Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра математики

ІГРИ З ОБМЕЖЕНИМ ГОРИЗОНТОМ І НУЛЬОВОЮ СУМОЮ НА ПРИКЛАДІ ГРИ В «ГО»

Курсова робота за спеціальністю ”Прикладна математика”6.040103

кандидат
фізико-математичних наук,
доцент Чорней Р. К.

_____
(*підпис*)
«____»_____ 2020р.

Виконала студентка
Федчук А. Д.
«____»_____ 2020р.

Київ - 2020

Тема: Ігри з обмеженим горизонтом і нульовою сумою на прикладі гри в «ГО»

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	07.10.2019	
2.	Огляд літератури за темою роботи.	11.10.2019	
3.	Аналіз літератури за темою роботи.	03.04.2020	
4.	Створення програми для реалізації поставленої задачі.	29.12.2019	
7.	Оформлення курсової роботи.	08.05.2020	
8.	Створення презентації до курсової роботи.	11.05.2020	

Зміст

Анотація

В даній праці містяться основні відомості про гру «ГО», розвиток та сучасний стан програм, за допомогою яких відтворюється ігровий процес між гравцем-людиною та ботом. Вона спрямована на аналіз процесу гри та його опис за допомогою конкретного методу. В якості такого методу розглядається ітераційний метод Марковських процесів прийняття рішення. Також у роботі є огляд необхідних теоретичних матеріалів для практичного аналізу та реалізації програми.

Вступ

Гра «ГО» вважається однією з найскладніших ігор для комп'ютера. Це викликано певними труднощами в описанні процесу для програми з оптимальним використанням ресурсів, проте вже досить тривалий час людство займається дослідженням цієї задачі. На сьогоднішній день існує вже декілька досить ефективних методів, які розробники використовують в свої застосунках – це методи Монте Карло та UCT. Однак упродовж останніх років з'явилися нові підходи до створення програм для «ГО». Першою програмою, яка реалізувала поєднання методу Монте Карло та техніки глибинного навчання з використанням нейроних мереж.

Мета і завдання дослідження полягає у розробці та реалізації алгоритму на базі ітераційного методу з теорії Марковських процесів прийняття рішення. Використання запропонованого методу дає змогу отримати додатковий інструмент для вивчення процесу гри та для реалізації успішної програми.

Робота складається з двох розділів. Перший розділ присвячено аналізу розвитку комп'ютерного «ГО» та самій грі. Спершу в ньому розглядається сама гра, її правила та основні проблеми; також наведений огляд методів, що використовуються в програмах з найвищим рівнем майстерності. У другому розділі описується теорія динамічного програмування, яка використовується в ітераційному методі та адаптація цих відомостей до процесу гри. Також створено та описано програму, яка приначенна для використання цього методу для пошуку оптимальних стратегій в грі.

1 РОЗДІЛ: Введення основних понять та відомості про гру «ГО»

У цьому розділі вводяться основні поняття, які стосуються гри та будуть використовуватися для вирішення даної задачі. Зокрема буде розглянуто шлях та розвиток програмування алгоритмів, для створення програм здатних грати в «ГО». Розділ містить основну інформацію про сучасний стан комп'ютерного «ГО» та аналіз актуальності цієї теми.

1.1 Визначення елементів та класифікації гри

Теорія ігор займається вивченням математичних моделей прийняття рішень у конфліктній ситуації [1]. Модель конфліктної ситуації це гра, де гравцям відповідають різні сторони конфлікту. Для зручності, визначимо головні поняття, які будуть використовуватися надалі. Гра – це сукупність правил, які її описують [2]. Основною складовою гри є хід – можливість вибору між альтернативами у визначених грою умовах. Ходи можуть бути особистими або випадковими. Приклад розігрування гри від початку та до кінця називається партією. У свою чергу, партія складається з виборів, тобто альтернатив, які обираються в конкретних ситуаціях даної партії. Для випадкових ходів альтернативи описуються ймовірностями переходів. Спільні принципи, яким підлягають вибори називаємо стратегією.

В подальшому, буде розглянута гра з двома гравцями. Таку гру називають грою з нульовою сумою, якщо сума доходів(прибутків) обох гравців наприкінці партії дорівнює нулю. Тобто все, що один гравець виграв - другий програв. В грі нульовою сумою ситуація рівноваги називається сідловою точкою – ситуація в якій верхня та нижня ціни гри співпадають. Нижньою ціною гри називаємо гарантований виграш першого гравця при будь-якій стратегії другого, а верхньою – гарантований програш другого гравця.

В грі з сідловою точкою оптимальний розв'язок буде отримано, якщо обидва гравця будуть притримуватися стратегій, що їй відповідають. Ці стратегії називають чистими. Також розрізняють мішані стратегії частковими випадками яких є чисті стратегії. В

грі з двома гравцями оптимальною обирається стратегія орієнтована на максимізацію доходу.

До того ж існує класифікація ігор за кількістю ходів. Гра з обмеженим горизонтом – це гра зі скінченою кількістю ходів.

1.2 Правила гри «ГО»

«ГО» – це гра з двома гравцями, де гравці по черзі ставлять на дошку білі або чорні фішки, які називаються *каменями*. Розмір поля може варіюватися, але класичний варіант це розмір дев'ятнадцять ліній на дев'ятнадцять. Камені ставляться на перетині ліній. Гравець, який грає чорними каменями ходить першим. В грі використовується поняття території – кількість обмежених, тобто захоплених, гравцем пунктів. Перемагає той, хто захопить більше території. Також є можливість захопити камені супротивника. Для цього їх потрібно оточити. Оточені камені знімаються з дошки та стають *полоненими*. Коли гравець пасує, він додає один камінь свого кольору до полонених каменів.

Проте можливі ситуації, коли камені не знімаються з дошки, проте все одно вважаються полоненими. Зазвичай група каменів вважається мертвою або полоненою, якщо вона оточена каменями супротивника та має не більше двох *очей*(вільний пункт оточений каменями одного кольору). Також територія може бути нейтральною. Нейтральна територія, це пункти, які нікому не приносять очок, бо вони оточені каменями двох гравців одночасно. Дуже часто це пункти, що залишаються на кордоні між територіями супротивників.

Існують різні види підрахунку очок наприкінці гри. Один з варіантів – підрахунок пунктів, які займають камені гравця додається до кількості незайнятих пунктів, обмежених цим гравцем. В другому варіанті підраховується сума вільних пунктів обмежених гравцем з урахуванням того, що полонені камені розташовують на вільних пунктах. Розглядатимуться ігри з другим способом підрахунку.

Під час гри можуть утворюватися пункти, в які ходити заборонено. Наприклад, заборонено робити *самовбивчий хід*. *Самовбивчим ходом* називається хід, при якому утворюється так зване положення *атарі* – ситуація в якій у каменя або групи каме-

нів не залишається останнє вільне *даме*, зазвичай це ситуація перед захопленням цієї групи. Самовбивчим ходом може бути спроба поставити камінь в *око* групи каменів супротивника. *Даме* – вільні пункти навколо каменю чи групи каменів по горизонталі чи вертикалі. При різному розташуванню на дошці група або камінь мають різну кількість даме. До того моменту, доки у групи чи каменю є вільне даме, вони залишаються на дошці. Також в грі заборонено повторення позицій – *правило ко*. *Ко* – позиція, в якій щойно був захоплений камінь. Така позиція дійсна, коли захоплюється лише один камінь супротивника та він може захопити лише один камінь в тому ж пункті.

Гра закінчується коли обидва гравці пасують підряд.

1.3 Розвиток та сучасна ситуація комп'ютерного «ГО»

«ГО» вже тривалий час є справжнім викликом для програмістів, адже гра характеризується чималою кількістю проблем, технічно складних для розв'язання. Не так складно написати гру, яка буде грати за правилами від початку та до кінця, складно написати її так, щоб вона грала добре [6]. Більшість конкурентних програм вимагає багато ресурсів, що стосуються різних аспектів гри, проте теж мають свої недоліки.

Розвиток комп'ютерного «ГО» розпочався ще в 1960-і роки. В 1980-х роках він суттєво просунувся вперед завдяки появі доступної техніки(комп'ютерів)та спонсорів щорічних турнірів з великою грошовою нагородою. Серед перших вдалих програм, які проявили себе на перших змаганнях були *Hsu* та *Liu's Dragon*. З 1989 по 1991 рік *Goliath* розроблена Марком Буном була лідером у всіх партіях, її наздоганяла *Go Intellect*, розроблена Кеном Ченом, а також *Goemate* Чена Цзіхін, яка станом на 2002 рік очолювала турнірну таблицю. В цей ж період декілька розробників об'єдналися в команду та перемогли більшість турнірів. *Goliath* гра в 1991 році вперше ообіграла переможця кубку турніру та трьома сильними гравцями з форою сімнадцять каменів. Наступним важливим етапом стали перемоги програми *Handtalk*, розробленої Ченом Цзіхіном, яка здобула перемоги в 1995 та 1997 роках з певною форою. Програми *Handtalk*, *Go4++* також досягли значних успіхів в іграх без фори проти гравців з високим рівнем майстерності. [6]

Загалом на той час було десять програм-лідерів, за ними шли тридцять програм з

середнім рівнем майстерності. Зазвичай останні програми були написані ентузіастами та дослідниками в той час, як перша десятка програм розповсюджувалися комерційно. Часто серед розробників зустрічаються й досить сильні аматорські гравці, а інколи навіть професіонали.

Найважливішими та найсильнішими в історії розвитку розробки програм для гри в «ГО» стали програми *GNU Go*, *CrazyStone*, *MoGo*, *KCC Igo*(також відома як *SilverStar* або *Ginsei Igo*) *Zen* та найкраща програма на 2020 рік – *AlphaGo*. [7]

До розробки *GNU Go*, програми Go завжди поширювалися лише як двійкові файли,а алгоритм засекречувався. В результаті, кожна нова розробка програми для «ГО» фактично починалася з нуля, що могло стати перешкодою для швидшого розвитку ефективності програм для цієї гри. *GNU Go* – це безкоштовна програма, яка підходить для багатьох платформ. Алгоритми, які використовуються в програмі та вихідний код у вільному доступі та задокументовані. Розробники сподіваються, що такий підхід сприятиме покращенню програми та розвитку комп'ютерного «ГО» в цілому. [8]

Crazy stone програма розроблена Ремі Кулом, яка використовує метод Монте Карло для пошуку оптимального рішення та шаблонне навчання. У 2007 році *Crazy Stone* зайняла друге місце після *MoGo*, розробники якої стверджують, що надихалися програмою Ремі Кула. У липні 2007 року *Crazy Stone* обіграла *KCC Igo* на 11 очок. Варто зауважити, що *KCC Igo* вважається найсильнішою комерційною програмою Go. Упродовж всіх років свого існування *Crazy stone* розвивалася, та у 2008 році навіть претендувала на титул найсильнішої програми гри «ГО» у світі. [7]

AlphaGo – комп'ютерна програма, яка поєднує дерево розширеного пошуку з глибокими нейронними мережами. Ця програма стала справжнім проривом в цій сфері. Програма містить дві нейроні мережі. Одна(мережа політики) з них обирає наступний хід, інша(мережа цінності) – передбачає переможця гри. Спершу *AlphaGo* брала участь у численних аматорських іграх. Таким чином розробники допомогти програмі розвинути розуміння розумної гри людини. Далі програма грала проти різних версій себе тисячі разів, навчаючись на власних помилках. З часом *AlphaGo* вдосконалювалась та ставала все сильнішою та кращою у навчанні та прийнятті рішень. Тривалий час вона перемагала чемпіонів світу Go на різних та на 2020 рік вважається найкращою

програмою для гри «ГО». [9]

1.4 Проблеми, які виникають при створенні програми

Існує чимало проблем, з якими стикається кожен розробник, коли починає створювати програму для цієї гри. Одна з основних – складність оцінки ефективності програм, бо складно порівнювати людину з машиною, адже сильні та слабкі сторони у них відрізняються. Один неправильний хід може зруйнувати всю гру [6] – це недолік, якій властивий обом сторонам.

Складність «ГО» полягає у надзвичайно великій кількості варіантів для ходу, пошуку грального простору, та велика тривалість гри. В зв'язку з цим програми для цієї гри безумовно швидші та показують кращі результати на малих дошках. Одна з рис, яка відрізняє дану гру з поміж інших – оцінка позиції, що є досить ускладненою в порівняні з іншими іграми. Таким чином гарна оцінка поля від продуктивності численних допоміжних пошуків, кожен з яких відповідає за підбір тактики.

Ще одною важливою проблемою є проблема розміру та структури простору. Поле розміром 19 на 19 набагато більше в порівнянні з іншими іграми. Кількість окремих позицій – $3^19 * 19$, при цьому трохи більше відсотка від цих позицій - дозволені. Такі великі числа приводять в приклад розміру видимого всесвіту [6]. Велика кількість заборонених позицій зумовлена правилом Ко. Для того, щоб можна було виявити та запобігти ходу, який утворює заборонену позицію, програма має розпоряджатися великою кількістю інформації про переходи. Ситуація на дошці може бути надзвичайно хаотичною, тому єдиний метод оцінки залишається повний вичерпний аналіз.

Також розрізняють чимало супробпроблем даної гри, для яких розроблені спеціалізовані методи, які мають високий рівень точності. Одна з них субпроблема *Life and Death*. Вона полягає в аналізі, чи може слабка група захиститися від захоплення. Більшість програм містить деякі знання про цю проблему та шляхи її вирішення. Зазвичай така група рятується створюючи два ока.

Іншою важливою субпроблемою є безпека каменів та території. Її визначення схоже на розв'язок субпроблеми життя чи смерті. Одна з основних відмінностей полягає в тому, що для перевірки безпеки на великому полі не можна використовувати прямий

пошук, оскільки простір надто великий. Інша відмінність – трактування співіснування *seki*. В той час як камені у безпеці, якщо їх не може захопити супротивник, територія в безпеці лише якщо можна довести, що ні один з каменів супротивника не може вижити на оточеній території.

1.5 Огляд методів, що використовуються в комп'ютерному «ГО»

Існує два основних методи збору інформації для програми для «ГО». Перший – це шаблони і цей спосіб ще можна назвати прямим, а другий – структуроване представлення з використанням ієрархій компонентів, що використовується для опису гри більш високо рівня. Після створення представлень першим або другим способом, відбувається збір та обробка знань, які потім використовуються для здійснення вдалих ходів. Розглянемо ці процеси детальніше.

1.5.1 Шаблони та порівняння з шаблоном

Паттерни або шаблони простий та сильний метод для кодування знань для комп'ютерного «ГО». Шаблони являють собою опис певних позицій відносно каменів, можливих у ході грі. Тому вони не прив'язані до розташування каменів відносно дошки. Багато ходів можна описати через паттерни; їх можна використовувати на всіх етапах, від початку та до кінця. Майже всі програми «ГО» мають базу даних, яка складається з шаблонів. Більшість шаблонів пишуться від руки, але також були створенні такі методи, які імпортують шаблони з професійних ігор. *Pattern-learning system* або система шаблонного навчання використовується найчастіше. Дуже багато різних позицій можуть бути зведені до одного шаблону в різних орієнтаціях та в 2 кольорових комбінаціях.

Але такий метод використовує чимало комп'ютерних ресурсів, адже потрібно порівняти кожен шаблон з ситуацією на столі в кожній орієнтації та у різних кольорових комбінаціях. Методи фільтрація на основі хеш-таблиць або дерев значно зменшили набір потенційних шаблонів, які треба порівнювати з ситуацією на дошці.

Використання паттернів в програмі для цієї гри зазвичай складається з трьох частин: *pattern map*, *pattern context*, *pattern information*. [6]

Шаблонне співставлення показує який пункт описується шаблоном та який статус пункту(зайнятий білим каменем, зайнятий чорним каменем чи пустий) дозволений у цьому шаблоні. Це корисно з урахуванням крайових та центральних патернів, адже наявність кутів або боків дошки впливають на достовірність багатьох шаблонів.

Контекст шаблону визначає додаткові нелокальні обмеження для того щоб ситуація на всій дошці відповідала шаблону. Серед найвпливовіших обмежень – підрахунок балів свободи та загальну безпеку каменів на краю шаблону.

Інформація про шаблон містить знання, які можуть бути застосовані в разі відповідності шаблону, такі як локально хороші і погані ходи.

Ефективний час роботи алгоритму зіставлення шаблонів залежить від великої кількості факторів. Після ходу більшість попередніх співпадінь з шаблонами та неспівпадінь залишаються дійсними. Щоб використовувати цей факт, для кожного матчу може бути збережено безліч залежностей. Після кожного ходу можуть бути повторно використані збіги, таким чином безліч залежностей залишаються незмінними, тому така оптимізація процесу дуже ефективна в грі.

Багато програм використовують інформацію, про шаблони для створення ходу, вони містять код, що надає перевагу ходам, які запропоновані паттерном. Зазвичай за вибір та оцінку ходів відповідає інша частина програми. Наприклад, шаблонний хід, що збільшує пункти свободи для групи каменів, може бути дуже корисним в ситуації, коли цій групі загрожує небезпека, але у випадку, коли група в безпеці це не має жодного значення. [6] Інші типи шаблонів використовуються для побудови надструктур таких як з’єднання та роз’єднання.

1.5.2 Дерева пошуку

Пошук використовується на різних рівнях та багатьма незалежними шляхами. Основними вважаються три види пошуку в деревовидних структурах: односпрямований пошук, багатоцільовий пошук та пошук по всій дошці. Повний пошук складний для реалізації. Таким чином пошукові системи спеціалізуються на досягненні більш обмеженого результату. Головна перевага в тому, що розглядається лише частина дошки для досягнення так би мовити однієї локальної цілі, що в багато разів швидша, ефективні-

ша ніж оцінка всієї дошки. Один з варіантів використання цілеспрямованого пошуку є представлення цікавих локальних ходів, в якості вкладу в вибірковий процес прийняття рішення про реалізацію ходу.

Односпрямований пошук використовує стандартні методи використання методик пошукових дерев для визначення тактичного стану блоків, груп та територій. Більшість цілеспрямованих пошуків виконуються по разу для кожного гравця повертаючи статус компонентів. Знання тактичного статусу компонентів покращує представлення дошки та є попередньою умовою для створення значимою функції очок. Найпростішим прикладом тактичного пошуку є *драбини(ladders)* – глибокозахоплюючі послідовності, де всі ходи атакуючого є загрозою захоплення та всі ходи захисників це примусові відповіді для уникнення моментального захоплення. Дуже часто це все лише один можливий хід. Драбини можуть проходити по всій дошці та загинатися.

Одна з важливих відмінностей від стандартного пошуку по дереву – програми повинні знаходити всі ходи, які досягають деякої тактичною цілі. Зазвичай в звичайному пошуку по дереву процес може бути зупинений після того, як один успішний хід знайдено. Пошук всіх ходів потрібен для того, щоб знайти багатоцільові ходи, а також оптимізувати вибір між всіма тактично хорошими ходами, використовуючи вторинні цілі.

Багатоцільовий пошук намагається досягти комбінації набору основних цілей. Ці комбінації можуть бути цілями високого рівня – захоплення хоча б одного блоку чи збереження всіх компонентів кордону незахопленими. Оскільки пошук кожної можливої комбінації цілей, що взаємодіють між собою неможливий, то для вибору найкращих комбінацій необхідно використовувати *евристику*, тобто шлях емпіричних досліджень. В більшості поточних програм реалізація багатоцільових пошукових систем погано розвинена.

Існує велика різноманітність підходів до проблеми глобального ходу в «ГО». На практиці використовується багато оригінальних комбінацій методів пошуку та методів, які базуються на знаннях програми. В зв’язку зі складністю оцінки та високим коефіцієнтом розгалуження системи «ГО», повний пошук має бути досить перебірливим та неглибоким. В деяких програмах пошук по всій дошці в основному використовує-

ться в ролі оцінки для так званих *тихих* позицій, або груп у яких немає переможних тактичних ходів.

1.5.3 Метод Монте Карло та UCT

Метод *Монте Карло* базується на випадкових іграх та легко оцінює кінцеві позиції після в кінці гри. Програма, що реалізовує цей алгоритм шукає ходи, які мають найбільшу долю виграшів, опираючись при цьому на дуже велику кількість партій. Цей метод непогано реалізовує ігровий процес так оцінка випадкових ходів не наближує програму до відтворення найкращих ходів. Саме тому потрібен глибокий та детальний пошук.

UCT метод (*Upper Confidence bounds applied to Trees*) це розширення реалізації пошуку методом *Монте Карло*, де для кожної гри перші ходи вибираються за допомогою дерева пошуку, що виросло в пам'яті і як тільки кінцевий вузол знайдений, новий хід додається до дерева, а частина гри що залишилася, продовжується випадковим чином.

UTC метод вирішує проблеми вибору ходу таким чином, що в дереві важливі ходи шукаються частіше ніж потенційно погані ходи. *UTC* вибирає найкращі ходи розглядаючи кількість відсотків виграшів при використанні цього ходу і чим більше це значення, тим більша ймовірність, що його буде обрано. А якщо цей хід було обрано, але партія програна це понижує, то це понижує відсоток виграшів, таким чином інші ходи будуть використовуватися частіше.

Починаючи від кореню *UTC* шукає шлях через дерево вираховуючи цінність для кожної потенційної позиції спираючись на долю виграшів та те, скільки разів позиція була використана. Якщо діти в вузлі не були *відвідані*, то один з них вибирається випадковим чином.

1.6 Висновки за першим розділом

Гра «ГО» вирізняється серед інших програм своєю складність для реалізації за допомогою алгоритму. Для цього є ряд різних причин. Перш за все ця гра оперує великою кількістю операцій та має надзвичайно велику кількість можливих комбінації та варіантів ходів, які практично неможливо перебрати. Також досить серйозною проблемою

є оцінка позиції, яку теж тяжко реалізувати алгоритмічно. Проте за останні роки людство суттєво просунулося уперед. Упродовж останніх років розроблялися для розробки програм для «ГО» почали використовуватися метод *Монте Карло* та *UCT*, які суттєво підвищили рівень майстерності комп'ютерного «ГО». Але справжнім проривом стала програма *AlphaGo*, яка поєднала метод *Монте Карло* з глибинним навчанням за допомогою багаторівневих нейронних мереж. Уявлення про розвиток та здобутки в сфері комп'ютерного «ГО» дають можливість адекватно оцінити шляхи розв'язання поставленої задачі.

2 РОЗДІЛ: Розробка алгоритму пошуку оптимального розв'язку в грі на ітераційного методу Марковських процесів прийняття рішень

2.1 Основні поняття Марковських процесів з доходами

Спершу визначимо основні поняття, які будуть використовуватися для характеристики методів. *Марковськім процесом* називають математичну модель для вивчення складних систем. Вважається, що система знаходиться в якомусь стані, якщо вона повністю описується значеннями змінних, що задають цей стан. Тому при змінні значень змінних вважаємо, що система переходить в інший стан. Марковські процеси за часом поділяються на два види: процеси з *дискретним* часом та процеси з *неперервним* часом. Перший вид характеризує процеси в яких ми зосереджуємо увагу лише не переходах систему між станами та нумеруємо їх по часу. Процеси ж з неперервним часом - процеси в яких розглядається випадковий час між переходами. В подальшому розглядатимуться процеси з дискретним часом.

Розглянемо ймовірнісну природу переходів для процесів з дискретним часом. Можна ввести набір умовних ймовірностей p_{ij} того, що система, що знаходиться у стані i після чергового переходу опиниться в стані j. Так як система обов'язково має попасти в деякий стан після переходу, то

$$\sum_{j=1}^{N} p_{ij} = 1 \tag{1}$$

Марковські процеси з доходами – це процес, який приносить дохід при переході зі i стану в j. Такий дохід позначається r_{ij}. Множина доходів утворює матрицю доходів R. Нехай є можливість вибору поведінки та послідовності переходів між станами. Утворені можливості поведінки шляхом комбінування різних виборів назвемо стратегіями. Кожна стратегія має пов'язані з нею розподіли ймовірностей та доходів для виходу зі стану. Приналежність змінної до певної стратегії будемо позначати індексом зверху, який відповідатиме номеру стратегії. Кількість стратегій в кожному стані має бути скінчена, але може бути різною для різних станів.

Величина q_i^k визначається як очікуваний дохід за перехід при виході зі стану i та при виборі стратегії k. Вона обраховуватиметься за формулою

$$q_i^k = \sum_{j=1}^{N} p_{ij}^k r_{ij}^k \tag{2}$$

Для максимізації отриманого прибутку за n кроків в залежності від поточного стану обирається стратегія, якої треба притримуватися. Номер стратегії, яка обирається в стані i та яка використовується на n-ому кроці позначаємо $d_i(n)$. Оптимальною називається поведінка, яка максимізує повний очікуваний дохід для всіх i та n.

2.2 Ітераційний метод

В теорії Марковських процесів прийняття рішення існує ітераційний метод розв'язку задач для процесів послідовних рішень. Він застосовується у випадку, коли процес здійснює переходи упродовж довгого проміжку часу та нас цікавить дохід цього процесу. Цей метод дозволяє знайти оптимальний розв'язок за невелику кількість ітерацій, кожна з яких складається з двох етапів: визначення ваги та покращення розв'язку.

Визначення ваг визначає ваги як функцію розв'язку, в той час як покращення розв'язку визначає розв'язок як функцію ваг. В першому блоці циклу визначаються величини g та V_i, що відповідають вибору p_{ij} та q_i, а в другому – обчислюються значення p_{ij} та q_i, які збільшують прибуток для даного набору V_i. Ітераційний цикл можна починати з будь-якого блоку. Якщо початковим блоком обрати визначення ваг, то потрібно підібрати початковий розв'язок. Якщо в якості початкового блоку виступає покращення розв'язку, то потрібно задати набір початкових ваг. Оптимальний розв'язок вважається знайденим, коли співпадуть розв'язки двох послідовних ітерацій.

Серед властивостей цього методу можна визначити, що пошук оптимального розв'язку в ньому зводиться до обчислення системи лінійних рівнянь; кожний наступний розв'язок більший за прибутком за попередні та ітераційний цикл закінчується при отриманні розв'язку, що забезпечує найбільший допустимий розв'язок. [10]

Розглянемо детальніше процедури визначення ваг та покращення розв'язку.

2.2.1 Визначення ваг

При визначенні ваг розглядається система при фіксованому розв'язку з заданим Марковським процесом з доходами. Розглядається, що в процесі відбулося n переходів та $V_i(n)$ визначається як повний очікуваний дохід, який отримується після виконання цих переходів, за умови, що i - початковий стан. Ці величини повинні задовольняти рекурентним співвідношенням

$$V_i(n) = q_i + \sum_{j=1}^{N} p_{ij}^k V_j(n-1)) \qquad i = \overline{1,N} \qquad n = 0,1,2,... \tag{3}$$

Для ергодичних Марковських процесів $V_i(n)$ має асимптотичний вигляд

$$V_i(n) = ng + V_i \qquad i = \overline{1,N} g = \sum_{i=1}^{N} \pi_i q_i \tag{4}$$

де π_i – це гранична ймовірність відповідного стану. Для обчислення граничних ймовірностей станів будь-якого процесу використовуються рівняння

$$\pi(n+1) = \pi(n)P \sum_{i=1}^{N} \pi_i = 1 \tag{5}$$

де $\pi(n+1)$ – вектор-рядок, який складається з ймовірностей, того, що система буде знаходитися в кожному стані через $n+1$ переходів.

Розглянемо поведінку системи при великій кількості переходів. Значення величин $V_i(n)$, отримані з (4), можна підставити в (3). Отримуємо

$$ng + V_i(n) = q_i + \sum_{j=1}^{N} p_{ij}((n-1)g + V_j) \qquad i = \overline{1,N} \tag{6}$$

$$ng + V_i = q_i + (n-1)g \sum_{j=1}^{N} p_{ij} + \sum_{j=1}^{N} p_{ij} V_j \qquad i = \overline{1,N} \tag{7}$$

З 1 отримуємо, що ці рівняння набувають вигляду

$$g + V_i = q_i + \sum_{j=1}^{N} p_{ij} V_j \qquad i = \overline{1,N} \tag{8}$$

Отримуємо систему з N лінійних рівнянь, що пов'язують величини V_i та g з матрицями перехідних ймовірностей та доходів процесу. Потрібно визначити N величин V_i та величину g, а отже $N+1$ невідому. V_i – відносні ваги розв'язку. Для полегшення розв'язку системи рівнянь одне зі значень розглядається рівним нулю. Отримані таким чином ваги V_i відрізнятимуться від відповідних величин, визначених рівністю (4), на константу.

Відносним вагам можна надати фізичну інтерпретацію. Якщо розглянемо перш два стани системи, то для будь-якого великого n отримуємо

$$V_1(n) = nq + V_1 V_2(n) = nq + V_2 \tag{9}$$

Різниця $V_1(n) - V_2(n) = V_1 - V_2$ показує наскільки вигідніше починати роботу системи з першого стану, а не другого стану. Саме ця властивість охарактеризовує ці величини вагами. Відносні ваги необхідні для покращення розв'язку.

2.2.2 Покращення розв'язку

Якщо притримуватися оптимального плану до $n+1$ кроку, то кращу стратегію в i-ому стані на $n+1$ кроці можна знайти максимізувавши відносно всіх стратегій в i-ому стані вираз

$$q_i^k + \sum_{j=1}^{N} p_{ij}^k V_j(n) \tag{10}$$

Для великих n можна використовувати рівність (4), щоб отримати критерій максимізації у вигляді виразу

$$q_i^k + \sum_{j=1}^{N} p_{ij}^k (ng + V_j) \tag{11}$$

який повинен бути максимізованим в i-ому стані. Зважаючи, що (1) то доданки та ng та довільна стала відносно ваги V_i утворюють складову, яка не залежить від k. Таким чином, щоб прийняти рішення в стані i достатньо максимізувати вираз

$$q_i^k + \sum_{j=1}^{N} p_{ij}^k V_j \tag{12}$$

відносно всіх стратегій в i-ому стані. До того ж в цьому виразі можна використати відносні ваги з рівняння (8).

Процедура покращення розв'язку полягає в тому, що для кожного стану i використовуючи відносні ваги, визначені для старого розв'язку, знайти стратегію k, яка максимізує критерій (12).

2.2.3 Розробка програми для реалізації методу

Тепер адаптуємо введенні положення та означення до даної задачі. Множина станів, яка характеризує гру описує множину ходів. Розглядається матриця розміром п'ять на п'ять, а отже дошка складається з двадцяти п'яти пунктів, то всього станів буде двадцять п'ять. Множина ймовірностей переходів залежить від умов гри та розміру поля, тому для порожньої дошки буде розміром двадцять п'ять елементів на двадцять п'ять елементів та складатися з однакових значень – $\frac{1}{25}$. Після закінчення одного ходу на дошці буде два камені, камінь гравця та камінь, який поклала програма. Елементи матриці, що описують пункти, на яких стоять камені набудуть нульового значення, адже ставити камені на ці пункти більше немає можливості. Ймовірності переходів на інші пункти збільшаться та набудуть значення – $\frac{1}{23}$. В програмі реалізовано метод, який після кожного ходу генерує нову матрицю ймовірностей переходів у вільні пункти.

В якості стратегій розглядатиметься два варіанти поведінки: ставити камені біля каменів супротивника з метою захоплення його фішок чи ставити камені не поряд з каменями супротивника, щоб захопити по-більше вільної території.

Процес також описується множинами доходів. Множина доходів у грі - це множина значень очок, які отримає гравець при переходів в стан з заданим положенням каменю. Тобто це кількість очок, який гравець отримає при виборі певного розташування каменю. Для підрахунку доходів після кожного ходу викликається функція, яка аналізуючи ситуацію на дошці генерує матрицю доходів. А на основі матриці доходів обраховуються загальні доходи для всіх потенційних переходів з поточного стану.

Для обчислення ваг та прибутку системи викликається окрема функція. Вона обраховує коефіцієнти отриманої системи рівнянь та записує рівняння таким чином, щоб усі змінні були по одну сторону рівняння, а по іншу - значення передаються в окрему функцію, яка за допомогою методу Гауса знаходить значення загальний дохід системи та очікувані доходи(ваги). Для зручності значення ”крайнього”за індексом доходу береться нульове. На базі отриманих розв’язків рахується значення критерію для кожного стану, за яким буде в подальшому вирішуватися, який краще робити хід в даному стані.

В додатку А наведений текст коду для реалізації гри. Для створення застосунку використовувалася мова програмування *Python* та її бібліотека *Kivy*. *Kivy* – це кросплатформена бібліотека з відкритим кодом, яка використовується для швидкої розробки застосунків, яка підтримує створення інноваційних користувацьких інтерфейсів, зокрема мультитач.

В ході роботи також використалася бібліотека *random* для генерації випадкових ходів програми. Таким чином була змога перевірити роботу програми. В створені програмі реалізований інтерфейс та частково функціонал гри. Для створення інтерфейсу спершу були створені графічні складові гри: вільний пункт на дошці, камені білого та чорного кольору. За правилами гри, в якості ходу гравець кладе камінь свого кольору на точку перетину ліній, зображених на дошці.

В програмі дошка це віджет у вигляді таблиці. Складовими цієї таблиці є кнопки. Ці кнопки уособлюють собою пункти, на які гравець може поставити камінь, тому в якості фону кнопки використовується графічне зображення перетину ліній. В програмі задається розмір поля та залежно від цього значення встановлюється розмір вікна. Розмір пункту сталий.

Для того, що гравець походив. йому треба натиснути на відповідний пункт, та у відповідь на цій кнопці з’явиться камінь чорного кольору. За правилами, гравець, який ходить першим ходить чорними каменями. Для кнопок прописані реакції на дві події: натискання кнопки та відпускання. У відповідь на натискання кнопки програма замінює картинку фону кнопки на картинку з каменем, а при відтисканні запускається функція, яка генерує випадковим чином індекс кнопки, на яку програма поставить свій

камінь. В ході генерації перевіряється чи обраний пункт не зайнятий. Якщо на згенерованому пункті стоїть камінь, то процес продовжується. Для цього створений масив так званих статусів пунктів. Статус з відповідними індексами містить інформацію про пункт. Може бути три значення: ”,’w’,’b’. Перше значення означає, що пункт пустий, друге значення означає, що в пункті стоїть білий камінь та третій – що пункт зайнятий чорним каменем. Також реалізовано перевірку згенерованого пункту на те чи хід на цей пункт буде самогубством, чи ні. Ця функція може повертати два значення 0 або 1, де 0 – на пункт можна ставати, 1 – цей хід буде самогубством.

Після натискання та відпускання кнопки вона стає неактивною, тому гравець не може на неї натиснути.

Висновки

«ГО» – це гра, процес якої складно реалізувати для комп'ютера і на це існує немало причин. Проте сучасні методи, які використовують розробники часто дають хороші результати та їхні програми здатні навіть змагатися з гравцями високого рівня майстерності. Проте лише одна програма на сьогоднішній день демонструє результати, які дійсно вражають. *AlphaGo* – це програма, яка трохи "відійшла"від традиційних методів та використала нову методу в поєднані з методом *Монте Карло*, що дало поштовх для стрімкого розвитку комп'ютерного «ГО». Розробники цієї програми не лише розпочали новий період в історії цієї сфери, а й збільшили кількість можливостей для подальших досліджень. Тобто тепер, для наступних розробників більше матеріалу та інформації, а отже більше ресурсів в майбутньому для переведення комп'ютерного «ГО» на наступний рівень розвитку.

Існування певних проблем при реалізації гри для комп'ютера спонукає до подальшого дослідження цієї проблеми та пошуку її оптимальних розв'язків. Використання Марковських процесів для цієї цілі дає можливість розглянути проблему під новим кутом з використанням нових методик. Реалізований метод – це спроба оптимізації пошуку оптимально розв'язку способом, який для такої задачі раніше не використовувався, що дає підстави для розширення загального переліку методів, які можуть бути ефективними для такої задачі.

Список літератури

[1] Г. Оуен Теория игр/ Г. Оуен: [Пер. с англ. под ред. А. А. Корбута со вступ. статьей Н. Н. Воробьева] - М.: Мир, 1971. - 230 с.

[2] Дж. фон Нейман, О. Моргенштерн Теория игр и экономическое поведение/ ж. фон Нейман, О. Моргенштерн: [Перев. с англ. под ред. и с доб. Н. Н. Воробьева] — Главная редакция физико-математической литературы, изд-ва ”Наука”1970. — 708 с.

[3] Carson K. Leung, Felix Kanke, Alfredo Cuzzocrea Data analytics on the board game Go for the discovery of interesting sequences of moves in joseki/ Carson K. Leung, Felix Kanke, Alfredo Cuzzocrea: - Procedia Computer Science, 2018. - 831-840 с. ISSN 1877-0509.

[4] Правила Го - это очень просто![online] Available at: http://ufgo.org

[5] International Rules[online] Available at: http://home.snafu.de/jasiek/int.html [Accessed 24 Aug. 1998]

[6] Martin Müller (2002). Computer Go [online] p. 20. Available at: https://www.sciencedirect.com/science/article/pii/S0004370201001217

[7] Go-Playing Programs [online] Available at: https://senseis.xmp.net [Accessed 13 Nov. 2000].

[8] Daniel Bump (2009) GNU Go Documantation [online] Available at: https://www.gnu.org/software/gnugo/gnugo_1.html [Accessed 19 Feb. 2009].

[9] AlphaGo [online] Available at: https://deepmind.com/research/case-studies/alphago-the-story-so-far

[10] Р. А. Ховард Динамическое программирование и Марковские процессы/ Р. А. Ховард: [Пер. с англ. В.В.Рыкова под ред. Н.П.Бусленко] - Издательство ”Советское радио Москва, 1964. - 195 с.

Додаток А

(обов'язковий)

Текст програми "GOApp"

```
from kivy.app import App
from kivy.graphics import Color, Rectangle
from kivy.uix.gridlayout import GridLayout
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.button import Button
import random
import copy
from decimal import Decimal

from gauss import gauss

size_of_game = 5
width = size_of_game*70
height = width

from kivy.config import Config
Config.set('graphics','height',height)
Config.set('graphics','width',width)
Config.set('graphics','resizable',0)

class RootWidget(FloatLayout):

    def __init__(self, **kwargs):
        super(RootWidget, self).__init__(**kwargs)

        with self.canvas.before:
```

```
            Color(0.87, 0.72, 0.53, 1); # green; colors range fro
            self.rect = Rectangle(size=(width,height))

        grid = GridLayout(cols=size_of_game, rows=size_of_game,

        Buttons = [[0 for i in range(size_of_game)] for j in ran
        status = [['' for i in range(size_of_game)] for j in ran

        self.add_widget(self.grid(grid,Buttons,status))

    def grid(self,grid,Buttons,status):

        for i in range(size_of_game):
            for j in range(size_of_game):
                grid.add_widget(self.players_stone(Buttons,status

        for i in range(size_of_game):
            for j in range(size_of_game):

                def move(instance):
                    if self.status_of_cells(status)==0:
                        pass
                    else:
                        btn = self.bots_stone(Buttons,status)
                        btn.background_normal = 'w.png'
                        btn.background_down = 'w.png'
                        btn.disable = True

                Buttons[i][j].bind(on_release=move)
```

```
        return grid

    def status_of_cells(self,status):
        count_1 = 0
        for i in range(size_of_game):
          for j in range(size_of_game):
             if status[i][j]='':
                 count_1=count_1+1
        return count_1

    def bots_stone(self,Buttons,status):

        def generate_p(status):
            if self.status_of_cells(status) == 0:
                raise Exception("No turns left")
            pij_n = Decimal(1/self.status_of_cells(status))

            p=[]
            for i,row in enumerate(status):
                for j,col in enumerate(row):
                    local_status = copy.deepcopy(status)
                    local_status[i][j] = 'b'
                    p.append([pij_n if local_status[i][j]='' el

            return p

        def calculate_go_income(local_status):
            return 0

        def generate_current_weights(status):
```

```
    r=[]
    for i,row in enumerate(status):
        for j,col in enumerate(row):
            local_status = copy.deepcopy(status)
            local_status[i][j] = 'b'
            r.append([calculate_go_income(local_status)
    return r

def generate_outcomes_per_state(p,r):
    q = [0 for j in range(len(p))]
    for i,row in enumerate(p):
        for j,col in enumerate(row):
            q[j] += p[i][j]*r[i][j]
    return q

def calculate_g_vi(p,q):
    tmp = []
    for i,row in enumerate(p):
        tmp.append([1])
        tmp[i].extend([float(-p[i][j]) if j!=i else floa
    return gauss(tmp,list(map(float,q)))

def calculate_criterion(p,q,v):
    criterion = []
    for i,row in enumerate(p):
        criterion.append(q[i])
        for j,col in enumerate(row):
            criterion[-1] += col*Decimal(v[j])

    return criterion
```

```
        p = generate_p(status)
        r = generate_current_weights(status)
        q = generate_outcomes_per_state(p,r)

        g, v = calculate_g_vi(p,q)
        v = list(v)
        v.append(0)
        criterion = calculate_criterion(p, q, v)

        index1 = random.randrange(0, size_of_game, 1)
        index2 = random.randrange(0, size_of_game, 1)

        while status[index1][index2]!='':
          index1 = random.randrange(0, size_of_game, 1)
          index2 = random.randrange(0, size_of_game, 1)

        status[index1][index2] = 'w'

        color = 'b'
        print(self.check(status,index1,index2,color))
        return Buttons[index1][index2]

    def check(self,status,index1,index2,color):
        check=0

        if index1==0:
            if index2==0:
                if status[index1+1][index2]==color and status[in
                        check=1
```

```
        elif index2==size_of_game-1:
            if status[index1+1][index2]==color and status[in
                check=1
        elif index2<size_of_game-1 and index2>0:
            if status[index1][index2-1]==color and status[in
                if status[index1+1][index2]==color:
                    check=1

    elif index1<size_of_game-1 and index1>0:
        if index2==0:
          if status[index1-1][index2]==color and status[inde
              if status[index1][index2+1]==color:
                  check=1
        elif index2<size_of_game-1 and index2>0:
          if status[index1-1][index2]==color and status[inde
              if status[index1][index2-1]==color and status[
                  check=1
        elif index2==size_of_game-1:
          if status[index1-1][index2]==color and status[inde
              if status[index1][index2-1]==color:
                  check=1

    elif index1==size_of_game-1:
        if index2==0:
            if status[index1-1][index2]==color and status[in
                    check=1
        elif index2==size_of_game-1:
            if status[index1-1][index2]==color and status[in
                check=1
        elif index2<size_of_game-1 and index2>0:
```

```
                if status[index1][index2-1]==color and status[in
                    if status[index1-1][index2]==color:
                        check=1
        return check

    def players_stone(self,Buttons,status,i,j):
        Buttons[i][j] = Button(background_color=(1,1,1,1),
                               background_normal='button.png',
                               background_down='button.png',
                               border=(0,0,0,0),
                               size=(width/size_of_game, height/

        def callback(event):
            if status[i][j] == '':
             Buttons[i][j].background_normal = 'b.png'
             Buttons[i][j].background_down = 'b.png'
             Buttons[i][j].disable = True
             status[i][j] = 'b'

        Buttons[i][j].bind(on_press=callback)

        return Buttons[i][j]

class GoApp(App):

    def build(self):
        return RootWidget()

if __name__ == '__main__':
    GoApp().run()
```