме ця функція повинна знати граматичні правила даного формату. Парсер працює як кінцевий автомат станів, що репрезентують поточний момент в обробці дерева партії. Реалізація даного підходу виглядає як набір рекурсивних функцій, що розпізнають та повертають в обробленому форматі кожна свою структуру даних формату SGF.

Серед структур, що розглядалися у файлах SGF основними є такі:

* Файл – складається з дерев партій.
* Партія – складається зі списку піддерев.
* Піддерево – представляється рекурсивно як список піддерев або пустий список.
* Вузол – складається зі списку властивостей. Може містити піддерева.
* Властивість – задається як пара двох елементів: ідентифікатора властивості та значення властивості.
* Ідентифікатор властивості – у форматі SGF це дві великі латинські літери, що дають назву властивості.
* Значення властивості – число або рядок. Кінцевий елемент SGF-файлу.

Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати, до того ж вона займає менше пам’яті комп’ютера. Це не окремий тип колекцій, особливо враховуючи, що їм не обов'язково бути саме списками. Доступні також такі вбудовані структури даних, як вектор, хеш-словник, набір та інші. При спробі отримати з порожньої колекції послідовність її елементів (викликом seq) повертається nil. При спробі отримати з послідовності (на її останньому елементі) залишок (rest) буде повернуто іншу логічну послідовність. Це дозволяє послідовностям та протоколу послідовностей бути лінивими.

Частіше доцільно використовувати іншу структуру даних, що репрезентує поточний стан дошки. Ця структура вже не дерево, вона має вигляд матриці значень клітинок дошки (тобто ця клітинка пуста, зайнята білим каменем або зайнята чорним каменем). Ця структура більш зручна, якщо треба аналізувати поточний стан дошки, адже дерево не несе в собі подібну інформацію. Дерево зберігає зміни в дошці на момент кожного ходу, а для пошуку по стану дошки зручніше використовувати структуру, що може зберігати кожен стан дошки окремо. Також така структура даних використовується методом пошуку шаблону, саме на її основі рахується хеш поточного стану дошки. Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати.

Лістинг 3.2. Функція tree2board

(defn tree2board [tree]

(let [board (vec (repeatedly 19 (fn [] (vec (repeat 19 :empty)))))

player (mkplayer)]

(reduce (fn [board move]

(let [y (char2index (first move))

x (char2index (second move))]

(assoc board x

(assoc (board x) y (stone (player))))))

board tree)))

Ця функція повертає вектор рядків дошки, що складаються з послідовності ідентифікаторів, що репрезентують поточний стан клітинки дошки. Для цього використовуються 3 ідентифікатори:

* :empty;
* :black;
* :white.

Для візуалізації поточного стану дошки використовується функція print-board, що виводить на монітор дошку з каміннями гравців. Вивід цієї функції схожий на вивід дошки у програмі GnuGo. Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати. Це не окремий тип колекцій, особливо враховуючи, що їм не обов'язково бути саме списками. Доступні також такі вбудовані структури даних, як вектор, хеш-словник, набір та інші. При спробі отримати з порожньої колекції послідовність її елементів (викликом seq) повертається nil. При спробі отримати з послідовності (на її останньому елементі) залишок (rest) буде повернуто іншу логічну послідовність. Це дозволяє послідовностям та протоколу послідовностей бути лінивими.

Найочевиднішими структурами даних для збереження партій у Го, як вже було з’ясовано, є дерево та матриця. Однак при деяких розрахунках подібні структури дуже неоптимальні, саме тому була створена інша структура даних, що зберігає дошку як набір груп.

Зрозумілими перевагами подібної структури даних є:

* Спрощення розрахунку того, живі групи каменів, чи мертві.
* Зменшення витратності роботи більшості алгоритмів, зв’язаних з розрахунками зв’язків між сусідніми каменями.

Недоліками є складність впорядковування подібної структури даних, адже після додання будь-якого каменя може скластися ситуація, коли необхідно перераховувати не тільки одну локальну групу, але і її сусідів. Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати.



Рис. 3.2. Приклад розподілу ділянки дошки на групи каменів

У цьому алгоритмі також можливо додати паралельність виконання функції. Обробка гілки піддерева може виконуватися у окремому потоці. Таке рішення створює проблему: можливо виконати надмірні обчислення, якщо піддерево, яке можливо не оброблювати обраховується раніше, ніж піддерево, яке містить шуканий шлях.

## 3.3. Модуль політики

Більшість алгоритмів потребує функцію генерування можливих ходів (ходи, що не є заборонені правилами гри) та функцію евристичної оцінки поточного стану дошки. Розробка функції оцінки стану виходить за рамки даної роботи, тому замість неї буде використовуватися випадкова величина.

Функція генерування можливих ходів (movegen у коді) використовує поточний стан дошки board та колір гравця, що робить крок player. Вона повертає список можливих ходів у вигляді пар координат. Ця функція повинна розуміти правило зняття з дошки каменів та правило Ко.

Розрахунок нового рівня можна легко зробити паралельним. Достатньо шукати максимум кожного піддерева у окремому потоці. В мові Clojure це можна зробити використовуючи функцію pmap. Ця функція працює як звичайний map, але запускає функції у окремих потоках, повертаючи Future. Використовуючи об’єкт такого типу можливо дізнатися коли закінчиться виконання відповідної функції та отримати її результат.

Така реалізація дозволяє просто зробити виконання модулю політики паралельним, але вона не дає змогу контролювати цей процес. Якщо є така задача, то паралелізм можна реалізувати самостійно, використовуючи ті самі об’єкти типу Future.

Лістинг 3.3. Псевдокод модуля політики

function politics(node, depth) ◁ returns integer - value of best play

if node is a terminal node or depth ≤ 0 then

return the heuristic value of node

end if

𝛼 ← −∞

for child in node do ◁ evaluation is identical for both players

𝛼 ← max (𝛼, -policits(child, depth-1))

end for

return 𝛼

end function

Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати.

Також функція генерування можливих ходів (movegen у коді) використовує поточний стан дошки board та колір гравця, що робить крок player. Вона повертає список можливих ходів у вигляді пар координат. Ця функція повинна розуміти правило зняття з дошки каменів та правило Ко.

## 3.4. Альфа-бета відсічення

Метод Негамакс генерує повне дерево гри. Тобто він розглядає усі варіанти ходів та усі можливі варіанти відповідей на ці ходи. Саме тому він дуже обчислювально витратний. Однією з можливих оптимізацій цього методу може бути відсічення вершин, що можливо не розглядати. Тобто не рахувати ті піддерева, обчислення яких не змінить результат.

Альфа-бета відсічення – модифікація методу Мінімакс, алгоритму пошуку, що зменшує кількість вузлів, які необхідно оцінити в дереві пошуку мінімаксного алгоритму і при цьому дозволяє отримати ідентичний результат. Цей алгоритм використовується в програмуванні ігор, де грають два гравці (хрестики-нулики, шахи, Го).

У цьому алгоритмі також можливо додати паралельність виконання функції. Обробка гілки піддерева може виконуватися у окремому потоці. Таке рішення створює проблему: можливо виконати надмірні обчислення, якщо піддерево, яке можливо не оброблювати обраховується раніше, ніж піддерево, яке містить шуканий шлях. Дивлячись на приклад псевдокоду методу Альфа-бета відсічення на лістингу 3.3, зрозуміло, що доцільно тіло циклу, що виконується для кожного можливого хода, виконувати як паралельні процеси, що не залежать один від одного.

Використовуючи цей алгоритм, програма повністю припиняє оцінювати піддерево, якщо знайшла доказ того, що всі ходи в піддереві гарантовано гірші, ніж якийсь хід, оцінений раніше. Таке піддерево та ходи не потребують подальшого розглядання.

В основі роботи алгоритму стоять два значення: α – найкраще значення маршруту для максимізуючого гравця та β – найкраще значення маршруту для мінімізуючого гравця. Зрозуміло, що α прямує до плюс нескінченності, а β прямує до мінус нескінченності. Дерево партії, що побудоване на основі SGF-файлу виходить дуже надлишковим. Воно зберігає багато інформації, що потрібна людям, але не потрібна для пошуку на комп’ютері у такій структурі даних. Найчастіше для пошуку потрібні тільки ті властивості у дереві гри, що відповідають позиціям каменів гравців, усі ж інші властивості – зайві. Тому була реалізована функція simplify, що фільтрує та спрощує дерево гри, залишаючи тільки послідовність ходів гравців. З такою структурою даних набагато легше працювати.

Приклад дерева та роботи методу Альфа-бета відсічення продемонстрований на рисунку 3.3. На ньому квадратні вершини – це вершини, де приймає рішення максимізуючий гравець, круглі вершини – де мінімізуючий. Позначені сірим та закреслені рисками ті піддерева, розгляд яких не плине на вирішальний результат, тому їх обчисленням можна знехтувати.

Використовуючи цей алгоритм, програма повністю припиняє оцінювати піддерево, якщо знайшла доказ того, що всі ходи в піддереві гарантовано гірші, ніж якийсь хід, оцінений раніше. Таке піддерево та ходи не потребують подальшого розглядання.

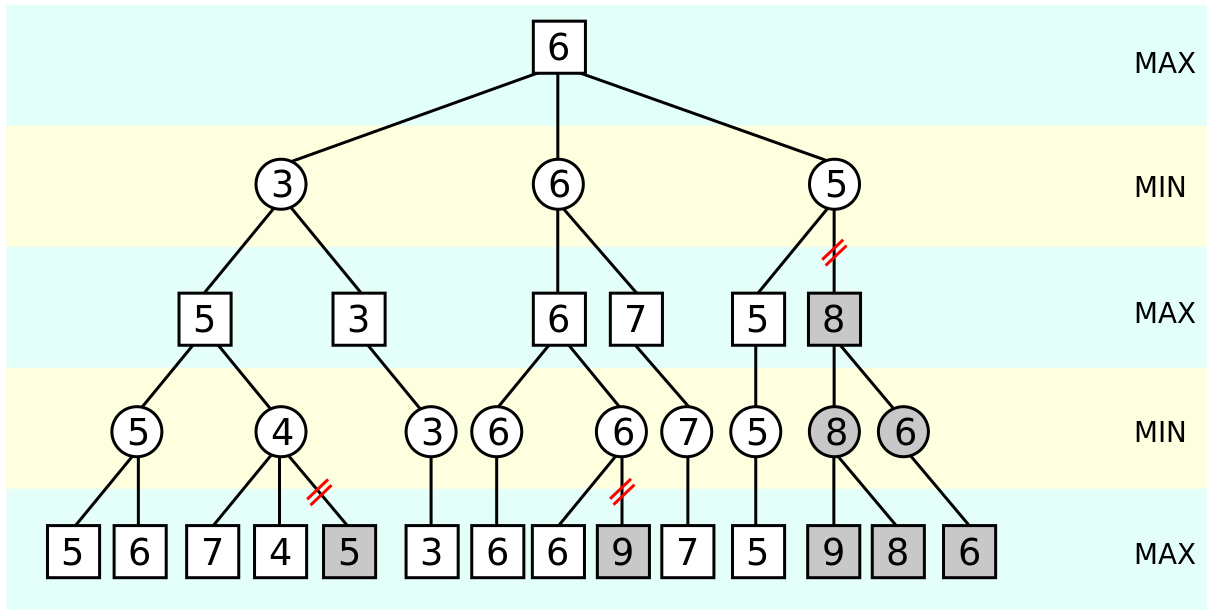


Рис. 3.3. Ілюстрація роботи алгоритму Альфа-бета відсічення

Альфа-бета алгоритм, при найкращому порядку ходів, побудує значно менше дерево перебору. Кількість вузлів приблизно дорівнює кореню квадратному з числа позицій, що переглядаються при повному переборі. Альфа-бета відсічення дуже чутливе до порядку ходів, тому потрібно врахувати, що при найгіршому порядку ходів, тобто коли відсічення за β розглядає останній хід, Альфа-бета відсічення прогляне стільки ж позицій, скільки і Негамакс. Швидкість прорахунку на практиці також дуже залежить від можливого діапазону оцінок.

У цьому алгоритмі також можливо додати паралельність виконання функції. Обробка гілки піддерева може виконуватися у окремому потоці. Таке рішення створює проблему: можливо виконати надмірні обчислення, якщо піддерево, яке можливо не оброблювати обраховується раніше, ніж піддерево, яке містить шуканий шлях. Дивлячись на приклад псевдокоду методу Альфа-бета відсічення на лістингу 3.4, зрозуміло, що доцільно тіло циклу, що виконується для кожного можливого хода, виконувати як паралельні процеси, що не залежать один від одного.

Кількість вузлів приблизно дорівнює кореню квадратному з числа позицій, що переглядаються при повному переборі. Альфа-бета відсічення дуже чутливе до порядку ходів, тому потрібно врахувати, що при найгіршому порядку ходів, тобто коли відсічення за β розглядає останній хід, Альфа-бета відсічення прогляне стільки ж позицій, скільки і Негамакс. Швидкість прорахунку на практиці також дуже залежить від можливого діапазону оцінок.

Лістинг 3.4. Псевдокод методу Альфа-бета відсічення

function AlphaBeta(color, depth, α, β)

if depth = 0 then

return Evaluate(color)

end if

moves ← GenerateMoves

for move in moves do

makeMove(move)

eval ← -AlphaBeta(-color, depth-1, -β, -α)

unmakeMove(move)

if eval ≥ β then

return β

end if

if eval > α then

α ← eval

if depth = defaultDepth then

bestmove ← move

end if

end if

end for

return α

end function

## 3.5. Порівняння з шаблоном

Шаблоном у грі Го називається деяка послідовність ходів на локальній частині дошки, що часто використовується і приносить приблизно однакову користь для обох гравців.

Ефективне розпізнавання шаблонів у грі Го та їх грамотне використання є вирішальним як для гравця-людини, так і для програми. На поточний момент було створено базу шаблонів різних етапів партії на основі професійних партій в Го. Така база використовуються майже усіма програмами, що грають у Го.



Рис. 3.4. Ілюстрація пошуку за квадратним шаблоном

Пошук шаблону може бути як використовуючи квадратні ділянки, так і більш спеціалізованим способом – використовуючи ділянку у вигляді ромбу.



Рис. 3.5. Ілюстрація пошуку за шаблоном у вигляді ромбу

Також цікавим способом роботи з шаблоном є використання графів. Цей спосіб не розглядається в даній дипломній роботі, однак для порівняння він наведений на рисунку нижче.



Рис. 3.6. Ілюстрація пошуку за шаблоном на основі графа

Створення такої бази не входить в завдання даного дипломного проекту, тому методи пошуку за шаблоном використовують заздалегідь згенеровану базу. Однак розглянемо псевдокод можливого методу генерації подібної бази шаблонів гри Го. В даному прикладі шаблон має заданий розмір 5 на 5. До того ж, для спрощення пошуку шаблонів та їх подальшого використання, нехай перший хід у шаблоні буде завжди в його центр.

Ефективне розпізнавання шаблонів у грі Го та їх грамотне використання є вирішальним як для гравця-людини, так і для програми. На поточний момент було створено базу шаблонів різних етапів партії на основі професійних партій в Го. Така база використовуються майже усіма програмами, що грають у Го.

Лістинг 3.4. Псевдокод алгоритму створення бази шаблонів за базою партій в Го

for each game record R in the game record collection do

for each move M in R do

Play M on the game board

Obtain the 5-by-5 region R centered by M

Rotate and flip R into its canonical form

if R is not in our pattern database then

Add R into the pattern database

Set frequency number of R to be 0

end if

Increase the frequency number of R by 1

end for

end for

Для пошуку по такій базі шаблонів доречно використовувати хешування. Якщо метод хешування задовольняє умові адитивності (тобто маючи хеш послідовності можна легко обрахувати та знайти хеш нової послідовності, отриманої декількома ходами з початкової, просто додаючи оператором XOR хеши нових ходів), то пошук по базі шаблонів буде значно спрощено.

Відповідність хешів надає інформацію про входження послідовності ходів партії до шаблону з бази. Якщо ж використовувати метод хешування, що не залежить від послідовності, а тільки від результуючого набору каменів, то можна покращити пошук, адже тепер по результуючому набору каменів можна буде знайти різні піддерева, що привели до нього. Ефективне розпізнавання шаблонів у грі Го та їх грамотне використання є вирішальним як для гравця-людини, так і для програми. На поточний момент було створено базу шаблонів різних етапів партії на основі професійних партій в Го. Така база використовуються майже усіма програмами, що грають у Го.

Одним з найпопулярніших методів хешування для настільних ігор є хешування Зобріста. Якщо говорити про це хешування, то порівняння хешів зводиться до виконання операції XOR над хешем шаблону та хешем підпослідовності поточного стану партії. Це є дуже ефективним способом знаходження шаблонів, однак недолік полягає у тому, що таку перевірку треба робити для всіх станів дошки партії починаючи з якогось стану. Тобто цей метод має виконувати повний перебір усіх підпослідовностей та усіх шаблонів з бази. Для знаходження шаблонів у партії з 200 ходів, використовуючи базу даних з 1000 шаблонів треба виконати 200 тисяч операцій XOR, не говорячи про рахування хешу Зобріста. Однак, це є недоліком усіх подібних методів хешування.

Алгоритм пошуку підходящого шаблону складається з таких кроків:

1. Розбиття поточного стану дошки на локальні ситуації, до кожної з яких можна застосувати метод пошуку шаблону.
2. Обчислити хеш Зобріста кожної ситуації.
3. Для кожної ситуації, з усієї бази шаблонів знайти ті, що частково співпадають з нею. Тобто ситуаціє є частиною послідовності ходів даного шаблону.
4. Аналіз знайдених шаблонів та вибір серед них найбільш доречних.
5. Прийняття рішення, щодо подальшого кроку на основі знань, отриманих з знайдених шаблонів.

Більшість пунктів виходять за рамки даної роботи. Розроблювана система містить у собі реалізацію функції, що рахує хеш Зобріста по поточному стану дошки або її частини і функцію пошуку входження шаблону локальної ситуації у базу шаблонів партій з Го.

## 3.6. Хешування Зобріста

Одним із методів пошуку деякого положення каменів на дошці може бути хешування. Якщо використовувати хешування, по якому можна легко дізнатися, що стоїть на якій клітинці, то функція пошуку деякого зразка на дошці буде простою. Прикладом такого хешування може бути хешування Зобріста.

Хешування Зобріста – це хеш-функція, що використовується в комп’ютерних програмах, які грають в абстрактні настільні ігри, такі як шахи або Го і використовують транспозиційні таблиці, особливий вид хеш-таблиць, що індексуються позиціями дошки і використовується, щоб уникнути аналізу однієї і тієї ж самої позиції декілька разів.

Хешування Зобріста починається з генерування випадкового бітового рядка для кожного можливого елемента настільної гри, тобто для кожної комбінації фігури і положення. Наприклад для шах це 12 штук та 64 позицій дошки, тобто 768 різних бітових рядків; для Го це 2 фігури та 361, тобто 722 рядки. Використовуючи такі рядки будь-яка конфігурація дошки може бути розбита на незалежні компоненти фігура/положення, кожен з яких має відповідний бітовий рядок. Результуючий хеш дошки рахується використовуючи побітове XOR між усіма бітовими рядками усіх компонент, тобто фігур та їх положень, на дошці.

Такий самий метод можна використовувати для підрахунку хешу частини дошки, а не усієї. Цей метод дуже універсальний, він підходить майже для всіх настільних ігор. Якщо використовувати хешування, по якому можна легко дізнатися, що стоїть на якій клітинці, то функція пошуку деякого зразка на дошці буде простою. Прикладом такого хешування може бути хешування Зобріста.

Безсумнівним доводом використання такого методу хешування для пошуку за базою шаблонів є те, що використовуючи лише одну операцію XOR та маючи хеші можливо перевірити їх відношення між собою та зробити відповідні висновки.

У даній системі хешування Зобріста реалізовано функцією (zobrist board), що приймає стан дошки або її частину (тобто структуру даних типу board) і повертає число типу bigint. Якщо використовувати хешування, по якому можна легко дізнатися, що стоїть на якій клітинці, то функція пошуку деякого зразка на дошці буде простою. Прикладом такого хешування може бути хешування Зобріста.

## 3.7. Метод Монте-Карло пошуку в дереві

Методи Монте-Карло – загальна назва групи чисельних методів, заснованих на одержанні великої кількості реалізацій стохастичного (випадкового) процесу, який формується у той спосіб, щоб його ймовірнісні характеристики збігалися з аналогічними величинами задачі, яку потрібно розв’язати. Використовується для розв’язування задач у фізиці, математиці, економіці, оптимізації, теорії управління тощо. Метод Монте-Карло – це метод імітації для приблизного відтворення реальних явищ. Він об’єднує аналіз чутливості (сприйнятливості) і аналіз розподілу ймовірностей вхідних змінних. Цей метод дає змогу побудувати модель, мінімізуючи дані, а також максимізувати значення даних, які використовуються в моделі. Побудова моделі починається з визначення функціональних залежностей у реальній системі. Після чого можна одержати кількісний розв’язок, використовуючи теорію ймовірності й таблиці випадкових чисел.



Рис. 3.7. Ілюстрація роботи методу Монте-Карло для розрахунку значення числа Пі

Для пошуку у структурах даних гри Го існує метод Монте-Карло пошуку в дереві. Він генерує велику кількість випадкових партій до кінця гри. Потім він рахує статистику виграшу та програшу у цих іграх. Використовуючи цю статистику, він приймає рішення щодо наступного ходу, вибираючи хід, що має найбільшу вірогідність виграшу.

Як видно на плакаті “Схема алгоритму методу Монте-Карло пошуку у дереві”, цей метод складається з чотирьох кроків:

1. Вибір – перший крок методу Монте-Карло пошуку в дереві. На цьому кроці алгоритм обирає найбільш підходящу вершину дерева для продовження моделювання. Функція, що реалізує цей крок виконана у вигляді рекурсивного спуску у дереві, обираючи вершини з найбільшим значенням value у кожній. Значення value також перераховується кожен раз на кроці підйому.
2. Така неієрархічна кластеризація полягає в розділенні набору даних на певну кількість окремих кластерів. Існує два підходи. Перший полягає у визначенні меж кластерів як найбільш щільних ділянок в багатовимірному просторі вихідних даних, тобто визначення кластера там, де є велике "згущення точок". Другий підхід полягає в тому щоб мінімізувати відмінності об'єктів.
3. Розширення – єдиний крок, на якому створюється нова вершина. Нова вершина створюється випадковим чином як продовження вершини, обраної на попередньому кроці. Вершина обирається випадковим чином зі списку можливих ходів, згенерованих функцією genmove. Нова вершина додається до дерева гри зі стандартними показниками виграшу та з пустим піддеревом.
4. Моделювання – процес створення у пам’яті піддерева великої кількості випадковим чином згенерованих партій з Го. Як і на попередньому етапі ходи обох гравців обираються випадковим чином зі списку можливих ходів або ж використовуючи поле політики. Для спрощення подальшого переобчислення ще на етапі моделювання дерева варто зберігати winrate – статистику виграшів та програшів першого гравця (того, за якого грає комп’ютер). Для методу Монте–Карло пошуку в дереві потрібна статистика включає в себе тільки відношення виграшів до програшів у піддереві вершини, що розглядається. Інші модифікації цього алгоритму потребують більшу кількість параметрів для розглядання та збереження у вузлах дерева.
5. Ітеративний процес. Обчислюються центри кластерів, якими потім і далі вважаються координатні середні кластерів. Об'єкти знову перерозподіляються. Процес обчислення центрів і перерозподілу об'єктів продовжується.
6. Переобчислення – останній етап методу Монте-Карло пошуку в дереві. Після моделювання великої кількості ігор створену статистику виграшів та програшів потрібно зберегти для подальшого використання. Враховуючи те, що деякі значення winrate у всіх вершинах вже є (вони були створенні на попередніх ітераціях алгоритму), на даному кроці алгоритму потрібно оновити ці значення, тобто – переобчислити їх.

Після останнього кроку алгоритму виконання переходить знову до першого. Подібне зациклення відбувається до того моменту, поки не буде прийнято рішення зупинити роботу алгоритму. Подібне рішення може бути прийнято на основі кількості змодельованих ігор, поточній глибині пошуку або ж на основі розміру дерева у пам’яті або обчислювальної витратності останньої ітерації алгоритму. Для пошуку у структурах даних гри Го існує метод Монте-Карло пошуку в дереві. Він генерує велику кількість випадкових партій до кінця гри. Потім він рахує статистику виграшу та програшу у цих іграх. Використовуючи цю статистику, він приймає рішення щодо наступного ходу, вибираючи хід, що має найбільшу вірогідність виграшу.

## 3.8. Метод верхньої оцінки значущості для дерева

Існують методи, що покращують роботу попереднього методу. Вони змінюють функцію оцінки значущості вузлів, що змінює порядок обходу та створення дерева варіантів партій у пам’яті. Один із прикладів може бути метод UCT – варіація методу Монте-Карло для пошуку в дереві що використовує змінену функцію верхньої оцінки значущості для дерева.

На основі цих значень, використовуючи функцію, показану нижче, алгоритм вираховує значення значущості вузлів дерева. На першому етапі вибору шляху для продовження аналізу використовуються саме ці значення:



де winrate – значення проценту виграшу у піддереві, що розглядається, visits – значення кількості візитів поточної ноди при проходженні кожного кроку алгоритму.

Цей метод потребує розширення додаткової інформації, що зберігається в структурі дерева варіантів гри. Він додає нове значення до статистики виграшів та програшів у кожний вузол – кількість раз, що цей вузол був відвіданий.

Така нова функція додає до статистики виграшів нове значення, що зменшується кожен раз, коли дану вершину було обрано на якійсь ітерації і збільшується кожен раз, коли було обрано якусь іншу дочірню вершину батьківської вершини (тобто іншу вершину, що має спільну батьківську вершину і яка знаходиться на тому самому рівні глибини). Таке змінення оцінювальної функції дозволяє сконцентруватися на варіантах, які дають велику ймовірність виграшу у той же час не забуваючи про створення нових варіантів гри. Таке змінення оцінювальної функції дозволяє сконцентруватися на варіантах, які дають велику ймовірність виграшу у той же час не забуваючи про створення нових варіантів гри. Таке змінення оцінювальної функції дозволяє сконцентруватися на варіантах, які дають велику ймовірність виграшу у той же час не забуваючи про створення нових варіантів гри.

# 4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА МЕТОДИКА ТЕСТУВАННЯ

Системи, аналогічні створюваній, завжди є частинами більших систем що грають в Го або аналізують партії з Го. Саме тому дуже важко порівнювати їх швидкодії, адже їх не можна використовувати без глобального контексту системи. У розробленій системі це зробити можливо, адже у ній методи пошуку реалізовані у вигляді окремих модулів.

Тому краще оцінювати і порівнювати розроблену систему з аналогічними по розміру та простоті використання.

* 1. Порівняння розміру початкового коду систем пошуку у структурах даних гри Го

Розроблена система, на відміну від Master Go, поставляється у вигляді бібліотеки, тобто jar-файлу, який можливо використовувати в усіх мовах програмування, що підтримують модулі мови програмування Java. Тому це можливо майже в усіх мовах, що використовують платформу Java.

Інші ж системи розроблені використовуючи мову програмування C++, до того ж вони не створені у вигляді бібліотек. Тому для використання їх як окремих модулів у своїй системі, спочатку треба виділити їх у бібліотеку, що може бути досить складною задачею.

Розроблена система розділена на модулі, кожен з яких як окремо, так і вкупі з іншими можливо використовувати без втрати швидкодії. Там, де можливо буде проведено порівняльний аналіз розміру відповідних модулей так їх швидкодії. Проведений аналіз залежіть від версії розробленої системи, адже у наступних її варіаціях можлива зміна не тільки самих алгоритмів а і їх реалізації. Інші ж системи розроблені використовуючи мову програмування C++, до того ж вони не створені у вигляді бібліотек. Тому для використання їх як окремих модулів у своїй системі, спочатку треба виділити їх у бібліотеку, що може бути досить складною задачею.

У системі Fuego модулі розроблені так, що їх можливо використовувати окремо. Fuego складається з п’яти окремих бібліотек:

* GtpEngine – реалізація протоколу GTP.
* SmartGame – додаткові функції, що можуть бути використані у різних настільних іграх.
* Go – класи та функції, специфічні для Го.
* SimplePlayers – проста реалізація гравця для Го.
* GoUct – реалізація методу UCT для Го.

У розробленій системі модулі розбиті приблизно таким самим чином, але вона має більше методів для пошуку у структурах даних гри Го. Однак їй бракує системи, що реалізує протокол GTP.

GTP – це текстовий протокол, що найчастіше використовується програмами, що грають в Го для комунікації між собою. Він працює по мережі у режимі реального часу. При подальшому розвитку системи, цей протокол має бути реалізований, адже він є ключовим для комунікації між системами. Інші ж системи розроблені використовуючи мову програмування C++, до того ж вони не створені у вигляді бібліотек. Тому для використання їх як окремих модулів у своїй системі, спочатку треба виділити їх у бібліотеку, що може бути досить складною задачею.

Якщо порівнювати розміри програмних кодів продуктів, то програмний код Kombilo займає 1.9 мегабайт, програмний код Fuego займає 1.1 мегабайт, а програмний код розробленої системи займає близько 250 кілобайт. Однак таке порівняння не зовсім доречно, адже розглянуті системи мають зовсім різний рівень функціональних можливостей та кількість реалізованих методів.

* 1. Порівняння платформ систем пошуку інформації у структурах даних гри Го

Однією з суттєвих відмінностей розробленої системи безумовно є використання платформи Java. Саме завдяки такій реалізації є можливим використовувати бібліотеку у будь-якій програмі, що використовує можливості даної платформи. Розроблена система постає у вигляді модуля Java з назвою com.kpi.diploma.sgl. Наведемо приклад, у якому розроблена система використовується у програмах, що написані на мові програмування Java.

Лістинг 4.1. Використання розробленої системи у програмі на мові програмування Java

import com.kpi.diploma.sgl;

public class Main {

public static void main(String[] args) {

System.out.println("parsed tree: " +

sgl.sgf2tree("resources/game.sgf"));

}

}

Через те, що більшість систем, аналогічних розроблюваній не модульні і порівнювати їх безпосередню швидкодію буте досить важко, доречно буде порівняти загальну швидкодію мов, на яких написані порівнювані системи, тобто мов C++ та Java.

Для порівняння була обрана мова Java, а не Clojure, адже розроблена система Clojure все одно компілюється в Java байт-код та поставляється у вигляді jar-пакету.

Подібні результати свідчать про те, що загалом твердження “Java – повільна мова” це міф. Навіть зважаючи на те, що Java витрачає час на запуску JVM (Java Virtual Machine) – більшість тестів вона проходить досить швидко.

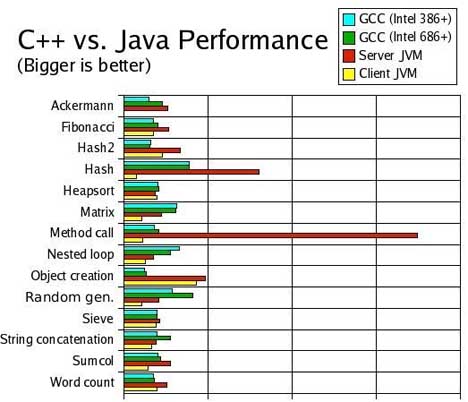


Рис. 4.1. Графік порівняння швидкодії мов програмування C++ та Java

Далі проведемо порівняльний аналіз популярних функцій між розроблювальною бібліотекою і бібліотекою Fuego.

Таблиця 4.1

Порівняльний аналіз функціоналу Fuego та розробленої системи

|  |  |  |
| --- | --- | --- |
|  | Fuego | Власна розробка |
| Кількість строчок коду | >10000 | 1700 |
| Розмір початкового коду | 1.1Mb | 250kb |
| Розмір програмної розробки | 3Mb | 570kb |
| Кількість класів | >200 | 17 |

**4.3. Методика розробки та тестування системи пошуку у структурах даних гри Го**

Основою для розроблювальної системи є мова, що має у собі REPL що значно впливає як на процес розробки, так і на процес тестування, адже така динаміка у мові дозволяє розроблювати та тестувати не тільки окремі модулі, але й окремі функції. Важливою особливістю REPL є те, що подібне середовище розробки має здатність зберігати, синхронізувати та наслідувати свій стан між різними середовищами.

REPL (Read-eval-print loop — цикл читання-обчислення-друку) — просте інтерактивне середовище програмування. Цей термін найчастіше вживається стосовно інтерактивного середовища мови програмування Lisp. В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві.

В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві. Назва read-eval-print loop походить від імен примітивів мови Lisp, що реалізують таку функціональність:

* Функція read читає один вираз і перетворює його в відповідну структуру даних у пам'яті.
* Функція eval приймає одну таку структуру даних і обчислює вираз, що відповідає їй.
* Функція print приймає результат обчислення виразу і друкує його користувачеві.

Щоб реалізувати REPL-середовище для певної мови, досить реалізувати три перелічені вище функції й об'єднати їх у нескінченний цикл. В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві.

REPL-середовище дуже зручне при вивченні нової мови, бо надає користувачеві швидкий зворотний зв'язок. Також подібне середовище дуже допомагає при тестуванні, адже існує можливість не тільки тестувати окремі функції, але і змінювати їх, довантажувати у вже працюючу систему.

Основна увага при тестуванні була зосереджена на тестах модулів та функцій, тобто unit-тестуванню. Було створено набір тестів до кожної функції пошуку та роботи з SGF-файлами.

Лістинг 4.2. Приклад unit-тестів для функцій, що працюють з SGF-файлами

(deftest sgf2tree\_test

(is (= first-result (sgf2tree first-file)))

(is (= second-result (sgf2tree second-file))))

(deftest tree2board\_test

(is (= first-board (tree2board first-result)))

(is (= second-board (tree2board second-result))))

***Приклад роботи використовуючи REPL середовище***

Однією з найбільших перевагах при роботі з динамічними мовами програмування є можливість будувати систему і одночасно взаємодіяти з нею. Для того, щоб зробити це можливим спочатку треба мати можливість перевизначати частини програми у той час, поки вона виконується, однак деякі частини мови Clojure не дозволяють працювати настільки інтерактивно. Наприклад еффект зміни макросу в коді не буде прийнятий до виконання до того моменту, пока код не буде знов відкомпільований. Також усі зміни методів defrecord чи deftype не будуть мати ніяких ефектів на існуючі об’єкти цих типів. В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві.

Можливості, що надає Clojure для завантаження коду з файлів не достатньо для того, щоб вирішити ці всі проблеми. Тому було створена спеціальна версія стандартних утиліт, що входять у пакунок з Clojure, що називаються tool.namespace щоб зробити більш інтелектуальним виклик методу require, щоб він впізнавав залежності між простірами імен та перезавантажував їх по необхідності. В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві.

Але ці методи з простіру імен tools.namespace це тільки частина рішення, адже важливо мати впевненість у тому, що версія програми, що виконається співпадає та відноситься саме до того коду, що збережений у файлах, що використовуються. Тобто важливо не тільки те, що програма виконається саме з останньої версії початкового коду, що написаний, але і те, що стан, у якому знаходиться програма створений саме поточною версією коду. Використовуючи REPL середовище дуже просто отримати стан програми, що не може бути досягнутий поточною версією коду, тобто мати неконсистентний стан.

Тому після кожної значної зміни коду треба перезапускати програму. Однак було б дуже не привабливо для цього перезавантажувати кожен раз віртуальну машину Java і перезавантажувати увесь код модуля чи навіть всього проекту. Для цього важливо розробити програму так, щоб існував спосіб швидко запинити її виконання, відкинути увесь тимчасовий стан і знов її запустити, щоб повернутися до відповідного стану. При правильній реалізації увесь цей процес може зайняти швидше за одну секунду.

Для того, щоб розробити таку систему треба розроблювати модулі системи як тимчасові об’єкти. Замість того, щоб робоча програма були синглтоном і була прив’язана до віртуальної машини, можна розробити модуль так, щоб у будь-який момент часу була можливість відкинути стару та неактуальну версію модуля, та почати використовувати нову. Такий підхід схожий до того, як здійснюється керування віртуальними машинами у хмарному середовищі або на підхід у мові програмування Erlang.

Подібний підхід потребує дуже уважної розробки. По-перше ніякого глобального стану не повинно існувати – адже глобальні змінні у будь-якому місці знищують усю робочу систему. По-друге усі ресурси, що використовує система повинні зручно звільнятися, для спрощення руйнування тимчасового об’єкту робочої програми. Розглянемо приклад подібної розробки. Однак було б дуже не привабливо для цього перезавантажувати кожен раз віртуальну машину Java і перезавантажувати увесь код модуля чи навіть всього проекту. Для цього важливо розробити програму так, щоб існував спосіб швидко запинити її виконання, відкинути увесь тимчасовий стан і знов її запустити, щоб повернутися до відповідного стану. При правильній реалізації увесь цей процес може зайняти швидше за одну секунду.

Наприклад ефект зміни макросу в коді не буде прийнятий до виконання до того моменту, пока код не буде знов відкомпільований. Також усі зміни методів defrecord чи deftype не будуть мати ніяких ефектів на існуючі об’єкти цих типів. В такому середовищі користувач може вводити вирази, які середовище одразу обчислить, а результат обчислень відобразить користувачеві.

Для створення об’єкту, що репрезентує систему потрібно створити конструктор. Опціональними параметрами у такого конструктора будуть параметри системи, наприклад база даних, відкриті файли та мережеві з’єднання. Створення такої системи, це не те ж саме, що її запуск – створення повинно бути без побічних ефектів. Найчастіше конструктор системи створює об’єкти усіх компонент, від яких він залежіть і повертає мапу відповідних об’єктів. Гарною практикою буде створення різних конструкторів для різних середовищ розробки – для тестування, для розробки.

З такою мапою, що грає роль контексту, усі глобальні об’єкти становляться локальними. Складність додає тільки постійна необхідність передавати цей контекст в усі функції, але з допомогою лексичних замикань подібний додатковий параметр зникає майже з усього коду. До того ж більшість функцій не потребує увесь контекст – найчастіше достатньо передати його підмножину. Однак було б дуже не привабливо для цього перезавантажувати кожен раз віртуальну машину Java і перезавантажувати увесь код модуля чи навіть всього проекту. Для цього важливо розробити програму так, щоб існував спосіб швидко запинити її виконання, відкинути увесь тимчасовий стан і знов її запустити, щоб повернутися до відповідного стану. При правильній реалізації увесь цей процес може зайняти швидше за одну секунду.

Далі потрібно створити функції що запускають та зупиняюсь робочу систему. Найкраще їх розроблювати таким чином, щоб вони виглядали як прості функції – тобто повертали новий стан, що відповідає програмі що працює, або що зупинилася, але вони також потребують виконання операцій з побічними ефектами – відкриття або закриття посилання на базу даних чи запуск веб-серверу. Початкове створення подібних функцій може бути нетривіальним, однак після декількох спроб можна отримати функціонал, котрий буде працювати і не буде потребувати подальших змін протягом усього процесу розробки.

Однак було б дуже не привабливо для цього перезавантажувати кожен раз віртуальну машину Java і перезавантажувати увесь код модуля чи навіть всього проекту. Для цього важливо розробити програму так, щоб існував спосіб швидко запинити її виконання, відкинути увесь тимчасовий стан і знов її запустити, щоб повернутися до відповідного стану. При правильній реалізації увесь цей процес може зайняти швидше за одну секунду.

Після того, як створено функції init та start, пишеться функція go, як комбінація вже створених двох:

Лістинг 4.2. Реалізація функції go

(defn go

"Ініціалізація поточного середовища розробки та його запуск."

[]

(init)

(start))

Насамкінець, потрібно розробити функцію reset – серце розроблювальної системи. Вона визначається використовуючи функцію з простіру імен clojure.tools.namespace.repl/refresh, а саме:

Лістинг 4.3. Реалізація функції reset

(defn reset []

(stop)

(refresh :after 'user/go))

Ця функція може у будь-який час бути викликана як у коді, так у середовищі REPL для того, щоб:

* Зупинити поточний робочий об’єкт системи.
* Перезавантажити усі файли з початковим кодом, що були змінені.
* Створити та запустити об’єкт нової робочої програми.

Після розроблення усіх цих допоміжних функцій розробка та тестування виглядають наступним чином:

* Створення нового об’єкту робочої програми використовуючи функцію reset.
* Після будь-якої зміни у початковому коді, котру необхідно протестувати чи просто перевірити, виконується перезапуск програми функцією reset.

***Недоліки***

Основним недоліком є те, що будь-яка помилка у будь-якому місці початкового коду завадить загрузити усю програму, навіть синтаксична помилка.

Гіршою проблемою є те, що якщо на етапі завантаження системи буде знайдена помилка компіляції – не буде запущено усе середовище розробки REPL. Подібну проблему можна вирішити, якщо за початкову версію при початку будь-якої розробки використовувати останню версію без помилок. Подібну проблему можна вирішити, якщо за початкову версію при початку будь-якої розробки використовувати останню версію без помилок.

# ВИСНОВКИ

У даному дипломному проекті було проаналізовано програми, що дозволяють шукати у структурах даних гри Го, проаналізовано найпопулярніші алгоритми для пошуку у таких структурах даних. На основі проаналізованих даних, було прийнято рішення створити систему пошуку у структурах даних гри Го, що буде закривати недоліки розглянутих систем.

Серед розглянутих методів були обрано такі: метод Монте-Карло пошуку в дереві, методи засновані на ймовірності та метод порівняння з шаблоном. Також було прийнято рішення реалізувати додаткові алгоритми, такі як метод Альфа-бета відсічення та метод верхньої оцінки значущості для дерева.

Також було проведено порівняльний аналіз платформ доступних для розробки та мов програмування на цих платформах. З усіх платформ було обрано платформу Java як одну з найбільш популярних та розповсюджених платформ на даний момент. Проаналізувавши мови програмування, що доступні на даній платформі було обрано мову програмування Clojure, адже вона найкраще підходить для розробки подібної системи через те, що являється Lisp-мовою та має гарну підтримку паралельного програмування.

Була розроблена система, що дозволяє працювати з SGF-файлами: відкривати їх, читати та перетворювати у формат внутрішнього дерева; система, що дозволяє використовувати три алгоритми пошуку в дереві гри Го: метод Альфа-бета відсічення, метод порівняння з шаблоном і метод Монте-Карло пошуку в дереві; і система, що дозволяє візуалізовувати результат виконання алгоритмів у вигляді простої дошки.

Порівнявши розроблену систему з існуючими аналогами було зроблено висновок, що вона не є такою ефективною, як аналоги, але потенційно може отримати більшу популярність та підтримку. До того ж для любителя не стільки принципова швидкодія системи, скільки те, чи зручно її використовувати та які можливості ця система надає.

Розроблена система є модульною бібліотекою, тобто вона дозволяє використовувати розроблені методи окремо один від одного. Ця властивість надає можливість для простої підтримки та розвитку розробленої системи.

# СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Wikipedia. Game of Go [Електронний ресурс]. — Режим доступу : https://en.wikipedia.org/wiki/Go\_(game). — Дата доступу :

грудень 2015.

1. Wikipedia. Zero-sum game [Електронний ресурс]. — Режим доступу : https://en.wikipedia.org/wiki/Zero-sum. — Дата доступу : грудень 2015.
2. Kombilo – Go database program [Електронний ресурс]. — Режим доступу : http://www.u-go.net/kombilo/. — Дата доступу :

грудень 2015.

1. Master Go at Sensei’s Library [Електронний ресурс]. — Режим доступу : http://senseis.xmp.net/?MasterGo. — Дата доступу :

грудень 2015.

1. Bi Go Software [Електронний ресурс]. — Режим доступу : http://bigo.baduk.org/. — Дата доступу : грудень 2015.
2. Wikipedia. Minimax Tree Search [Електронний ресурс]. — Режим доступу : https://en.wikipedia.org/wiki/Minimax. — Дата доступу : грудень 2015.
3. UCT at Sensei’s Library [Електронний ресурс]. — Режим доступу : http://senseis.xmp.net/?UCT. — Дата доступу : грудень 2015.
4. Programming Language – Groovy [Електронний ресурс]. — Режим доступу : http://groovy.codehaus.org/. — Дата доступу : грудень 2015.
5. The Scala Programming Language [Електронний ресурс]. — Режим доступу : http://www.scala-lang.org/. — Дата доступу : грудень 2015.
6. Clojure – Home [Електронний ресурс]. — Режим доступу : http://clojure.org/. — Дата доступу : грудень 2013.
7. Fuego [Електронний ресурс]. — Режим доступу : http://fuego.sourceforge.net/. — Дата доступу : грудень 2015.
8. Depth Stream [Електронний ресурс]. — Режим доступу: https://msdn.microsoft.com/en-us/library/jj131028.aspx. — Дата доступу: січень 2015.
9. Паттерн MVVM [Електронний ресурс]. — Режим доступу: http://professorweb.ru/my/WPF/documents\_WPF/level36/36\_5.php. — Дата доступу: січень 2015.
10. MVVM Pattern in WPF [Електронний ресурс]. — Режим доступу: http://www.mindscapehq.com/products/wpfelements/mvvm-pattern-in-wpf. — Дата доступу: січень 2015.
11. Гамма, Э. Паттерны проектирования. Приёмы объектно-ориентированного проектирования [Текст] / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидс. : пер. с англ. — СПб. : Питер, 2007 — 395 с.
12. Unit-тестирование средствами .NET [Електронний ресурс]. — Режим доступу: https://habrahabr.ru/post/98062/. — Дата доступу: січень 2015.
13. Unit Testing Framework [Електронний ресурс]. — Режим доступу: https://msdn.microsoft.com/en-us/library/ms243147(VS.80).aspx. — Дата доступу: січень 2015.
14. Принцип единственности ответственности [Електронний ресурс]. — Режим доступу: http://blog.byndyu.ru/2009/10/blog-post.html. — Дата доступу: січень 2015.
15. Принцип разделения интерфейса [Електронний ресурс]. — Режим доступу: http://blog.byndyu.ru/2009/11/blog-post\_19.html. — Дата доступу: січень 2015.
16. Алгоритм швидкого сортування (QuickSort) [Електронний ресурс]. — Режим доступу: http://itblogger.org.ua/algorutmu/alhorytm-shvydkoho-sortuvannya-quiqksort/. — Дата доступу: січень 2015.
17. Принципы проектирования классов (S.O.L.I.D.) [Електронний ресурс]. — Режим доступу: http://blog.byndyu.ru/2009/10/solid.html. — Дата доступу: січень 2015.

# ДОДАТКИ