МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Львівський національний університет імені Івана Франка

Факультет електроніки та комп'ютерних технологій

Кафедра оптоелектроніки та інформаційних технологій

Допустити до захисту

Завідувач кафедри
_____ проф. Кушнір О. С.
«___» _____2023 р.

Кваліфікаційна робота

Бакалавр
(освітній ступінь)

Еволюційна гра "Яструб-Голуб"

Виконав:
студент IV курсу групи ФЕП– 41с
спеціальності 121 – Інженерія
програмного забезпечення

_____ Я.В. Сворень

Науковий керівник:
_____ доц. І.М. Катеринчук
«____» _____2023 р.
Рецензент:
_____ доц. С.Р. Вельгош

Львів 2023

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА

Факультет__Електроніки та комп'ютерних технологій_____
Кафедра____Оптоелектроніки та інформаційних технологій_____
Освітній ступінь ____бакалавр_____
Галузь знань_____12 "Інформаційні технології"_____
(шифр і назва)
Спеціальність _____121 "Інженерія програмного забезпечення_____
(шифр і назва)

«ЗАТВЕРДЖУЮ»
Завідувач кафедри_____
_____
"____" _____20___року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ (БАКАЛАВРСЬКУ) РОБОТУ СТУДЕНТУ

Свореню Ярославу Вікторовичу
(прізвище, ім'я, по батькові)

1.Тема роботи:
Еволюційна гра «Яструб-Голуб»
керівник роботи ___Катеринчук Іван Миколайович к.ф.-м.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені Вченою радою факультету від "31" жовтня 2022 року № 30/22
2. Строк подання студентом роботи __04.06.2023 року_____
3. Вихідні дані до роботи:
1. Hans Peters Game Theory: A Multi-Leveled Approach. Springer-Verlag Berlin Heidelberg 2008. 365p. DOI: 978-3-540-69291-1
2. Tim Rees An Introduction to Evolutionary Game Theory URL: https://www.cs.ubc.ca/~kevinlb/teaching/cs532a%20-%202004-5/Class%20projects/Tim.pdf (дата звернення: 20.10.2022)
3. Harold W.Kuhn Lectures on the Theory of Games / HaroldW. Kuhn – Princeton and Oxford: Published by Princeton University Press, 2003. – 107p.
4. Erich Prisner Game Theory Through Examples / Erich Prisner – Published and Distributed by The Mathematical Association of America, 2014. – 287 p.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд літератури по теорії еволюційних ігор.
2. Розробка прототипу гри "Яструб - Голуб".
3. Побудова графічного інтерфейсу гри "Яструб - Голуб".

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_Рисунки, знімки екрану, таблиці_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання
1 листопада 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної (бакалаврської) роботи	Строк виконання етапів роботи	Примітка
1	Огляд літератури по теорії еволюційних ігор	01.11.2022 р. - 30.11.2022 р.	
2	Розробка прототипу гри “Яструб - Голуб”	01.12.2022 р. - 12.03.2023 р.	
3	Побудова графічного інтерфейсу гри “Яструб - Голуб”	13.03.2023 р. - 03.05.2023 р.	
4	Оформлення роботи	04.05.2023 р. - 02.06.2023 р.	
5	Попередній захист дипломної роботи	05.06.2023 р.	

Студент _____ Сворень Я.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Катеринчук І.М.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розглянуто теорію еволюційної гри “Яструб-Голуб” та як аналіз цієї гри допомагає зрозуміти ефективний ігровий дизайн суперницьких ігор за її прикладом. Побудовано програму для симуляції гри “Яструб-Голуб” з можливістю вирахування найкращої стратегії для кожної з можливих стратегій противника. Побудовано графічний інтерфейс до програми для симуляції гри “Яструб-Голуб”.

ABSTRACT

The theory of the evolutionary game "Hawk-Dove" is considered and how the analysis of this game helps to understand the effective game design of rival games based on its example. A program was built to simulate the game "Hawk-Dove" with the possibility of calculating the best strategy for each of the possible strategies of the enemy. A graphical interface to the program for simulating the game "Hawk-Dove" has been built.

ЗМІСТ

ВСТУП

Одним з самих відомих прикладів аналізу стратегії гри гравців вважається еволюційна гра “Яструб-Голуб”, яка була представлена у 1973 році Джоном Мейнардом Смітом та Джорджом Прайсом у роботі під назвою “Логіка тваринного конфлікту”. Саме у цій роботі був наведений ясно зрозумілий приклад моделі гри, в якій у кожного гравця була можливість вибрати яструба, який перемагає голубів та забирає їхні ресурси, та який вступає в бійку з іншими яструбами за ресурси, або вибрати голуба, який дружньою ділить порівну ресурси з іншими голубами, та мусить віддавати їх попутному яструбу.

Ідея цієї роботи закладалася в тому, щоб відповісти на одне єдине питання: яку стратегію краще всього вибрати гравцю цієї гри? Питання насправді непросте, оскільки гра триває довго, і персонажа свого можна міняти після кожної ітерації гри. Якщо обоє гравці гратимуть тільки за голубів, то будуть порівну ділити ресурси і переможця не буде. Якщо ж обоє гравців братимуть тільки яструбів, то гра перетвориться у банальне підкидання монетки, хто більше у боях за ресурси випадково переможе. У цій роботі буде дано відповідь на це запитання та продемонстровано на практиці чому знайдена нами стратегія є найкращою з усіх можливих стабільних стратегій для цієї гри.

1. Теоретичні відомості

1.1 Еволюційна теорія ігор

Еволюційна теорія ігор (ETI) перетворилася на галузь, яка поєднує принципи теорії ігор, еволюції та динамічних систем для інтерпретації взаємодії біологічних агентів. Практики цієї галузі використовували теорію для успішного пояснення біологічних явищ, але використовують ETI ще й для інтерпретації класичних ігор з іншої точки зору.

Існує кілька основних компонентів аналізу ETI різних можливих ігор [1]. По-перше, ігрові агенти та їхні стратегії повинні бути змодельовані з населенням гравців, з чого випливає що кожен мешканець населення дотримується одної незмінної стратегії, в той час як гравець гри визначає співвідношення населення свого, обираючи кількість населення кожної групи населення з незмінною стратегію. По-друге, дотримування кожної зі стратегій має приводити до певного результату, який може буде виміряний якоюсь певною одиницею ресурсів, яка допомагатиме порівнювати успішність конкретної стратегії. По-третє, процес розвитку популяції необхідно визначити, тобто що допускаються мутації всередині населення, які спотворюватимуть початкове співвідношення, задане гравцем.

Ці прості компоненти можна комбінувати для отримання комплексних розв’язувань проблем, як для прикладу обирання кращої стратегії гравцеві. В ідеалі, під динамічним процесом зміни стратегій гравців буде сходитися до деякого стабільного значення, відхилення від якого буде приводити до гірших результатів. Теоретики еволюційних ігор часто стверджують еволюційне розв’язування гри, як справжнє визначення раціональної стратегії гри [2].

Концепція моделювання груп гравців для визначення раціональної гри не є новою. Подібна ідея була запропонував Джоном Форбсом Нешом молодшим у своїй докторській дисертації у 1950 році на тему некооперативних ігор за введення терміну “Рівновага Неша”, яка стала міцним фундаментом еволюційних ігор. Проте справжнім народженням ETI можна вважати роботу Мейнарда Сміта «Еволюція та теорія ігор», опублікованою у Cambridge University Press у 1982 році.

Еволюційна теорія ігор – це зовсім інший підхід до класичного аналізу ігор. Замість прямого обчислення властивостей гри симулюються популяції гравців, які використовують різні стратегії, і використовується процес, схожий на природний відбір, щоб визначити, як еволюціонує популяція. Для представлення популяцій у багато-агентних іграх із різними стратегічними просторами потрібен різний ступінь складності. Під агентом мається на увазі жива людина, яка приймає участь у грі. Під гравцем мається на увазі персонаж, який може мінятися протягом гри.

Розглянемо гру, яка складається з n агентів. Стратегія кожного агента під номером i позначатимемо як S_i. Підхід ETI полягав би в моделюванні кожного агента групою гравців. Популяція для агента під номером i буде потім розділена на групи $E_{i1}, E_{i2}, \ldots, E_{ik}$, де k може бути різним для різних населеннь. Усі особи в групі E_{ij} дотримуються однакової (можливо змішаної) стратегії, прописаної у S_i. Наступним кроком буде випадкова гра між членами популяцій один проти одного. Субпопуляції, які показали найкращі показники, зростатимуть, а ті, які не будуть ефективними, скорочуватимуться. В ідеалі еволюція мала б сходитися до певного стабільного стану для кожної популяції, що представляло б найкращу відповідь (можливо, змішаної) стратегії для кожного агента. Під змішаною стратегією мається на увазі небірнарність стратегії, коли гравець приймає те саме рішення протягом усієї гри.

1.2 Поняття симетричної та асиметричної еволюційної гри

Особливим випадком є симетрична гра для двох гравців. У симетричній грі матриці виграшу та дії ідентичні для обох агентів [3]. Ці ігри можуть бути змодельовані окремою сукупністю індивідів, які грають один проти одного. Коли гра є асиметричною, для імітації кожного агента необхідно використовувати різні групи гравців.

Розглянемо на прикладі логіку симетричної гри. У симетричній грі обидва агенти мають скінченну кількість стратегій, які вони можуть присвоїти кожному зі своїх гравців. Візьмемо для прикладу гру у футбольного нападаючого та воротаря. Підсумувати правила гри можна класичними правилами пенальті, де якщо нападаючий заб'є більше голів, чим відіб'є воротар, то він перемагає і навпаки, якщо воротар відбив більше, то переміг він. Спростуємо рішення кожного з гравців до банального вибору: лівий чи правий напрямок? В такому разі ми можемо візуалізувати симетричну гру таким чином:

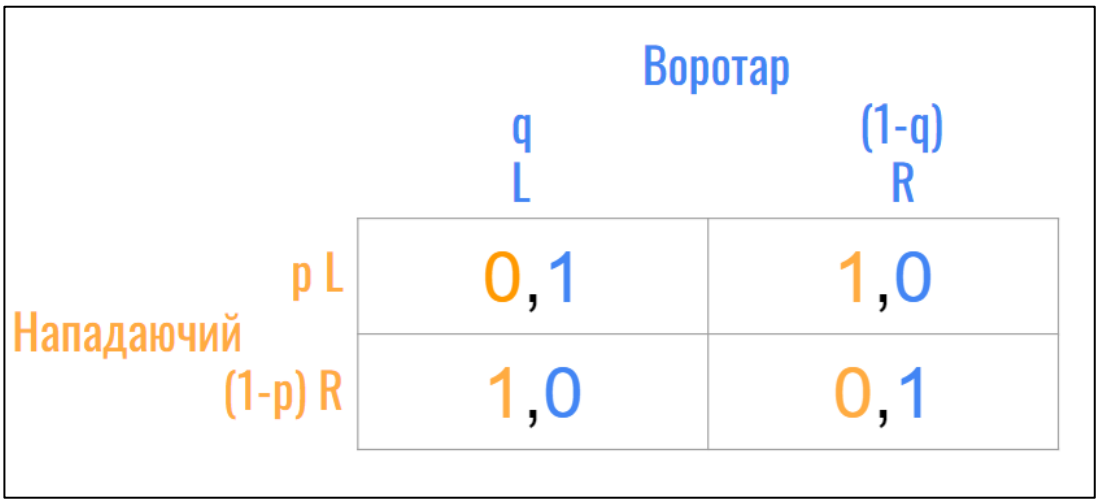


Рис 1.1 Біматриця нагород спрощеної гри у пенальті

Візуалізація еволюційних ігор у формі біматриці [4] вважається стандартним та зустрічається у безмежній кількості робіт на тему еволюційних ігор. На місці вказання лівого чи правого напрямку гравців зазвичай заміняються на ймовірність прийняття рішення, бо вказувати результат рішення не є суттєвим насправді. Конкретно у цьому випадку ми бачимо зліва нападаючого, якого ще називають рядковим, бо можливі результати після прийнятого ним рішення

йдуть по рядку, в той ж час як у воротаря вони йдуть по колонці. Першим у кожній комірці матриці йде результат для рядкового гравця, другим – колонкового гравця. Конкретно у нас бачимо, що воротар отримує 1 бал, відбивши м'яч, у верхньому лівому та нижньому правому результаті матриці, бо у цих випадках він кидається в той ж бік, в який нападаючий б'є м'яч, і у нашому гіпотетичному прикладі ми упускаємо можливість того, що воротар м'яч не зловить. З іншого боку, якщо воротар та нападаючий обрали різні напрямки, то воротар жодного балу не отримує, а нападаючий – так.

1.3 Поняття рівноваги Неша

Повертаючись до вступу згадуємо про Джона Форбса Неша молодшого та його дисертацію у 1950 році. Саме ця 28-сторінкова дисертація ввела поняття рівноваги Неша (в оригіналі Nash Equilibrium). Рівновагою Неша ми називаємо сукупність стратегій S_i усіх гравців, дотримуючись принципу того, що кожна стратегія S_i, яка належить агенту i є найкращою з можливих стабільних стратегій не тільки в якийсь конкретний момент, а передбачаючи дії суперників загалом найкращим підходом для конкретного агента [5]. Ця комбінацій стратегій зумовлює, що жоден з агентів не може збільшити виграш, змінивши вибір стратегії в односторонньому порядку, коли інші учасники не змінюють свого вибору.

Ми можемо підсумувати рівновагу Неша як комбінацію стратегій, де кожен агент опиняється у найкращому з можливих положень, бо по-перше його стратегія є найкращою можливою реакцією на стратегії інших, а по-друге його стратегія дає йому найкращу можливу стабільну долю ресурсів, за які йде боротьба.

Обчислення рівноваги Неша можна поділити на два випадки.
Перший випадок:

Рис 1.2 Біматриця нагород, що містить чисту рівновагу Неша

В цьому випадку знайти рівновагу Неша можна дуже просто. Уявляємо на хвилину, що ми є перший гравець. Знаючи, що другий гравець зараз обрав рішення ймовірності q, яка з наших опцій нам більше пасує? Дивимось на колонку q і підкреслюємо собі число 18. Далі уявляємо собі, що другий гравець натомість прийняв рішення з ймовірністю $(1-q)$, тоді підкреслюємо собі число 3. Далі повторюємо ті самі дії, уявивши себе другим гравцем, просто цього разу ми обираємо серед вигідних нам значень, дивлячись на рядок p та дивлячись на рядок $(1-p)$. Отримуємо результат нижче:

Рис 1.3 Біматриця з підкресленими бажаними нагородами гравців

В результаті бачимо, що у нижній правій комірці матриці обидва значення підкреслені. З цього випливає, що ми маємо рівновагу Неша у цій комірці, яку можна інтерпретувати двома бінарними стратегіями, а саме перший гравець буде завжди обирати рішення $(1-p)$, другий гравець – $(1-q)$ і в результаті обидва гравці залишаться у найкращому з можливих стабільних становищ, бо поки один з гравців дотримується своєї стратегії, інший гравець не може придумати кращу

альтернативу своїй стратегії, яка принесе йому більшу кількість ресурсів. Також варто зазначити, існує ймовірність того, що дві комірки матимуть два підкреслені числа, як у класичної гри “Battle of the sexes”. У цьому випадку яку бінарну стратегію гравець би не обрав, вона буде частиною одної з можливих рівноваг Неша такої матриці значень.

Другий випадок:

		Воротар	
		q L	(1-q) R
Нападаючий	p L	0,2	5,0
	(1-p) R	3,0	0,7

Рис 1.4 Біматриця, що не містить чисту рівновагу Неша

Візьмемо собі для прикладу наш старий приклад про воротара і нападаючого, тільки тепер розподілимо по-різному кількість балів, які вони отримують, зловивши чи забивши м’яч, який йшов у ліву чи у праву половину воріт. У цьому випадку перший спосіб пошуку рівноваги Неша знайти не вийде, оскільки в жодній з комірок обидва числа підкреслені не будуть. З цього випливає, що бінарна стратегія (тільки наліво чи тільки направо) не буде оптимальним варіантом для гравців. Натомість потрібно знайти правильну змішану стратегію ймовірностей вибору кожному гравцеві, яка буде агентові найбільш вигідна. Стратегією S_i агента i є комбінація цих ймовірностей. А комбінація стратегій усіх агентів – рівновага Неша цієї матриці.

Нехай нападаючий б’є вліво з ймовірністю p. Тоді решта ймовірності $(1-p)$ залишається для удару в праву половину воріт. Крім того, нехай воротар кидається ловити м’яч вліво з ймовірністю q, з чого випливає що кидається вправо з ймовірністю $(1-q)$. Ми можемо знайти оптимальну стратегію для агенті тоді, коли нам не гратиме ролі яке опонент прийме рішення, бо наша

стратегія приноситиме максимально можливий стабільний результат. З цього випливає, що якщо ми шукаємо найбільш оптимальну стратегію для першого гравця (Нападаючого), то для нас не має грати ролі котре з рішень прийме другий гравець (Воротар). Для початку нам необхідно обчислити формулу виплати, яку ми отримуємо, якщо опонент прийняв своє перше, друге, третє рішення, після чого ми їх всі прирівнюємо, бо котре би він не прийняв, для нас ролі воно не грає. Так і знаходимо необхідне нам значення p.

Знаходимо оптимальну стратегію для першого гравця:

$$\Pi_2(L) = p \cdot 2 + (1-p) \cdot 0 = 2p$$
$$\Pi_2(R) = p \cdot 0 + (1-p) \cdot 7 = 7 - 7p$$
$$\Pi_2(L) = \Pi_2(R) \equiv 2p = 7 - 7p \rightarrow p = \frac{7}{9}$$

В результаті отримуємо, що найоптимальнішою стратегією для нападаючого отримувати максимально можливу стабільну кількість балів, згідно з матрицею, є співвідношення $p = \frac{7}{9}$, тобто що бити вліво з ймовірністю $\frac{7}{9}$, а вправо з ймовірністю $\frac{2}{9}$.

Знаходимо оптимальну стратегію для другого гравця:

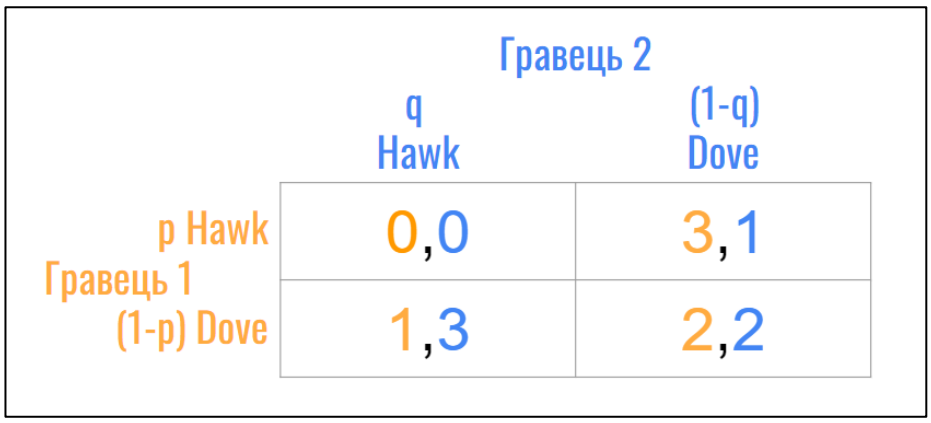
$$\Pi_1(L) = q \cdot 0 + (1-q) \cdot 5 = 5 - 5q$$
$$\Pi_1(R) = q \cdot 3 + (1-q) \cdot 0 = 3q$$
$$\Pi_1(L) = \Pi_1(R) \equiv 5 - 5q = 3q \equiv q = \frac{5}{8}$$

В результаті отримуємо, що найоптимальнішою стратегією для воротаря отримувати максимально можливу стабільну кількість балів, згідно з матрицею, є співвідношення $q = \frac{5}{8}$, тобто що кидатися вліво треба з ймовірністю $\frac{5}{8}$, а вправо з ймовірністю $\frac{3}{8}$.

В результаті бачимо, що знайшли рівновагу Неша, а саме $((\frac{7}{9}, \frac{2}{9}), (\frac{5}{8}, \frac{3}{8}))$.

1.4 Ознайомлення з еволюційною грою “Яструб-Голуб”

Врешті-решт, переходимо до самої гри “Яструб-Голуб”.



1.5 Біматриця нагород гри “Яструб-Голуб”

Для початку пригадаємо загальні правила гри “Яструб-Голуб”, яких повинні дотримуватись усі її коректні імплементації. Гра побудована для двох гравців, кожен з яких може поділити своє населення на певне співвідношення яструбів та голубів. Яструби та голуби займаються пошуком нових ресурсів, але гра не дає їм можливості відбирати чужі ресурси. Після кожної зустрічі двох агентів різних населенн ми можемо отримати один з трьох можливих випадків:

- Якщо зустрілись два яструби, то у кожного з них був шанс у 50% перемогти у боротьбі за фіксовану кількість нових ресурсів, після чого один яструб перемагає і забирає усі нові ресурси. Обидва яструби незалежно від перемоги чи програшу також втрачають фіксовану кількість ресурсів, такий собі податок. На превеликий жаль, цей податок завжди є більшим, ніж кількість ресурсів, яку можна виграти у боротьбі.
- Якщо зустрілись два голуби, то вони у 100% випадках рівно поділяють між собою щойно-знайдені ресурси і дружно йдуть по своїм справам. Жодного податку в такому випадку не має.
- Якщо зустрілись яструб і голуб, то яструб забирає собі усі щойно-знайдені ресурси, а голуб не отримує нічого, проте не втрачає ресурсів, які він назбирав до цієї зустрічі. Жодного податку в такому випадку не має.

Ця гра моделює наступну ситуацію: особи однієї великої популяції зустрічаються випадково, парами, і поводяться або агресивно (Яструб), або пасивно (Голуб). Боротьба ведеться, наприклад, за місця гніздування або території. Ця поведінка є генетично обумовленою, тому людина насправді не вибирає між двома моделями поведінки. Виплати відображають (дарвінівську) придатність, наприклад, кількість нащадків. У цьому контексті гравці 1 і 2 є лише двома різними членами однієї популяції, які зустрічаються: справді, гра є симетричною. Змішана стратегія $\mathbf{p} = (\boldsymbol{p_1}, \boldsymbol{p_2})$ (гравця 1 або гравця 2) природно тлумачиться як вираження частки популяції індивідів, що характеризуються тим самим типом поведінки. Іншими словами, $\frac{p_1}{|p|} \times 100$ населення є яструбами та $\frac{p_2}{|p|} \times 100$ населення є голубами, де $|p|$ – кількість всього населення.

З огляду на цю інтерпретацію, далі нас особливо цікавлять симетричні рівноваги Неша, тобто рівноваги Неша, в яких гравці мають однакову стратегію. Гра “Яструб–Голуб” має три рівноваги Неша, лише одна з яких є симетричною, а саме $((\frac{1}{2}, \frac{1}{2}), (\frac{1}{2}, \frac{1}{2}))$.

Симетрична рівновага зумовлює багато важливих властивостей, які ми будемо використовувати протягом цієї роботи, і тому саме вона нас цікавитиме, але важливо також пам’ятати, що є ще дві інші рівноваги Неша у гри “Яструб-Голуб”, а саме $((1,0),(0,1))$ та $((0,1),(1,0))$ [6]. Розглянемо детально чому за правилами цієї гри це справді вірно.

Візьмемо першу рівновагу $((1,0),(0,1))$. Якщо населення першого гравця буде на 100% складатися з яструбів, то найкраща стабільна стратегія для другого гравця – обрати населення, яке на 100% складається з голубів. Існує багато можливих імплементацій гри “Яструб-Голуб”, проте більшість з них поширюють одну властивість – “Якщо стикаються два яструби, то у кожного з них є шанс перемоги у 50%, а також при кожній зустрічі обидва яструби втрачають фіксовану певна кількість ресурсів незалежно від перемоги чи програшу. Крім того, ця фіксована кількість ресурсів є більшою за ресурси, які можна виграти

протягом зустрічі двох яструбів.”. З цього випливає, що обидвом гравцям не є вигідно грати тільки за яструбів, і їм вигідніше грати обом за голубів, бо не доведеться платити так званий податок за зустріч двох яструбів. Проте якщо обидва гравці грають за голубів, то один з гравців може поліпшити свою стратегію, граючи тільки за яструбів, отримуючи більше ресурсів.

Повертаємось до нашої рівноваги $((1,0),(0,1))$. Якщо перший гравець має тільки яструбів, то другому гравцеві не залишається кращого вибору, чим вибрати населення тільки з голубів, бо **a)** другий гравець не може обрати стратегію, яка стабільно буде йому приносити більшу кількість ресурсів через те, що він може тільки вибирати між яструбами і голубами, і кожен раз, коли він братиме яструба, він ризикує програти, що приводить до ризику гіршої кінцівки, чим якби він завжди тримався голуба, та **b)** перший гравець не може поміняти свою стратегію, бо максимальна можлива кількість ресурсів у нього буде тільки якщо він матиме населення тільки з яструбів. Тому в результаті ми отримуємо рівновагу Неша, нехай і не симетричну. Звісно, що перший і другий гравець можуть помінятися ролями, і ми отримуємо другу рівновагу $((0,1),(1,0))$.

1.5 Поняття еволюційно-стабільної стратегії

Як ми вже знаємо, теорія еволюційних ігор була побудована на Рівновазі Неша. Проте на цьому розвиток теорії еволюційних ігор не зупинився, і у 70х роках англійським біологом Джоном Мейнардом Смітом було вперше введено поняття еволюційно-стабільної стратегії (ЕСС) у його дослідженні стратегій, що зустрічаються у дикій природі [7]. Суть поняття підсумовується, як стратегія, якої якщо буде дотримуватись достатньо велика кількість населення агента, то вона не може бути витісненою жодною іншою з усіх можливих альтернативних стратегій.

Розглянемо абстрактне доведення поняття еволюційно-стабільної стратегії, яке ґрунтується на симетричних еволюційних іграх та симетричній рівновазі Неша. Також варто уточнити визначення **біматриці** – матриця, в кожній колонці якої є по два значення (нагорода першого та другого гравця).

Визначення №1: Нехай $G = (A, B)$ буде біматрицею гри розміром $m \times n$, яка складається з матриць об'єднання значень колонок матриць A, B, які містять виплати гравців гри. Тоді G є **симетричною біматрицею**, якщо $m = n$ та якщо матриця B є транспонованою матрицею A, тобто $B = A^T$. Простіше кажучи, для всіх колонок матриць A, B ми маємо $B_{ij} = A_{ji}$, де $1 < i < m, 1 < j < n$. Знайдена до G рівновага Неша $(p*, q*)$, що складається з сукупності ймовірностей кожного з гравців, буде вважатися **симетричною рівновагою Неша**, якщо $p* = q*$.

Лема №2: кожна симетрична біматриця G містить симетричну рівновагу Неша.

Тримаючи у пам'яті інтерпретації згаданих вище (№1) та (№2) важливо зазначити стосовно симетричної рівноваги Неша наступне:

Нехай $G = (A, B)$ буде симетричною грою. Знаючи, що гра є симетричною, достатньо визначити виплат матриці A, бо оскільки $B = A^T$ згідно з (№1), ми можемо розглядати симетричну гру A, маючи на увазі $G = (A, A^T)$.

Нехай A буде матрицею розміром $m \times m$. Позначимо Δ^m як сукупність можливих змішаних стратегій, тоді:

Визначення №3: стратегія $\boldsymbol{x} \in \Delta^m$ називається еволюційно-стабільною стратегією (ESS) для A, якщо для кожної стратегії $\boldsymbol{y} \in \Delta^m, \boldsymbol{y} \neq \boldsymbol{x}$ існує $\boldsymbol{\varepsilon_y} \in (0,1)$ що для всіх $\boldsymbol{\varepsilon} \in (0, \boldsymbol{\varepsilon_y})$ ми маємо

$$\boldsymbol{x}A\big(\boldsymbol{\varepsilon_y} + (1 - \boldsymbol{\varepsilon})\boldsymbol{x}\big) > \boldsymbol{y}A\big(\boldsymbol{\varepsilon_y} + (1 - \boldsymbol{\varepsilon})\boldsymbol{x}\big)$$

Це визначення [8] ЕСС стратегії $\boldsymbol{x}$ можна трактувати наступним чином: розглянемо довільну невелику за кількістю **мутацію** $(\boldsymbol{\varepsilon_y} + (1 - \boldsymbol{\varepsilon})\boldsymbol{x})$ у стратегії $\boldsymbol{x}$. З визначення (№1) випливає, що під атакою деякої мутації, оригінальна стратегія $\boldsymbol{x}$ буде кращою, ніж довільна стратегія $\boldsymbol{y}$, 100% населення якого дотримується цієї мутації. Іншими словами, якщо населення $\boldsymbol{x}$ є піддане мутації, в результаті якої маленька частка населення змінює свою поведінку на поведінку $\boldsymbol{y}$ (для прикладу: частка населення било пенальті завжди вліво, тепер б'є завжди вправо, тому співвідношення стратегії вліво-вправо змінилось), тоді стратегія $\boldsymbol{x}$, під впливом мутації $\boldsymbol{y}$, буде давати кращий результат, чим даватиме населення, яке на 100% складається з мутації $\boldsymbol{y}$. З цього і походить "еволюційна-стабільність", як отримання найкращої з можливих реакцій при різкому зміненні поведінки деякої частки населення гравця.

Перш ніж застосовувати (№3) до гри "Яструб-Голуб" варто розібрати доведення теореми цікавої властивості еволюційно-стабільної стратегії, оскільки її заключна властивість нам пригодиться.

Теорема №4: Нехай A буде $m \times m$ матрицею та нехай $\boldsymbol{x} \in \Delta^m$ буде ЕСС для A. Тоді $(\boldsymbol{x}, \boldsymbol{x})$ є рівновагою Неша для абстрактної гри $G = (A, A^T)$.

Доведення: нехай $\boldsymbol{y} \in \Delta^m$, тоді достатньо довести що $\boldsymbol{xAx} \geq \boldsymbol{yAx}$, де під xAx мається на увазі імплементація матриці A, в якій обидва гравці дотримуються стратегії $\boldsymbol{x}$. Згідно з (№3) маємо

$$\boldsymbol{x}A(\boldsymbol{\varepsilon_y} + (1 - \boldsymbol{\varepsilon})\boldsymbol{x}) > \boldsymbol{y}A(\boldsymbol{\varepsilon_y} + (1 - \boldsymbol{\varepsilon})\boldsymbol{x})$$

Для усіх $0 < \boldsymbol{\varepsilon} < \boldsymbol{\varepsilon_y}$ згідно з (№1). Дозволивши $\boldsymbol{\varepsilon}$ опуститись до нуля, маємо імплікацію $\boldsymbol{xAx} \geq \boldsymbol{yAx}$. □

Ця теорема показує, що, насправді, еволюційно-стабільна стратегії приводять до симетричної рівноваги Неша [9]. З чого випливає, що якщо потрібно знайти ЕСС, то достатньо приділити увагу лише симетричним рівновагам Неша.

Існує ще друга теорема, яка теж потребує уваги.

Теорема №5: Нехай A буде $m \times m$ матрицею. Якщо $\boldsymbol{x} \in \Delta^m$ є ЕСС, тоді, для усіх можливих $\boldsymbol{y} \in \Delta^m, \boldsymbol{y} \neq \boldsymbol{x}$ ми маємо

$$\boldsymbol{xAx} = \boldsymbol{yAx} \rightarrow \boldsymbol{xAy} > \boldsymbol{yAy}$$

З іншого боку, якщо $(\boldsymbol{x}, \boldsymbol{x}) \in \Delta^m \times \Delta^m$ є рівновагою Неша для $G = (A, A^T)$ і (№2) є задовільною, тоді $\boldsymbol{x}$ є ЕСС.

Доведення: Допустимо, що $\boldsymbol{x} \in \Delta^m$ є ЕСС. Нехай ми маємо $\boldsymbol{y} \in \Delta^m, \boldsymbol{y} \neq \boldsymbol{x}$ а також маємо $\boldsymbol{xAx} = \boldsymbol{yAx}$. Допустимо, що $\boldsymbol{yAy} \geq \boldsymbol{xAy}$. Тоді, для довільного $\boldsymbol{\varepsilon} \in [0,1]$ ми отримаєм $\boldsymbol{yA}\big(\boldsymbol{\varepsilon_y} + (1-\boldsymbol{\varepsilon})\boldsymbol{x}\big) \geq \boldsymbol{xA}\big(\boldsymbol{\varepsilon_y} + (1-\boldsymbol{\varepsilon})\boldsymbol{x}\big)$, що суперечить (№1).

З іншого боку, нехай $(\boldsymbol{x}, \boldsymbol{x}) \in \Delta^m \times \Delta^m$ буде рівновагою Неша для $G = (A, A^T)$ та нехай (№2) дотримується для $\boldsymbol{x}$. Якщо $\boldsymbol{xAx} > \boldsymbol{yAx}$, тоді ми отримуємо $\boldsymbol{xA}\big(\boldsymbol{\varepsilon_y} + (1-\boldsymbol{\varepsilon})\boldsymbol{x}\big) > \boldsymbol{yA}\big(\boldsymbol{\varepsilon_y} + (1-\boldsymbol{\varepsilon})\boldsymbol{x}\big)$ для достатньо малого значення $\boldsymbol{\varepsilon}$. Якщо $\boldsymbol{xAx} = \boldsymbol{yAx}$, тоді $\boldsymbol{xAy} > \boldsymbol{yAy}$, з чого випливає (№1) дотримується для довільного $\boldsymbol{\varepsilon} \in (0,1]$. □

Дві попередні теореми стверджують, що еволюційно-стабільні стратегії $\boldsymbol{x}$ виникають тільки у симетричних рівновагах Неша (№4) та справляються строго краще під впливом деякої мутації $\boldsymbol{y}$, ніж населення, яке на 100% складається з мутації $\boldsymbol{y}$.

Тому ми можемо дійти наступного висновку: еволюційно-стабільні стратегії для матриці A розміру $m \times m$ можна знайти наступним чином:

I. Обчислити симетричну рівновагу Неша для матриці $G = (A, B)$, в якої $B = A^T$. Способи обчислення рівноваги Неша були розглянуті у розділі **1.3.** цієї роботи.

II. Для знайденої симетричної рівноваги Неша $(\boldsymbol{x}, \boldsymbol{x})$ перевірити чи (№2) є задовільною. Якщо так, тоді $\boldsymbol{x}$ є еволюційно-стабільною стратегією.

Врешті-решт, розглянемо на прикладі нашої гри "Яструб-Голуб". Пригадаємо, що біматриця, яку ми раніше використовували, виглядає наступним чином:

		Гравець 2	
		q Hawk	(1-q) Dove
Гравець 1	p Hawk	0,0	3,1
	(1-p) Dove	1,3	2,2

Рис 1.6 Біматриця нагород гри "Яструб-Голуб"

Згідно з теорією, одною єдиною симетричною рівновагою Неша у гри "Яструб-Голуб" є $\boldsymbol{x} = (\frac{1}{2}, \frac{1}{2})$. Нехай $\boldsymbol{y} = (y, 1 - y)$ буде довільною стратегією. В такому разі умова $\boldsymbol{xAx} = \boldsymbol{yAx}$ у (№2) завжди буде задовільна, бо воно випливає з факту того, що $(\boldsymbol{x}, \boldsymbol{x})$ є рівновагою Неша. Тоді, нам необхідно переконатись лише в тому, що $\boldsymbol{xAy} > \boldsymbol{yAy}$ є задовільною для усіх $\boldsymbol{y} = (y, 1 - y) \neq \boldsymbol{x}$.

Для початку, обчислимо $\boldsymbol{xAy}$, що в нашому випадку рівне $\frac{1}{2}A\boldsymbol{y}$.

		Гравець 2	
		y	(1-y)
Гравець 1	1/2	0	3
	1/2	1	2

1/2Ay

Рис 1.7 Біматриця нагород гравця 1 зі стратегією $(\frac{1}{2}, \frac{1}{2})$

Маємо $\boldsymbol{xAy} \equiv \frac{1}{2} \cdot (0 \cdot y + 3 \cdot (1 - y)) + \frac{1}{2} \cdot (1 \cdot y + 2 \cdot (1 - y))$

$\equiv \frac{1}{2} \cdot (3 - 3y) + \frac{1}{2} \cdot (y + 2 - 2y)$

$\equiv \frac{3 - 3y + y + 2 - 2y}{2}$

$\equiv \frac{-4y + 5}{2}$

Далі нам необхідно обчислити $\boldsymbol{yAy}$.

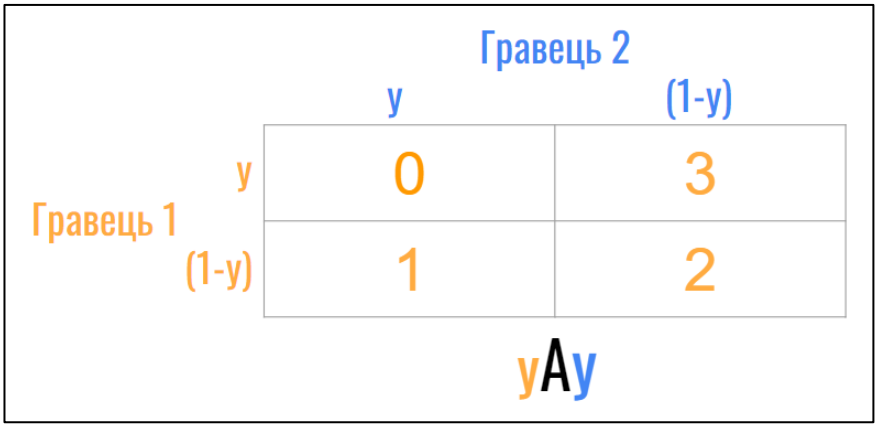


Рис 1.8 Біматриця нагород гравця 1 зі стратегією $(y, 1 - y)$

Маємо $\boldsymbol{yAy} \equiv y \cdot (0 \cdot y + 3 \cdot (1 - y)) + (1 - y) \cdot (1 \cdot y + 2 \cdot (1 - y))$

$\equiv \boldsymbol{y} \cdot (3 - 3y) + (1 - y) \cdot (2 - y)$

$\equiv 3y - 3y^2 + 2 - y - 2y + y^2$

$\equiv -2y^2 + 2$

Тоді $\boldsymbol{xAy} > \boldsymbol{yAy} \equiv \frac{-4y+5}{2} > -2y^2 + 2$

$\equiv -4y + 5 > -4y^2 + 4$

$\equiv 4y^2 - 4y + 1 > 0$

$\equiv (2x - 1)^2 > 0$, що є задовільною умовою для усіх значень y, окрім $\frac{1}{2}$, яким він як раз не може рівний, бо $\boldsymbol{x} \neq \boldsymbol{y}$. Тому ми урочисто довели, що $\boldsymbol{x} = (\frac{1}{2}, \frac{1}{2})$ є унікальною еволюційно-стабільною стратегією для A.

2. Розробка прототипу гри “Яструб-Голуб”

2.1 Visual Studio Code

Перед тим, як приступити до побудови нашого проекту, варто ознайомитись з програмним середовищем, у якому ми будемо працювати.

Почнімо з текстового редактору. Оскільки побудова симуляції гри “Яструб-Голуб” не є доволі складного для потреби у багатомодульному додатку, то програмні середовища, як Visual Studio, PyCharm чи IntelliJ IDEA тільки ускладнюватимуть наш майбутній проект.

Натомість можна обійтися звичайним текстовим редактором. Текстові редактори застосовують для розробки невеликих в об’ємі проектів, які можна розмістити в одному або декількох файлів всередині одної директорії. Всесвітньо визнаними приклади таких текстових редакторів є Visual Studio Code від Microsoft, Atom від GitHub, Notepad++ від в’єтнамського розробника Don Ho, або навіть Vim від голландського розробника Bram Moolenaar, якщо ви великий шанувальник роботи у терміналі та переконування людей, як насправді це зручно.

Мною було вибрано текстовий редактор Visual Studio Code.
Visual Studio Code (VS Code) - це безкоштовний текстовий редактор, розроблений компанією Microsoft. Він підтримує роботу на багатьох платформах, включаючи Windows, macOS, Linux та RPi OS. VS Code відзначається своєю широкою функціональністю, легкістю використання та розширюваністю.

VS Code є хорошим вибором для невеликих по об'єму проектів з-за своєї легкості використання. Він має інтуїтивний і зручний інтерфейс, що сприяє швидкому навчанню та продуктивності. Також, завдяки своїй швидкості, він працює добре навіть з великими проектами.

Одна з основних переваг VS Code порівняно з іншими текстовими редакторами, такими як Atom і Notepad++, полягає в його розширюваності. Він має велику кількість розширень (extensions), які дозволяють додавати нові функціональність і підтримку різних мов програмування. Розширення дозволяють налаштувати редактор під власні потреби та розширити його можливості.

Крім того, VS Code володіє великою кількістю інтеграцій зі сторонніми програмами. Наприклад, він має глибоку інтеграцію з системою контролю версій Git, що спрощує роботу з кодом, комітами та гілками. Також, VS Code підтримує інші популярні інструменти розробки, такі як дебагери, термінали, системи збирання проектів і багато інших.

VS Code має багато розширень, які забезпечують підтримку різних мов програмування. Ви знайдете розширення для популярних мов, таких як JavaScript, Python, Java, C++, PHP, Ruby та багато інших. Ці розширення надають функціональність, таку як підсвічування синтаксису, автодоповнення, перевірку помилок, рефакторинг коду та інше. Багатомовна підтримка робить VS Code універсальним інструментом для розробки на різних мовах програмування.

Щодо інтеграції з середовищами виконання, VS Code має відмінну підтримку для розробки в середовищі .NET. Ви можете встановити розширення для підтримки C# та розробки ASP.NET-додатків.
Також, VS Code має інтеграцію з Unity - популярним движком для розробки ігор. Розширення для Unity дозволяють вам розробляти, налагоджувати та керувати вашими проектами Unity, прямо з VS Code.

VS Code також надає можливості інтеграції з Docker, що дозволяє легко працювати з контейнеризованими додатками. З допомогою розширення Docker,

ви можете створювати, запускати та керувати контейнерами, правити Dockerfile та взаємодіяти з Docker-хостами безпосередньо з VS Code.

Крім того, VS Code має глибоку інтеграцію з хмарними сервісами, зокрема Microsoft Azure. З допомогою розширення Azure, ви можете розробляти, підлагоджувати та керувати вашими хмарними ресурсами Azure, такими як веб-додатки, функції, контейнери та інші. Ця інтеграція спрощує розробку та розгортання вашого додатку в хмарному середовищі.

Вкладка Extensions у Visual Studio Code має велику перевагу для звичайних користувачів, оскільки вона дає їм змогу доповнювати та налаштовувати редактор згідно зі своїми потребами. Звичайні користувачі можуть встановлювати розширення з легкістю без необхідності писати власний код або мати глибокі знання розробки. Це можуть бути розширення для автодоповнення коду, підсвічування синтаксису, розширеної підтримки Git, інструментів для форматування коду, роботи з базами даних, роботи з фреймворками та багато інших.

Загалом, Visual Studio Code - це потужний і розширюваний текстовий редактор, який володіє зручним інтерфейсом, широким спектром розширень і глибокою інтеграцією зі сторонніми програмами. Він є популярним вибором серед розробників, які працюють над невеликими проектами або потребують швидкого та налаштованого середовища для програмування

.

2.2 Git

Одним лиш текстовим редактором обходитись нема потреби. Для розробки нашого проекту пригодиться це Git. Всупереч популярній думці, Git є корисним інструментом навіть для програмістів, які працюють самі.

Git є розподіленою системою керування версіями, яка забезпечує збереження, відстеження та керування змінами у програмному коді [10]. Він є незамінним інструментом у репертуарі розробника програмного забезпечення.

Одним з найважливіших понять у сфері Git є так званий “коміт”. Коміт використовується для збереження змін у репозиторії. Коміт представляє собою фіксацію конкретного стану файлів в певний момент часу. Коли ви робите коміт у Git, ви створюєте новий "знімок" вашого проекту, який включає всі зміни, внесені після попереднього коміту. Кожен коміт включає повідомлення, яке пояснює, які зміни були зроблені в цьому коміті.

Одна з важливих переваг Git полягає в його здатності зберігати коміти (зміни) локально на комп'ютері розробника. Це означає, що ви можете зберігати проміжні зміни та експериментувати з кодом, не впливаючи на спільний репозиторій. Таким чином, Git дозволяє розробникам працювати незалежно та ефективно навіть у випадку, коли вони працюють самостійно.

Крім того, Git надає можливість публічного представлення комітів, коли ви готові поділитися своїм кодом з іншими [11]. Ви можете завантажити свої коміти на віддалений репозиторій, щоб інші розробники могли переглядати, вносити зміни та співпрацювати з вами над проектом. Відомими клієнтами для використання віддалених репозиторіїв є сервіси GitHub та GitLab.

Для роботи з Git на локальному рівні розробник може використовувати різні інтерфейси, включаючи командний рядок через Git Bash. Git Bash надає

командний інтерфейс, який дозволяє виконувати команди Git на рівні командного рядка вашої операційної системи.

Основні команди Git, які використовуються розробниками, включають:

git config: Ця команда використовується для налаштування конфігурації Git на вашому комп'ютері, такі як ім'я користувача та електронна пошта.

git init: Ця команда створює новий репозиторій Git в поточній папці. Вона ініціалізує порожній репозиторій, де ви будете зберігати ваші зміни.

git add: Ця команда додає зміни файлів до індексу Git. Вибіркове додавання дозволяє вам вибрати конкретні файли або папки, які потрібно включити до наступного коміту.

git rm: Ця команда видаляє файли з репозиторію. Ви можете вказати файли, які потрібно видалити, і Git відслідкує це в подальших комітах.

git status: Ця команда показує поточний стан вашого репозиторію Git. Вона вказує, які файли змінені, які файли додані до індексу та які файли очікують на коміт.

git commit: Ця команда створює коміт зі змінами, які ви додали до індексу. Ви повинні включити повідомлення коміту, що пояснює зміни, які ви внесли.

git checkout: Ця команда дозволяє переключатися між гілками в репозиторії або відновлювати стан файлів до попереднього коміту.

git push: Ця команда завантажує ваші локальні коміти на віддалений репозиторій, такий як GitHub або GitLab. Вона забезпечує синхронізацію змін між вашим локальним репозиторієм і віддаленим репозиторієм.

Використання цих команд дозволяє розробникам ефективно керувати змінами в своїх проектах, створювати коміти, відслідковувати стан файлів, співпрацювати з іншими розробниками та зберігати код у безпечному репозиторії.

2.3 Python

Урочисто доходимо до останнього програмного забезпечення, яке нам знадобиться для розробки нашого проекту – мова програмування.

Python - це високорівнева, інтерпретована мова програмування, яка була розроблена Гвідо ван Россумом у 1991 році. Її назва походить від захоплення Гвідо комедійним шоу "Монті Пайтон", що в оригіналі “Monty Python”. Він створив Python з метою створення простої та зрозумілої мови програмування, яка б забезпечувала швидкий та ефективний розвиток програм.

Однією з головних переваг Python є його простота та читабельність. Синтаксис мови спрощений та зрозумілий, що допомагає програмістам швидко освоїти мову та розробляти проекти без зайвих труднощів. Python покликаний бути "властивим читанню наче простого тексту" і має наочний стиль кодування, який полегшує співпрацю між розробниками та підтримку вже наявного коду.

Python володіє великою екосистемою модулів та пакетів, яка робить його ідеальним вибором для різноманітних проектів. Бібліотека стандартних модулів Python надає розширені можливості для роботи з файлами, мережами, базами даних, потоками, регулярними виразами та багатьма іншими завданнями розробки програмного забезпечення.

Python також відомий своїми великими можливостями у галузі аналізу даних та візуалізації. Модуль **pandas** надає потужні засоби для роботи з даними, такими як завантаження, фільтрація, групування, агрегування та аналіз. В поєднанні з бібліотеками візуалізації, такими як **matplotlib** та **seaborn**, Python стає потужним інструментом для візуалізації та розуміння даних.

Python також підтримує розробку графічних інтерфейсів користувача (GUI). Модуль **tkinter** надає інтерфейс для розробки графічних додатків з

використанням різних елементів керування, таких як кнопки, поле введення, вікна та інші. Для графічної розробки ігор, модуль **pygame** надає можливості для створення графіки, обробки подій та взаємодії з користувачем.

Python також славиться своєю широкою підтримкою та активною спільнотою розробників. Велика кількість модулів та пакетів доступна для використання, що спрощує розробку проектів та прискорює процес розробки. Python має велику кількість документації, навчальних матеріалів та форумів, що допомагає розробникам отримати допомогу та вирішити проблеми, які виникають під час розробки.

Загалом, Python є чудовим вибором для розробки проектів завдяки своїй простоті, сильним можливостям у візуалізації та аналізі даних, підтримці розробки графічних інтерфейсів та великій спільноті розробників. Він є однією з найпопулярніших мов програмування, використовуваних у багатьох галузях, включаючи науку про дані, веб-розробку та штучний інтелект.

У цьому розділі ми коротко згадали модуль **tkinter**, який пізніше буде розглянуто у третьому розділі цієї роботи детальніше для побудови графічного інтерфейсу програми, оскільки він славиться своєю простотою та інтуїтивністю у використанні, як і сам Python.

2.4 Планування прототипу гри “Яструб-Голуб”

Врешті-решт, переходимо до самої розробки гри. Як вже було декілька разів згадано у назвах розділів, наше завдання зараз є розробка прототипу. Під прототипом мається на увазі побудова робочої демо-версії нашої гри, розробка функцій для представлення логіки гри, оформлення функцій для використання користувачем та поліпшення якості програми для уникнення вразливостей від помилкового вводу даних користувачем.

Побудуємо приблизний план, як ми собі уявляємо гру.
Простіше всього буде візуалізувати симуляцію еволюційної гри для двох гравців. Оскільки інтерактиву наша еволюційна гра не має, користувач може задавати процент населення яструбів та голубів для кожного з гравців.

Потрібно ще зрозуміти, як визначити переможця серед двох гравців. Згідно з ігровою теорією, в еволюційної гри “Яструб-Голуб” є два варіанти як можна задати правила гри, щоб обирати переможця. Перший спосіб – це поставити певну ціль ресурсів, перший гравець досягнувши якої буде перемагати у грі. Другий спосіб – провести n кількість сутичок між населеннями гравців і порівняти хто зміг набрати більше ресурсів по завершенню останньої з сутичок.

Вдалою думкою буде імплементувати обидва способи у нашому проекті, проте для різних застосунків. Сфокусуємося на цілях проекту. Користувач зможе користуватись нашим проектом для симуляції гри “Яструб-Голуб”, задавши стратегії популяцій двох персонажів і визначення кращої з двох. Оскільки у залежності від вибраних стратегій гра може стати вельми динамічною, потрібно провести велику кількість симуляцій щоб визначити середнє значення і підсумувати яка ж стратегія загалом є кращою з двох.

В такому разі розумним рішенням є використати перший спосіб для визначення переможця гри – задання цілі ресурсів, перший гравець досягнувши якої стає переможцем. Тоді користувачу треба ще буде дати можливість задати початкову кількість ресурсів та кінцеву кількість ресурсів для персонажів. Врешті-решт, потрібно дати можливість користувачу вирішувати яку кількість симуляцій гри потрібно провести, щоб визначити кращу стратегію. Простим і зрозумілим варіантом буде відобразити процент виграшу першого гравця та процент другого гравця, бо давати за користувача готову відповідь що краще буде для нього/неї не надто інформативно.

Визначення кращої з введених стратегій це важливо, проте цього не достатньо. Як вже було розглянуто у теоретичних відомостей цієї роботи, одна з головних ідей еволюційних ігор закладається у знаходженні найкращої з усіх можливих стратегій під твого противника. Для цієї задачі нам згодиться другий спосіб визначення переможця чудово.

У випадку коли ми намагаємось визначити найкращу з можливих стратегій під конкретного противника, ми можемо застосувати перший спосіб визначення переможця, і порівнювати яка з стратегій за меншу кількість сутичок дійшла бажаної кінцевої цілі, але в такому разі різниця між сусідніми по співвідношенням стратегіями може бути однакове, або мінімальне, що обезцінює таку мірю вимірювання кращої стратегії.

Натомість другий спосіб фокусується на отриманій кількості ресурсів за n-ну кількість сутичок проти конкретного опонента. Звісно важливо провести велику кількість симуляцій не тільки кожної з можливих симуляцій, але й отриманих середній значень з симуляцій, на випадок високої динамічності стратегій у грі, проте важливо що кількість балів це об’єктивна міра якості вимірювання кращої стратегії, то за її допомогою нам буде просто визначити найкращу.

В підсумку маємо ідею застосунку двох способів визначення переможця еволюційної гри “Яструб-Голуб” всередині нашого проекту, який вмітиме як і провести n кількість симуляцій та порівняти яка з стратегій двох гравців є кращою, так і визначити найкращу стратегію під довільного противника у контексті нашої гри.

Тепер зосередимо увагу на програмі з позиції користувача. Наш користувач хоче мати можливість зручно виконувати ці дві функції. Зрозуміло, що вникати в те, як ми їх назвали та як їх викликати він/вона не буде, то нам потрібно дати користувачу деяке “головне меню”, де буде пропонуватись виконати одну з цих двох функцій. Хоч деякі аспекти ігри у її традиційному понятті ми таки можемо імплементувати у нашому проекті, як і всі відеоігри за останні 30 років наша теж матиме своє Main menu.

Окрім наших двох функцій користувачу потрібно дати можливість задавати стратегії для гравців. Очевидним є рішення зробити це окремою кнопкою у головному меню, бо по-перше двох кнопок у головному буде замало для головного меню, а по-друге контр-інтуїтивно просити користувача повторно вводити дані, які вже ввів до того для іншої функції. Врешті-решт, потрібно ще додати кнопку виходу з гри, і урочисто назвати цю панель опцій головним меню.

З боку користувача звісно буде приємно ще отримати зручний графічний інтерфейс, наявність якого стала не стільки приємним бонусом, скільки відсутність якого стала великим недоліком. Поки що сфокусуємось на логічному аспекті побудови такого проекту, а графічний інтерфейс намалюємо вже у наступному розділі.

2.5 Розробка логіки гри “Яструб-Голуб” для двох гравців

Процес інсталяції програмного забезпечення з пунктів 2.1., 2.2., 2.3. не є важливим для розгляду, оскільки послідовно дотримуючись інструкцій інсталяторів жодних проблем з жодним з цих п/з не виникне. Без лишніх слів, відкриваємо наш текстовий редактор Visual Studio Code та створюємо та переходимо у директорію для нашого проекту під назвою “Diplom”.

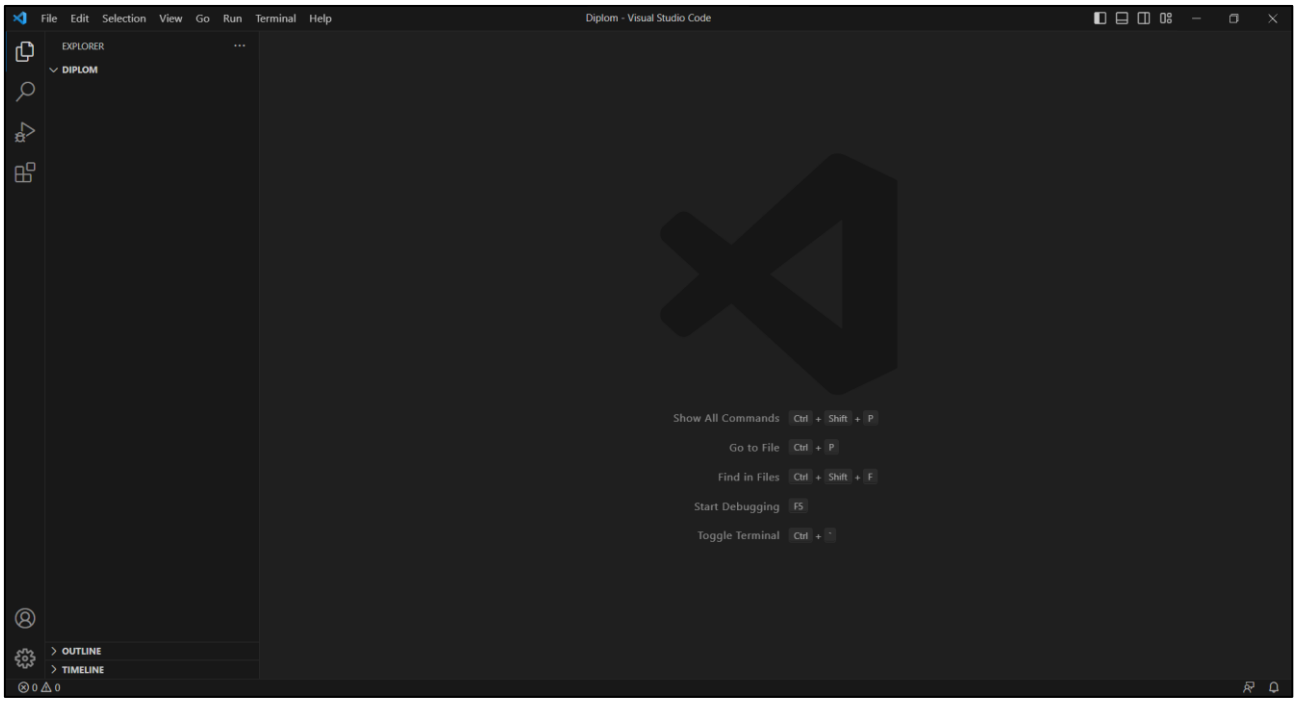


Рис 2.1 Стартове вікно Visual Studio Code

Відкриваємо наш Git Bash. Переходимо у директорію, яку ми щойно створили. У випадку того, що Git Bash був щойно встановлений на систему, то нам необхідно ініціалізувати глобальні змінні, значення яких використовуються для відображення хто затвердив нову версію програми. Користуючись командами

git config -- global user.name “Yaroslav Svoren”

git config --global user.email “yaroslav.svoren@lnu.edu.ua”

git config --global core.editor “code --wait”

задаємо ім’я, пошту та текстовий редактор по замовчуванню. Важливо підмітити, що як і у самому терміналі Linux, у Git Bash стрічка “code” належить саме Visual Studio Code. Вказуючи який текстовий редактор ми вибрали, ми вказуємо додатковий параметр “--wait”, щоб у випадку якщо Git Bash відкриє текстовий редактор для редагування великої кількості змінних, сам Git Bash перебував у стані очікування до того моменту, поки не закриється вікно текстового редактору.

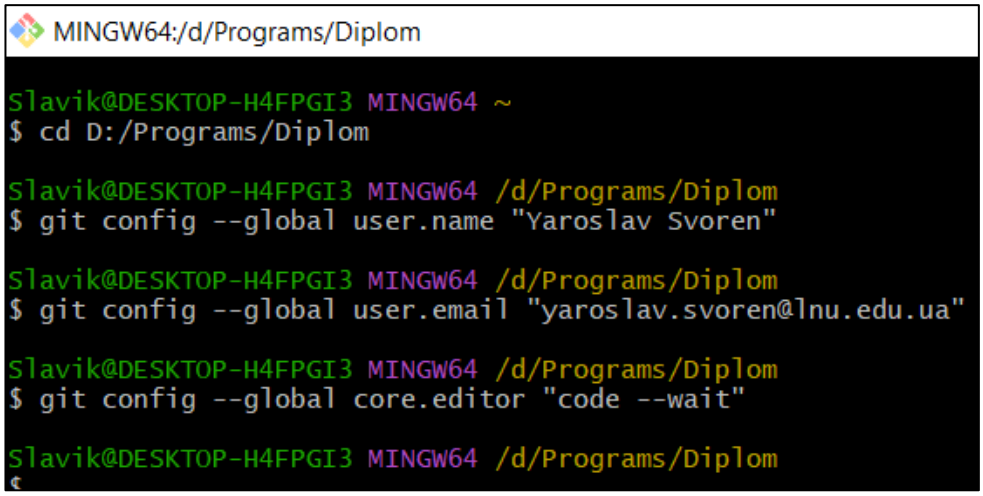


Рис 2.2 Задання глобальних змінних користувача Git

Тепер ми можемо ініціалізувати нашу локальну репозиторію. Вміст такої репозиторії зазвичай потім відсилають віддалену репозиторію на клієнтах GitHub чи GitLab, але оскільки ми виконуємо цей проект самотужки, то локальної репозиторії вистачить. Варто звернути увагу. Що Git Bash вже каже нам в якій гілці ми знаходимось, тобто у гілці “master”.

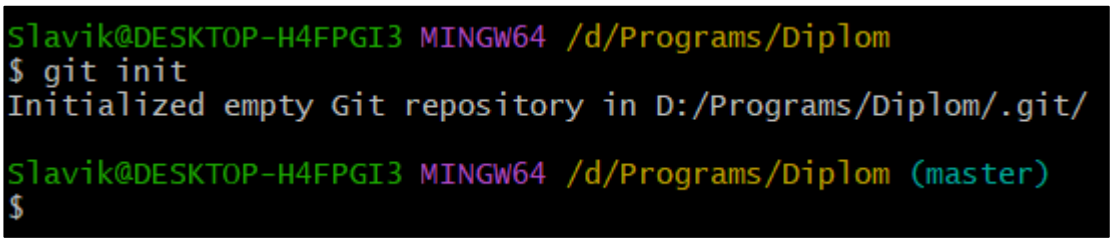


Рис 2.3 Ініціалізація репозиторії у Git

Переконатися в тому, чи були внесені якісь зміни до вмісту папки ми можемо командою **git status**, в результаті чого бачимо, що папка поки пуста.

```
Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$
```

Рис 2.4 Перевірка статусу репозиторії

Створюємо наш прототип під назвою **prototype.py**, над яким буде вестися робота у цьому розділі.

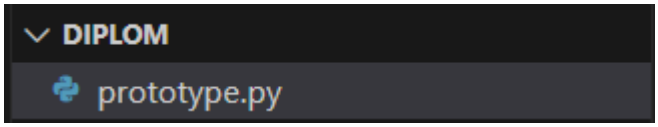


Рис 2.5 Файл прототипу програми

З цікавості введемо знову опцію **git status**, в результаті чого бачимо, що появився новий файл, які поки не належить до локальної репозиторії. Дочекаємось до наступної віхи після створення пустого файлу, перш ніж забивати наш локальний репозиторій.

```
Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        prototype.py

nothing added to commit but untracked files present (use "git add" to track)

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
```

Рис 2.6 Зміна статусу репозиторії Git

Переходимо до створення програми. Створюємо клас **Player**, який міститиме усі згадані нами вище змінні, які буде у майбутньому задавати користувач. Було непросто зрозуміти як правильно вирішувати хто з населення гравця попаде в сутичку за ресурси, то оптимальним рішенням стало використання модулю

random, в якому міститься однойменний метод, що повертає випадкове значення на проміжку **[0, 1]**. Клас **Player** отримує процент населення, яке у ньому буде яструбами у змінній **hawk_percentage** та голубами у змінній **dove_percentage**, після чого за допомогою методу **play_strategy()** генерується випадкове число на проміжку **[0, 1]** і якщо воно є нижче проценту населення, що є яструби, то з населення гравця на сутичку вирушає яструб. Якщо ж згенероване число є вище проценту населення, що є яструбами, тоді вирушатиме на сутичку голуб.

prototype.py > ...

```python
import random

class Player:
    def __init__(self, hawk_percentage, dove_percentage, start_resources):
        self.hawk_percentage = hawk_percentage
        self.dove_percentage = dove_percentage
        self.resources = start_resources
        self.default_resources = start_resources
        self.wins = 0

    def play_strategy(self):
        r = random.random()
        if r < self.hawk_percentage:
            return "hawk"
        else:
            return "dove"
```

Рис 2.7 Клас Player

Оскільки за нашим планом інших функцій гравець гри виконувати не повинен, то відправляємо поточний вміст **prototype.py** у локальну репозиторію командою **git add prototype.py** для підготовки файлу, потім **git status** для створення версії репозиторії, що утримує усі підготовані файли. Перед використанням **git commit** можна знову викликати **git status**, який повідомить, що всі файли у папці підготовлені.

```
Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git add prototype.py

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   prototype.py


Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git commit -m "Player class added"
[master (root-commit) bdaefa2] Player class added
 1 file changed, 16 insertions(+)
 create mode 100644 prototype.py

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
```

Рис 2.8 Зміна статусу репозиторії Git

З представленням гравця у програмі справились, тепер потрібно сформувати симуляцію гри. Створюємо клас **Simulation**, в якому будуть виконуватись усі майбутні дії проекту.

```
18    class Simulation:
```

Рис 2.9 Клас Simulation

Згідно з нашим планом, користувач повинен задавати значення тільки один раз, після чого він матиме можливість їх поміняти звісно, але усі функції повинні пам’ятати що він ввів з першого разу. Зрозуміло, що користувач, який вперше відкрив нашу програму, не повинен наперед знати, що він повинен нажимати кнопку для введення початкових стратегій гравцям, це має бути інтуїтивно. Тоді головне меню користувача зустрічати не буде, а натомість починатиметься робота програми саме з вікна вводу даних, після чого все переходитиме у головне меню.

Наша симуляція створена для двох гравців, то у конструкторі симуляції варто містити об’єкти обох гравців. Оскільки ми очікуємо, що користувач може захотіти поміняти введенні дані, то замість задання значень змінним у конструкторі створимо окрему функцію **initialize_simulation()**, виклик якої ми пізніше використаємо у конструкторі. Ця функція приймає параметрами процент яструбів та голубів обох гравців, а також кількість стартових ресурсів, бажану

кінцеву кількість ресурсів та кількість симуляцій, які ми будемо в майбутньому проводити для визначення кращої стратегії.

```
27      def initialize_simulation(self, player1_hawk_percentage, player1_dove_percentage, player2_hawk_percentage,
28                                player2_dove_percentage, start_resources, goal_resources, num_simulations):
29          self.player1 = Player(player1_hawk_percentage, player1_dove_percentage, start_resources)
30          self.player2 = Player(player2_hawk_percentage, player2_dove_percentage, start_resources)
31          self.goal_resources = goal_resources
32          self.num_simulations = num_simulations
```

Рис 2.10 функція initialize_simulation() для задання змінних класу Simulation

Також нам потрібно бути готовим до того, що користувач випадково чи навмисно введе помилкові дані, як наприклад сума проценту населення яструбів та голубів гравця не буде рівна одиниці, з чого випливає що було дані не про чітко 100% населення. Також очевидно поставити перевірку того, що кількість стартових ресурсів не може бути від'ємною, що бажана кінцева кількість ресурсів не може бути меншою за стартову кількість, і що кількість симуляцій повинна бути більшою нуля.

Ініціалізуємо відповідні локальні змінні у функції, після чого у циклі, з якого можна вийти тільки коли усі перечисленні умови введення будуть виконані, приймаємо значення від користувача. На випадок того, що користувач замість числа введе стрічку складаємо окрему функцію **take_input()**, яка не дозволить користувачу перейти до введення значення наступної змінної до поки не дасть прийнятне значення для поточної.

```
114     def take_input(self, display_message):
115         while True:
116             user_input = input(display_message)
117             try:
118                 value = float(user_input)
119                 return value
120             except ValueError:
121                 print("Error: Invalid input.")
```

Рис 2.11 Функція take_input() для перевірки вводу користувача

Користуючись **take_input()** отримуємо значення від користувача, і перевіряємо чи вони прийнятні. На випадок того, що користувач помилився створюємо булеву змінну **first_attempt**, яка набуває значення **False** після першої ітерації циклу, в результаті чого у випадку якщо цикл запустить другу ітерацію, то користувач побачить повідомлення, яке нагадуватиме що процент голубів та яструбів повинен сумуватися у рівно одиницю. Після того, як користувач дав прийнятні значення, викликаємо **initialize__simulation()** для збереження введених користувачем даних.

```
123     def input_strategies(self):
124         p1_hawk, p1_dove, p2_hawk, p2_dove, goal_resources, start_resources, num_simulations = 0, 0, 0, 0, 0, 0, 0
125         first_attempt = True
126 
127         while p1_hawk + p1_dove != 1 or p2_hawk + p2_dove != 1 or start_resources < 0 or goal_resources < start_resources:
128             if not first_attempt:
129                 print("Wrong values.\nEach player's Hawk percentage and Dove percentage must sum up to 1.\nTry again.\n")
130             first_attempt = False
131             p1_hawk = self.take_input("Enter Player1 Hawk percentage ([0, 1]):")
132             p1_dove = self.take_input("Enter Player1 Dove percentage ([0, 1]):")
133             p2_hawk = self.take_input("Enter Player2 Hawk percentage ([0, 1]):")
134             p2_dove = self.take_input("Enter Player2 Dove percentage ([0, 1]):")
135             start_resources = int(self.take_input("Enter amount of starting resources (>0):"))
136             goal_resources = int(self.take_input("Enter the win amount of resources (>start):"))
137             num_simulations = int(self.take_input("Enter the number of simulations for testing (>0):"))
138             print()
139 
140         self.initialize_simulation(p1_hawk, p1_dove, p2_hawk, p2_dove, start_resources, goal_resources, num_simulations)
```

Рис 2.12 Функція input_strategies() для отримання вводу від користувача

Можемо нарешті тепер скласти наш конструктор класу. Думаючи наперед даємо можливість конструктору задавати значення у нього. В такому разі конструктор передаватиме їх до **initialize_simulation()**. Така можливість нам пригодиться коли ми будемо генерувати вагому кількість симуляцій, стратегії яких наша програма не матиме можливості неправильно задати. Альтернатива з введенням їх вручну, коли кількість симуляцій – 200 зумовлює таку можливість зробити.

```
18  class Simulation:
19      def __init__(self, player1_hawk_percentage=None, player1_dove_percentage=None, player2_hawk_percentage=None,
20                   player2_dove_percentage=None, start_resources=None, goal_resources=None, num_simulations=None):
21          if(player1_hawk_percentage != None):
22              self.initialize_simulation(player1_hawk_percentage, player1_dove_percentage, player2_hawk_percentage,
23                                         player2_dove_percentage, start_resources, goal_resources, num_simulations)
24          else:
25              self.input_strategies()
```

Рис 2.13 Конструктор класу Simulation з необов’язковим вводом користувача

Оскільки ми маємо суттєвий прогрес у нашому проекті, додаємо нову версію нашого **prototype.py** у локальну репозиторію командами **git add, git commit**. Якщо цього разу вже ми введемо **git status**, то Git нам скаже, що помітив, що вміст файлу **prototype.py** було модифіковано.

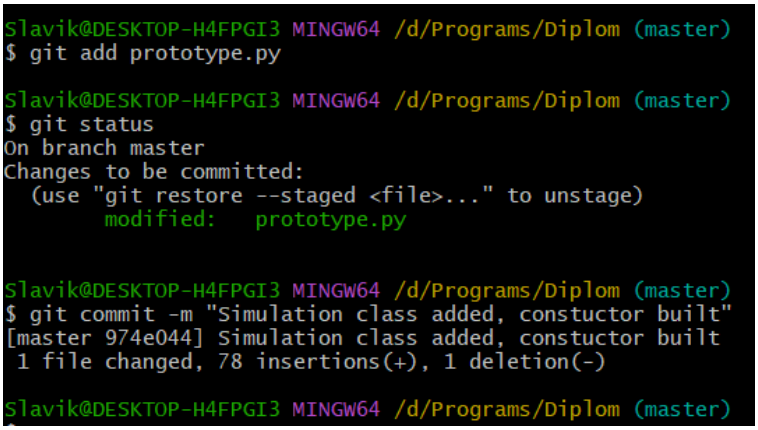


Рис 2.14 Зміна статусу репозиторії Git

Врешті-решт, можна будувати саму гру “Яструб-Голуб”. У цьому підрозділі зупинимось на першій функції проекту, а саме грі двох гравців та визначенні у кого з гравців краща стратегія. Пригадаємо на хвилинку правила гри, які ми розбирали у підрозділі 1.4. .

- Якщо зустрілись два яструби, то у кожного з них був шанс у 50% перемогти у боротьбі за фіксовану кількість нових ресурсів, після чого один яструб перемагає і забирає усі нові ресурси. Обидва яструби незалежно від перемоги чи програшу також втрачають фіксовану кількість ресурсів, такий собі податок. На превеликий жаль, цей податок завжди є більшим, ніж кількість ресурсів, яку можна виграти у боротьбі.
- Якщо зустрілись два голуби, то вони у 100% випадках рівно поділяють між собою щойно-знайдені ресурси і дружно йдуть по своїм справам. Жодного податку в такому випадку не має.
- Якщо зустрілись яструб і голуб, то яструб забирає собі усі щойно-знайдені ресурси, а голуб не отримує нічого, проте не втрачає ресурсів, які він назбирав до цієї зустрічі. Жодного податку в такому випадку не має.

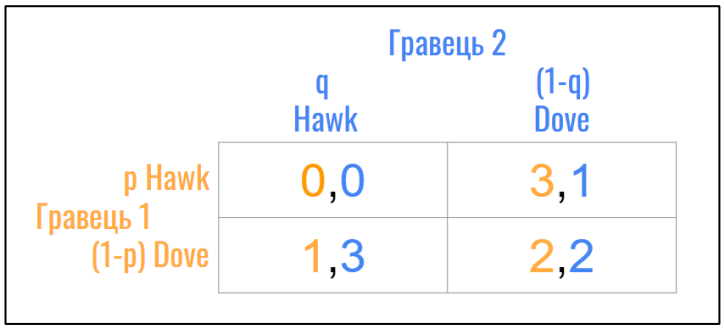


Рис 2.15 Біматриця нагород гри “Яструб-Голуб”

Тепер виникає питання як правильно це імплементувати. У підрахунку біматриці вказується **[0, 0]** у віконці зустрічі двох яструбів, бо у кожного з них є 50% шанс отримати нові ресурси, які в будь-якому разі будуть нижчими за податок, які кожен яструб повинен платити при зустрічі двох яструбів.

У випадку якщо зустрінеться голуб та яструб, то кількість ресурсів отриманих не залежить від жодних випадковостей, тому гравець з яструбом отримає 3 одиниці ресурсів, а голуб одну.

У випадку якщо два голуби зустрінуться, то кількість ресурсів поділять пополам. При тестуванні програми було виявлено, що в такому разі кожен гравець повинен отримувати одну одиницю ресурсів для більш репрезентабельної поведінки у програмі. Зрештою, кількість ресурсів у кожному випадку це питання імплементації, бо в еволюційної гри “Яструб-Голуб” чітких правил по нагородам нема, як у тих ж шахматах чи шашках.

Створюємо функцію **clash()**, яка буде поповнювати ресурси гравців в залежності від результату сутички. Для початку кожен гравець відправляє випадкового громадянина населення, після чого визначається результат сутички та кожен гравець отримують свою долю ресурсів.

Потреба у такій функції є через те, що навіть у вирахуванню результату одної симуляції гри може виконуватись безмежна кількість сутичок, в залежності від введених користувачем даних. Також вже на цьому моменті видно, як пригодилось нам мати об’єкти гравців всередині конструктору класу **Simulation**.

```
34      def clash(self):
35          p1_strategy = self.player1.play_strategy()
36          p2_strategy = self.player2.play_strategy()
37
38          if p1_strategy == "hawk" and p2_strategy == "hawk":
39              # Both players pay tax, 50% chance of winning the new resources
40              if random.random() < 0.5:
41                  self.player1.resources += 1
42              else:
43                  self.player2.resources += 1
44              self.player1.resources -= 2
45              self.player2.resources -= 2
46
47          elif p1_strategy == "hawk" and p2_strategy == "dove":
48              # Player1's hawk wins and keeps 75% of resources
49              self.player1.resources += 3
50              self.player2.resources += 1
51
52          elif p1_strategy == "dove" and p2_strategy == "hawk":
53              # Player2's hawk wins and keeps 75% of resources
54              self.player1.resources += 1
55              self.player2.resources += 3
56
57          elif p1_strategy == "dove" and p2_strategy == "dove":
58              # Doves split the new resources equally
59              self.player1.resources += 1
60              self.player2.resources += 1
```

Рис 2.16 Функція clash() для задання правил гри “Яструб-Голуб”

Перш ніж написати алгоритм для вирахування кращої з двох стратегій гравців, цікавою думкою прийшла ще зробити коротку функцію **find_nash_equilibrium()**, яка дивлячись на результат n симуляцій вираховує найближчу рівновагу Неша серед стратегій гравців.

```
106     def find_nash_equilibrium(self):
107         if self.player1.resources >= self.goal_resources or self.player2.resources <= 0:
108             print("Nash Equilibrium: Player 1 plays Hawk, Player 2 plays Dove", "\n")
109         elif self.player2.resources >= self.goal_resources or self.player1.resources <= 0:
110             print("Nash Equilibrium: Player 1 plays Dove, Player 2 plays Hawk", "\n")
111         else:
112             print("Nash Equilibrium: Both players play Hawks and Doves 50/50", "\n")
```

Рис 2.17 Функція find_nash_equilibrium() для апроксимації найближчої рівноваги Неша

Далі необхідно написати сам алгоритм вирахування кращої з двох стратегій гравців. Назвемо його **simulate_win_lose_game()**. Для початку задаємо змінні **player1_wins** та **player2_wins**, що утримуватимуть кількість симуляцій, які переміг кожен з гравців. Далі у циклі забираємо у гравців всі старі ресурси та даємо їм початкову кількість перед новою симуляцією.

Після цього, у циклі, до поки один з гравців не дійшов бажаної кількості ресурсів, або кількість ресурсів противника не опустилась до нуля, ми викликаємо функцію **clash()**, яка щоразу змінює кількість ресурсів наших гравців. Після того, як одна з чотирьох умов була досягнута, програма покидає цикл і вираховує хто з гравців переміг та додає одну перемогу до змінної переможного гравця.

```
62      def simulate_win_lose_game(self):
63          player1_wins = 0
64          player2_wins = 0
65
66          for _ in range(self.num_simulations):
67              self.player1.resources = self.player1.default_resources
68              self.player2.resources = self.player2.default_resources
69              while (
70                  self.player1.resources < self.goal_resources
71                  and self.player2.resources < self.goal_resources
72                  and self.player1.resources > 0
73                  and self.player2.resources > 0
74              ):
75                  self.clash()
76              if (
77                  self.player1.resources >= self.goal_resources
78                  or self.player2.resources <= 0
79              ):
80                  # Player 1 wins
81                  player1_wins += 1
82              elif (
83                  self.player2.resources >= self.goal_resources
84                  or self.player1.resources <= 0
85              ):
86                  # Player 2 wins
87                  player2_wins += 1
```

Рис 2.18 Функція simulate_win_lose_game() для проведення одної симуляції гри

Все, що залишається – це підрахувати у процентах скільки переміг кожен з гравців, вивести це у термінал, і також викликати в кінці функцію **find_nash_equilibrium()**, яка додаткова опрацює ще результати гри.

```
89          player1_win_percentage = (player1_wins / self.num_simulations) * 100
90          player2_win_percentage = (player2_wins / self.num_simulations) * 100
91
92          print(
93              "Player 1 ({}, {}), Player 2 ({}, {})".format(
94                  self.player1.hawk_percentage,
95                  self.player1.dove_percentage,
96                  self.player2.hawk_percentage,
97                  self.player2.dove_percentage
98              )
99
100         )
101         print("Player 1 win percentage: {:.2f}".format(player1_win_percentage))
102         print("Player 2 win percentage: {:.2f}".format(player2_win_percentage), "\n")
103
104         self.find_nash_equilibrium()
```

Рис 2.19 Підрахування кількості виграшів гравців у процентах

І тепер ми маємо повне право малювати головне меню нашої гри. Меню будуємо навколо змінної **change_strategy**, яка утримуватиме прийняте користувачем рішення покинути мені для одної з опцій. На початку програма виводить стратегії кожного з гравців, після чого повертає значення по замовчуванню змінній **change_strategy**, та виводить меню на екран. Поки що у нас опцій є 3, а саме 1. Вирахувати у кого з гравців краща стратегія. 2. Поміняти стратегію гравцям. 3. Вийти. У випадок того, що користувач щось не те введе, буде пропонуватися ввести число **5**, щоб повторно відобразити меню.

```
142     def menu(self):
143         change_strategy = ""
144         while(True):
145             print("Player 1 ({}, {}), Player 2 ({}, {})".format(
146                 self.player1.hawk_percentage,
147                 self.player1.dove_percentage,
148                 self.player2.hawk_percentage,
149                 self.player2.dove_percentage))
150 
151             change_strategy = ""
152             print("\t\tMain menu\n1. Play Hawk-Dove\n2. Change strategy of players\n3. Exit")
153             first_attempt = True
154             while(change_strategy != "1" and change_strategy != "2" and change_strategy != "3" and change_strategy != "4"):
155                 if not first_attempt:
156                     print("Invalid input. Input 5 to display the menu again.")
157                 change_strategy = input("Which option would you like? (1 | 2 | 3):")
158                 if change_strategy == "5" and not first_attempt:
159                     print("\n\t\tMain menu\n1. Play Hawk-Dove\n2. Determine best strategy for both players",
160                           "\n3. Change strategy of players\n4. Exit")
161                 first_attempt = False
162 
163             if (change_strategy == "1"):
164                 print()
165                 self.simulate_win_lose_game()
166 
167             elif (change_strategy == "2"):
168                     print()
169                     self.input_strategies()
170 
171             elif (change_strategy == "3"):
172                 return
173 
174 simulation = Simulation()
175 simulation.menu()
```

Рис 2.20 Функція menu() для головного меню програми

Генеруємо об’єкт нашого класу **Simulation** та викликаємо **menu()**. Задаємо стратегії, початкову та бажану кінцеву кількість ресурсів, а також кількість симуляцій. Далі вибираємо опцію **1**, і наша програма вираховувала, що перший гравець переміг у **93.5%** симуляціях (симуляцій задано 200, тому **0.5%** допустимо), а другий гравець переміг у **6.5%** симуляціях. Далі вибираємо другу опцію, щоб поміняти стратегії персонажів, вводимо нові дані, і бачимо, що дані оновились на головному екрані. Врешті-решт, вибираємо третю опцію і покидаємо меню.

```
PS D:\Programs\Diplom> & C:/Users/Slavik/AppData/Local/Progra
Enter Player1 Hawk percentage ([0, 1]):0.9
Enter Player1 Dove percentage ([0, 1]):0.1
Enter Player2 Hawk percentage ([0, 1]):0.8
Enter Player2 Dove percentage ([0, 1]):0.2
Enter amount of starting resources (>0):50
Enter the win amount of resources (>start):1000
Enter the number of simulations for testing (>0):200

Player 1 (0.9, 0.1), Player 2 (0.8, 0.2)
                Main menu
1. Play Hawk-Dove
2. Change strategy of players
3. Exit
Which option would you like? (1 | 2 | 3):1

Player 1 (0.9, 0.1), Player 2 (0.8, 0.2)
Player 1 win percentage: 93.50
Player 2 win percentage: 6.50

Nash Equilibrium: Player 1 plays Dove, Player 2 plays Hawk

Player 1 (0.9, 0.1), Player 2 (0.8, 0.2)
                Main menu
1. Play Hawk-Dove
2. Change strategy of players
3. Exit
Which option would you like? (1 | 2 | 3):2

Enter Player1 Hawk percentage ([0, 1]):0
Enter Player1 Dove percentage ([0, 1]):1
Enter Player2 Hawk percentage ([0, 1]):1
Enter Player2 Dove percentage ([0, 1]):0
Enter amount of starting resources (>0):50
Enter the win amount of resources (>start):200
Enter the number of simulations for testing (>0):300

Player 1 (0.0, 1.0), Player 2 (1.0, 0.0)
                Main menu
1. Play Hawk-Dove
2. Change strategy of players
3. Exit
Which option would you like? (1 | 2 | 3):3
PS D:\Programs\Diplom>
```

Рис 2.21 Вивід програми при проведенні 200 симуляцій та зміни стратегій

Тепер можна зберегти поточні результати у нашу локальну репозиторію. Цього разу введемо **git status** перед викликом **git add prototype.py**, і бачимо, що Git Bash помітив, що вміст файлу було модифіковано, але команди його підготувати до затвердження дано не було.

```
Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   prototype.py

no changes added to commit (use "git add" and/or "git commit -a")

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git add prototype.py

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git commit -m "simulation_win_lose_game built"
[master ee92b68] simulation_win_lose_game built
 1 file changed, 83 insertions(+), 1 deletion(-)

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
```

Рис 2.22 Зміна статусу репозиторії у Git

2.6 Вирахування найкращої стратегії

Залишилось написати ще одну функцію для нашого проекту. Наш гравець вже має можливість перевірити котра зі стратегій буде краще справлятись проти іншої, проте згідно з теоретичними відомостями нам ще потрібно вміти визначати найкращу з усіх можливих стратегій під довільного опонента.

Пригадаємо, що звичний автоматизований спосіб для визначення найкращої стратегії проти конкретну стратегію противника – перебрати усі можливі стратегії і перевірити яка стабільно набирає найбільшу кількість ресурсів.

Перш за все необхідно написати альтернативу **simulate_win_lose_game()**, яка буде визначати кількість ресурсів, яку набрав гравець проти противника за n кількість симуляцій всередині одної гри. Для цього створюємо функцію **simulate_resources_game()**, яка у циклі з n симуляцій виконує n сутичок та повертає кількість ресурсів гравця, який числиться першим всередині об’єкту симуляції. Звучить трошки заплутано, але це зроблено для того, щоб ми мали можливість створити тимчасовий об’єкт симуляції, всередині якого першим гравцем буде одна з сотні можливих стратегій від (1,0) до (0.99, 0.01) до (0, 1).

```
142     def simulate_resources_game(self):
143         for _ in range(self.num_simulations):
144             self.player1.resources = self.player1.default_resources
145             self.player2.resources = self.player2.default_resources
146             for _ in range(self.num_simulations):
147                 self.clash()
148         return self.player1.resources
```

Рис 2.23 Функція simulate_resources_game() для підрахування кількості ресурсів протягом num_simulations кількості сутичок гравців

Для того, щоб перебрати усі можливі ситуації та визначити найкращу складаємо функцію **best_strategy()**. Перш за все створюємо параметри **current_hawk, current_dove, opponent_hawk, opponent_dove**, оскільки нам знадобиться шукати найкращу стратегію для обох гравців.

Далі створюємо тимчасову симуляцію, де **current_...** – гравець, для якого ми будемо шукати найкращу стратегію, **opponent_...** – гравець, під стратегію якого буде визначатись найкраща стратегія. Створюємо тимчасову змінну **current_average_resources**, яка утримуватиме результати 10 ігор по набиранню ресурсів проти противника. Після чого ділитиме свій вміст на 10 та виводити у термінал. Таким чином ми вирахували скільки в середньому набирає гравець **current_...** проти противника **opponent_...** , провівши 10 ігор та вирахувавши середню кількість ресурсів гравця **current_...** .

```
150     def best_strategy(self, current_hawk, current_dove, opponent_hawk, opponent_dove):
151         # Analyzing current strategy against opponent
152         current_average_resources = 0
153 
154         for _ in range(10):
155             temp_simulation = Simulation(current_hawk, current_dove, opponent_hawk, opponent_dove, self.player1.default_resources, 1000, 100)
156             current_average_resources += temp_simulation.simulate_resources_game()
157         current_average_resources /= 10
158         print('''The average resources using current strategy is ''' + str(current_average_resources))
```

Рис 2.24 Початок функції best_strategy() що вираховує середню кількість ресурсів з поточною стратегією гравця під противника

Переходимо до вираховування найкращої стратегії проти противника **opponent_...** . Для цього створюємо змінні **best_hawk**, **best_dove**, **best_average_resources**, які міститимуть найкраще співвідношення яструбів, голубів та середню кількість ресурсів. За замовчуванням значення присвоєні (1, 0), а цикл вже за нас опрацює усі можливі значення, віднімаючи у циклі змінну **i** від **best_hawk**, **best_dove** та записуючи у **temp_hawk**, **temp_dove**. Далі генерується тимчасова симуляція, де перший гравець – **temp_...**, а другий гравець **– opponent_...** . Все працює по схожому алгоритмі, як і сторінкою вище, тільки у випадку того, що стратегія гравця **temp_...** привела до кращої середньої кількості ресурсів, то оновлюється вміст **best_...** на значення **temp_...** . В кінці програми виводиться знайдена найкраща стратегія під противника **opponent_...**, саме співвідношення населення противника та яку найкращу стратегію нам вдалося знайти і з якою кількістю ресурсів. Далі ми повертаємо значення **best_hawk**, **best_dove**, бо мені здалося гарною думкою дати можливість користувачу після вирахування найкращої стратегії запропонувати поміняти стратегію гравцям об’єкту головної симуляції.

```
        # Computing best strategy against opponent
        best_hawk = 1.0
        best_dove = 0.0
        best_average_resources = 0

        for i in range(1, 102):
            temp_hawk = abs((1 - i) / 100)
            temp_dove = abs(1 - temp_hawk)
            temp_simulation = Simulation(temp_hawk, temp_dove, opponent_hawk, opponent_dove, self.player1.default_resources, 1000, 100)
            temp_average_resources = 0

            for _ in range(10):
                temp_average_resources += temp_simulation.simulate_resources_game()
            temp_average_resources /= 10

            if best_average_resources < temp_average_resources:
                best_average_resources = temp_average_resources
                best_hawk = round(temp_hawk, 2)
                best_dove = round(temp_dove, 2)

        print("The best mixed strategy against the opponent (" + str(float(opponent_hawk)) + ", " + str(float(opponent_dove)) + ") is ("
              + str(best_hawk) + ", " + str(best_dove) + ")")
        print('''The average resources using best strategy is ''' + str(best_average_resources), "\n")
        return best_hawk, best_dove
```

Рис 2.25 Продовження функції best_strategy() що вираховує найкращу з можливих стратегій гравця під противника

Все, що залишилось – це доробити головне меню, щоб воно пропонувало користувачу другу опцію. Міняємо порядок опцій у головному меню на 1. Визначити кращу з двох стратегій. 2. Знайти найкращу стратегію для обох гравців. 3. Поміняти стратегії гравцям. 4. Вийти. В результаті запускаємо **best_strategy()** для обох гравців, визначаємо найкращу стратегію для обох, зберігаючи ці дані, і пропонуємо користувачу поміняти поточні стратегії гравців на найкращі. Також додаємо цикл на випадок того, що користувач знов почне помилкові дані вводити.

```
    elif (change_strategy == "2"):
        print()
        print("Player1:")
        p1_best_hawk, p1_best_dove = self.best_strategy(self.player1.hawk_percentage, self.player1.dove_percentage,
                                                        self.player2.hawk_percentage, self.player2.dove_percentage)
        print("Player2:")
        p2_best_hawk, p2_best_dove = self.best_strategy(self.player2.hawk_percentage, self.player2.dove_percentage,
                                                        self.player1.hawk_percentage, self.player1.dove_percentage)
        print("1. Assign the best strategy to Player 1.\n2. Assign the best strategy to Player 2.",
              "\n3. Assign the best strategies to both players\n4.Return to main menu.")
        new_strategy = ""
        first_best_strategy_attempt = True

        while(new_strategy != "1" and new_strategy != "2" and new_strategy != "3" and new_strategy != "4"):
            if not first_best_strategy_attempt:
                print("Invalid input. Input 5 to display the options again.")
            new_strategy = input("Which option would you like? (1 | 2 | 3 | 4):")
            if new_strategy == "5" and not first_attempt:
                print("1. Assign the best strategy to Player 1.\n2. Assign the best strategy to Player 2.",
                      "\n3. Assign the best strategies to both players\n4.Return to main menu.")
            first_best_strategy_attempt = False

        if (new_strategy == "1"):
            self.initialize_simulation(p1_best_hawk, p1_best_dove, self.player2.hawk_percentage, self.player2.dove_percentage,
                                       self.player1.default_resources, self.goal_resources, self.num_simulations)
        elif (new_strategy == "2"):
            self.initialize_simulation(self.player1.hawk_percentage, self.player1.dove_percentage, p2_best_hawk, p2_best_dove,
                                       self.player1.default_resources, self.goal_resources, self.num_simulations)
        elif (new_strategy == "3"):
            self.initialize_simulation(p1_best_hawk, p1_best_dove, p2_best_hawk, p2_best_dove,
                                       self.player1.default_resources, self.goal_resources, self.num_simulations)
```

Рис 2.26 Модифікація функції menu() для підтримування best_strategy()

Перевіряємо працездатність програми. Задаємо стратегії (0.9, 0.1), (0.8, 0.2), визначаємо найкращу стратегію для цієї пари гравців і бачимо, що оскільки обидві стратегії є агресивними, де більшість населення – яструби, то найкраща стратегія під кожну з їхніх агресивних стратегій – це стратегія (0, 1). Звісно у випадку протилежному, де обидва гравці вибрали пасивну стратегію програма скаже, що можна брати агресивну стратегію, проте агресивний випадок є більш репрезентабельним. Цей випадок показує, що найкраща стабільна стратегія під виходить завжди пасивна (0, 1), що нагадує нам з теоретичних відомостей Рівновагу Неша (0, 1), (1, 0).

```
PS D:\Programs\Diplom> & C:/Users/Slavik/AppData/Local/Programs/Python/
Enter Player1 Hawk percentage ([0, 1]):0.9
Enter Player1 Dove percentage ([0, 1]):0.1
Enter Player2 Hawk percentage ([0, 1]):0.8
Enter Player2 Dove percentage ([0, 1]):0.2
Enter amount of starting resources (>0):200
Enter the win amount of resources (>start):1000
Enter the number of simulations for testing (>0):200

Player 1 (0.9, 0.1), Player 2 (0.8, 0.2)
                Main menu
1. Play Hawk-Dove
2. Determine best strategy for both players
3. Change strategy of players
4. Exit
Which option would you like? (1 | 2 | 3 | 4):2

Player1:
The average resources using current strategy is 154.5
The best mixed strategy against the opponent (0.8, 0.2) is (0.0, 1.0)
The average resources using best strategy is 300.0

Player2:
The average resources using current strategy is 127.9
The best mixed strategy against the opponent (0.9, 0.1) is (0.0, 1.0)
The average resources using best strategy is 300.0

1. Assign the best strategy to Player 1.
2. Assign the best strategy to Player 2.
3. Assign the best strategies to both players
4.Return to main menu.
Which option would you like? (1 | 2 | 3 | 4):YaroslavSvoren
Invalid input. Input 5 to display the options again.
Which option would you like? (1 | 2 | 3 | 4):4
```

Рис 2.27 Вивід програми при вирахуванні найкращої стратегії та хибному вводі

```
Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git add prototype.py

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
$ git commit -m "demo built"
On branch master
nothing to commit, working tree clean

Slavik@DESKTOP-H4FPGI3 MINGW64 /d/Programs/Diplom (master)
```

Рис 2.28 Зміна статусу репозиторії у Git

3. Графічний інтерфейс програми

3.1 Ознайомлення з tkinter

Як ми вже планували у розділі 2.4. з позиції користувача не має жодних переваг розбиратись в тому, як правильно задати значення у командному рядку, коли усі програми в його арсеналі зустрічають його графічним інтерфейсом. Спробуємо стати одною з таких програм. Для створення графічного інтерфейсу програми я вибрав **tkinter**, який ми вже коротко описали у розділі 2.3. .

Tkinter є модулем вбудованої бібліотеки Python, який надає інтерфейс для створення графічних інтерфейсів користувача (GUI). Це означає, що з допомогою **tkinter** можна створювати вікна, кнопки, поле введення, списки, меню та інші елементи, які дозволяють взаємодіяти з програмою за допомогою графічного інтерфейсу [12].

Tkinter базується на бібліотеці **Tk**, яка створюється у **Tcl** (Tool Command Language). Вона надає високорівневий доступ до функціональності **Tk**, дозволяючи зручно працювати з графічними елементами і здійснювати їх керування.

За допомогою **tkinter** можна створювати інтерфейси, які дозволяють користувачеві взаємодіяти з програмою шляхом натискання кнопок, введення тексту, вибору елементів зі списку та інших дій. **Tkinter** також підтримує можливість стилізації і налаштування графічного вигляду елементів інтерфейсу. Загалом, **tkinter** є потужним інструментом для створення графічних інтерфейсів у Python. Він дозволяє розробникам швидко створювати прості програми з графічним інтерфейсом, що полегшує взаємодію з користувачами та надає їм зручність використання.

3.2 Побудова графічного інтерфейсу

Останнє завдання, яке нам потрібно виконати – побудувати графічний інтерфейс нашій програмі на **tkinter**. Назвемо нашу фінальну програму **hawkdove.py**, створивши копію нашого **prototype.py** у директорію **Diplom**.

Для початку до нашого модуля **random** приєднується сам модуль **tkinter**, а також для зручності окремо створюємо **font** та **messagebox** для задавання власноручних розмірів шрифтів у **tkinter** та побудови вікон, що попереджують користувача про помилку у введених даних. Клас **Player** ніяких змін не потребував для сумісності з **tkinter**.

hawkdove.py > ...

```python
import random
import tkinter as tk
from tkinter import messagebox
from tkinter import font

class Player:
    def __init__(self, hawk_percentage, dove_percentage, start_resources):
        self.hawk_percentage = hawk_percentage
        self.dove_percentage = dove_percentage
        self.resources = start_resources
        self.default_resources = start_resources
        self.wins = 0

    def play_strategy(self):
        r = random.random()
        if r < self.hawk_percentage:
            return "hawk"
        else:
            return "dove"
```

Рис 3.1 Клас Player

Перше, що повинно змінитись з прототипу – вікно введення даних. Почати варто написання функції **validate_input()**, яка буде перевіряти чи добрі дані ввів користувач. Далі напишемо саме вікно для введення даних. Оскільки **tkinter** приймає дані не у типі **String**, а **StringVar**, конвертувати дані потрібно через **StringVar.get()**, що дає нам тип **String**.

Після цього описуємо звичну нам перевірку введених даних на коректність, і у випадку якщо користувач ввів хибні дані – ми можемо відобразити спеціальне вікно про помилку командою **messagebox.showerror()**.

```
62      def validate_input(self):
63          player1_hawk = float(self.player1_hawk_percentage.get())
64          player1_dove = float(self.player1_dove_percentage.get())
65          player2_hawk = float(self.player2_hawk_percentage.get())
66          player2_dove = float(self.player2_dove_percentage.get())
67          start_resources = int(self.start_resources.get())
68          goal_resources = int(self.goal_resources.get())
69          num_simulations = int(self.num_simulations.get())
70
71          if (
72                  0 <= player1_hawk <= 1
73                  and 0 <= player1_dove <= 1
74                  and 0 <= player2_hawk <= 1
75                  and 0 <= player2_dove <= 1
76                  and start_resources > 0
77                  and goal_resources > start_resources
78                  and num_simulations > 0
79                  and player1_hawk + player1_dove == 1
80                  and player2_hawk + player2_dove == 1
81          ):
82              self.show_main_menu()
83          else:
84              messagebox.showerror("Invalid Input", "Please enter valid input values.")
85              self.create_widgets()
```

Рис 3.2 Функція validate_input() для перевірки вводу користувача

Тепер можна перейти до самого вікна введення даних користувачем. Використовуючи **tkinter.font** створюємо шрифт розміром 1.5 від розміру за замовчування та записуємо у змінну **custom_font**. Далі за допомогою **tk.Label()** наносимо звичайний текст для нашого вікна. Після кожного тексту йде вікно **tk.Entry()** для введення інформації, вміст з яких надійде відповідним змінним у конструкторі класу **Simulation**. За допомогою **.grid()** будуємо сітку (матрицю) нашого вікна. В самому кінці нас зустрічає **tk.Button()**, тобто кнопка з текстом **“Start Simulation”**, нажавши на яку виникає виклик функції **validate_input()**, і якщо дані були введено користувачем коректні, то з **validate_input()** вже буде викликано вікно головного меню та закрито вікно вводу даних користувачем.

```python
class Simulation:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Hawk-Dove Simulation")
        self.menu_root = None

        self.player1_hawk_percentage = tk.StringVar()
        self.player1_dove_percentage = tk.StringVar()
        self.player2_hawk_percentage = tk.StringVar()
        self.player2_dove_percentage = tk.StringVar()
        self.start_resources = tk.StringVar()
        self.goal_resources = tk.StringVar()
        self.num_simulations = tk.StringVar()

        self.create_widgets()
```

Рис 3.3 Конструктор класу Simulation

```python
def create_widgets(self):
    custom_font = font.Font(size=int(1.5 * font.nametofont("TkDefaultFont").actual()["size"]))
    tk.Label(self.root, text="Player 1 Hawk percentage ([0, 1]):", font = custom_font).grid(row=0, column=0, padx=10, pady=5,)
    tk.Entry(self.root, textvariable=self.player1_hawk_percentage, font = custom_font).grid(row=0, column=1)

    tk.Label(self.root, text="Player 1 Dove percentage ([0, 1]):", font = custom_font).grid(row=1, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.player1_dove_percentage, font = custom_font).grid(row=1, column=1)

    tk.Label(self.root, text="Player 2 Hawk percentage ([0, 1]):", font = custom_font).grid(row=2, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.player2_hawk_percentage, font = custom_font).grid(row=2, column=1)

    tk.Label(self.root, text="Player 2 Dove percentage ([0, 1]):", font = custom_font).grid(row=3, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.player2_dove_percentage, font = custom_font).grid(row=3, column=1)

    tk.Label(self.root, text="Amount of starting resources (>0):", font = custom_font).grid(row=4, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.start_resources, font = custom_font).grid(row=4, column=1)

    tk.Label(self.root, text="Win amount of resources (>start):", font = custom_font).grid(row=5, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.goal_resources, font = custom_font).grid(row=5, column=1)

    tk.Label(self.root, text="Number of simulations for testing (>0):", font = custom_font).grid(row=6, column=0, padx=10, pady=5)
    tk.Entry(self.root, textvariable=self.num_simulations, font = custom_font).grid(row=6, column=1)

    tk.Button(self.root, text="Start Simulation", font = custom_font, command=self.validate_input).grid(row=7, column=0, columnspan=2, pady=10)
```

Рис 3.4 Імплементація вікна для вводу користувачем даних

Забігаючи наперед ми можемо вже порівняти як виглядає наше вікно введення даних, нагадуючи структуру з функції **create_widgets()**.

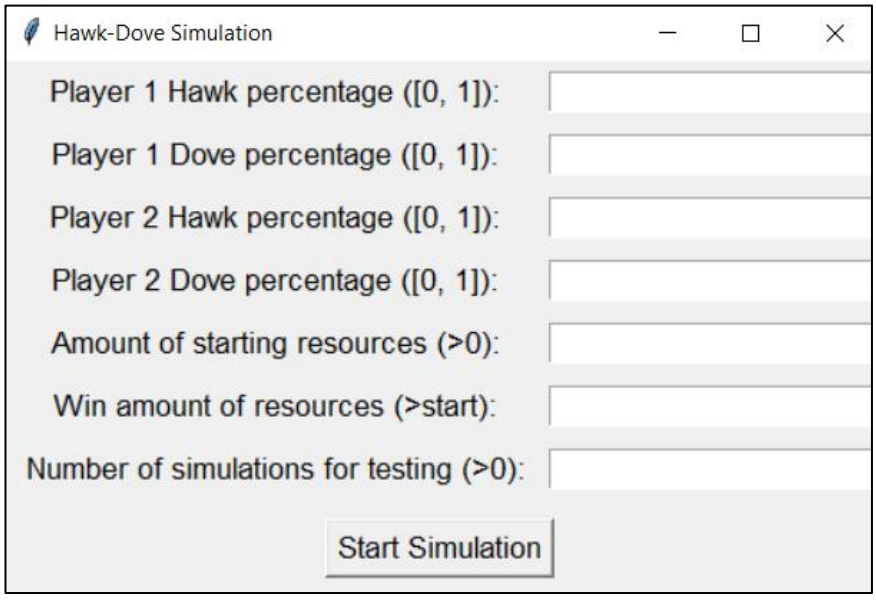


Рис 3.5 Вікно для вводу користувачем даних

Згідно з послідовністю програми, при умові, що користувач ввів коректні дані, функція **validate_input()** повинна викликати функцію, що знищить вікно введення даних та відобразить вікно головного меню. Для цього будуємо функцію **show_main_menu()**. Оскільки у конструкторі класу **Simulation** ми помістили вказівник на вікно у змінній **root**, ми можемо знищити це вікно з іншої функції класу за допомогою **self.root.destroy()**. Команда **tk.Tk().destroy()** знищує об’єкт вікна **tkinter**.

Після цього ініціалізуємо вікно головного меню так само, як і **root** у конструкторі, командою **tk.Tk()**. Даємо назву вікна командою **menu_root.title()**, створюємо два розміри шрифтів, один з яких у 2 рази більший за стандартний, інший у 1.5 . Малюємо великий текст з підписом того, що ми знаходимось у головному меню з розміром шрифта x2, після чого малюємо 4 кнопки нашого меню з розміром шрифта x1.5 . Всередині **tk.Button()** вказуємо у параметр **command=** яка функція буде викликана, якщо натиснути цю кнопку. Наперед готуємо пусті функції для майбутнього виклику кожної з функцій нашого проекту.

```
89  def show_main_menu(self):
90      self.root.destroy()
91
92      self.menu_root = tk.Tk()
93      self.menu_root.title("Main Menu")
94      custom_font = font.Font(size=3 * font.nametofont("TkDefaultFont").actual()["size"])
95      custom_font2 = font.Font(size=int(1.5 * font.nametofont("TkDefaultFont").actual()["size"]))
96      tk.Label(self.menu_root, text="Main Menu", font=custom_font).pack(pady=10)
97      tk.Button(self.menu_root, text="1. Play Hawk-Dove", font=custom_font2, command=self.show_simulate_win_lose_game).pack(pady=5)
98      tk.Button(self.menu_root, text="2. Calculate Best Strategies",font=custom_font2, command=self.calculate_best_strategies).pack(pady=5)
99      tk.Button(self.menu_root, text="3. Manually Set Strategies", font=custom_font2, command=self.change_strategies).pack(pady=5)
100     tk.Button(self.menu_root, text="4. Exit", font=custom_font2, command=self.exit_simulation).pack(pady=5)
101
102     self.menu_root.mainloop()
```

Рис 3.6 Функція show_main_menu() для відображення головного меню програми

Перша зміна, якої потребувала функція знаходження кращої стратегії серед гри двох гравців – це спосіб ініціалізації гравців, оскільки виникли деякі проблеми з сумісністю створення з **tkinter**. В результаті було знайдено рішення, записуючи введені користувачем дані у конструктор у типі **StringVar**, після чого функції всередині класу їх конвертуватимуть у числові.

```
104     def show_simulate_win_lose_game(self):
105         try:
106             player1_hawk = float(self.player1_hawk_percentage.get())
107             player1_dove = float(self.player1_dove_percentage.get())
108             player2_hawk = float(self.player2_hawk_percentage.get())
109             player2_dove = float(self.player2_dove_percentage.get())
110             start_resources = int(self.start_resources.get())
111             goal_resources = int(self.goal_resources.get())
112             num_simulations = int(self.num_simulations.get())
113 
114             self.player1 = Player(player1_hawk, player1_dove, start_resources)
115             self.player2 = Player(player2_hawk, player2_dove, start_resources)
```

Рис 3.7 Початок функції show_simulate_win_lose_game(), що ініціалізує усі необхідні змінні для роботи алгоритму

В кінці функції **show_simulate_win_lose_game()** також необхідно було змінити спосіб виведення результатів роботи, в результаті чого було дано назву вікну “Simulation Results”, сформовано змінну шрифту x1.5 розміру, після чого виводився знайомий нам текст про процентне співвідношення перемог гравців.

```
175             result_root = tk.Tk()
176             result_root.title("Simulation Results")
177 
178             custom_font = font.Font(size=int(1.5 * font.nametofont("TkDefaultFont").actual()["size"]))
179 
180             label = tk.Label(result_root, text=f"Player 1 ({self.player1.hawk_percentage}, {self.player1.dove_percentage})" +
181                              f", Player 2 ({self.player2.hawk_percentage}, {self.player2.dove_percentage})", font = custom_font)
182             label.pack()
183 
184             tk.Label(result_root, text=f"Player 1 Win Percentage: {player1_win_percentage:.2f}%", font = custom_font).pack(pady=5)
185             tk.Label(result_root, text=f"Player 2 Win Percentage: {player2_win_percentage:.2f}%", font = custom_font).pack(pady=5)
186 
187             tk.Button(result_root, text="Back to Main Menu", font = custom_font, command=result_root.destroy).pack(pady=10)
188 
189             result_root.mainloop()
```

Рис 3.8 Імплементація вікна виводу результатів симуляцій ігор гравців

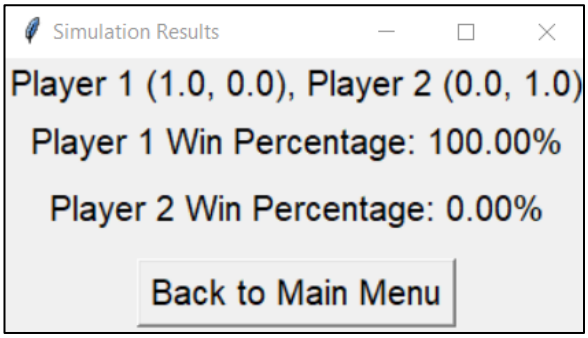


Рис 3.9 Вікно виводу результатів симуляцій ігор гравців

Переходимо до другої кнопки головного меню, яка відповідає за вказівку про вирахування найкращої стратегії під кожного з гравців. Як вже було згадано раніше, виникла проблема суміжності об’єктів з **tkinter**, в результаті чого довелось ініціалізувати об’єкти гравців всередині функції, яка з ними працює, тому функція **simulate_resources_game()** була перероблена на приймання об’єктів гравців, як параметрів. Сам алгоритм вирахування функції звісно залишився ідентичним з версії цієї функції з прототипу.

```
226         def simulate_resources_game(current_player:Player, opponent_player:Player, num_simulations):
227             for _ in range(num_simulations):
228                 current_player.resources = current_player.default_resources
229                 opponent_player.resources = opponent_player.default_resources
230                 for _ in range(num_simulations):
231                     clash(current_player, opponent_player)
232             return current_player.resources
```

Рис 3.10 Функція simulate_resources_game() для підрахування кількості ресурсів протягом num_simulations кількості сутичок гравців

Сама функція **best_strategy()** теж потребувала приймання об’єктів гравців, як параметрів. Створюванням цих об’єктів для обчислювання найкращої стратегії займалась головна стратегія для другої кнопкп головного меню – **show_best_strategy()**, в той час як **best_strategy()** та **simulate_resources_game()** приймали об’єкти від неї. Логіка функції **best_strategy()** змінена не була, спочатку вивід середньої кількості ресурсів гравця **current_...**, після чого знаходилась найкраща стратегія, яка записувалась у **best_...** та виводилась на екран.

```
234         def best_strategy(current_player:Player, opponent_player:Player, num_simulations, result_root):
235             #     Analyzing current strategy against opponent
236             current_average_resources = 0
237 
238             for _ in range(10):
239                 current_average_resources += simulate_resources_game(current_player, opponent_player, num_simulations)
240             current_average_resources /= 10
241 
242             custom_font = font.Font(size=int(1.5 * font.nametofont("TkDefaultFont").actual()["size"]))
243 
244             current_strategy_label = tk.Label(result_root, text="The average resources using the current strategy is " +
245                                               str(current_average_resources), font = custom_font)
246             current_strategy_label.pack(pady=5)
```

Рис 3.11 Початок функції best_strategy() для визначення та відображення середньої кількості ресурсів гравця при поточній стратегії

```
    #     Computing best strategy against opponent
    best_hawk = 1.0
    best_dove = 0.0
    best_average_resources = 0

    for i in range(1, 102):
        temp_hawk = abs((1 - i) / 100)
        temp_dove = abs(1 - temp_hawk)
        temp_player = Player(temp_hawk, temp_dove, current_player.default_resources)
        temp_average_resources = 0

        for _ in range(10):
            temp_average_resources += simulate_resources_game(temp_player, opponent_player, num_simulations)
        temp_average_resources /= 10

        if best_average_resources < temp_average_resources:
            best_average_resources = temp_average_resources
            best_hawk = round(temp_hawk, 2)
            best_dove = round(temp_dove, 2)

    best_strategy_label = tk.Label(result_root, text="The best stable mixed strategy against the opponent (" +
                                   str(float(opponent_player.hawk_percentage)) + ", " + str(float(opponent_player.dove_percentage)) +
                                   ") is ("  + str(best_hawk) + ", " + str(best_dove) + ")", font = custom_font)
    best_strategy_label.pack(pady=5)

    best_strategy_resources_label = tk.Label(result_root, text=("The average resources using the best strategy is " +
                                                                str(best_average_resources)), font = custom_font)
    best_strategy_resources_label.pack(pady=5)
```

Рис 3.12 Продовження функції best_strategy() для визначення та відображення найкращої з можливих стратегій гравця під противника

Залишилось оформити тільки третю та четверту кнопку, які відповідали за зміну співвідношень гравців та вихід. Можна було підмітити протягом цих скріншотів, що весь код знаходиться на один **TAB** далі чим повинен. Це зроблено навмисно, оскільки програма працює у циклі while(), вийти з якого можна тільки нажавши четверту кнопку у головному меню. Якщо нажати третю кнопку, то програма натомість знищує поточний об’єкт симуляції, і створює новий. Таке рішення було прийнято через проблему суміжності об’єктів з **tkinter**, і варто підмітити що жодної підозри у користувача виникнути не може, тому що нажавши третю кнопку у головному меню, тобто кнопку зміни співвідношень персонажів, зустрічає користувача те саме стартове вікно, що зустрічало на початку запуску програми. Глобальна змінна **exit** була використана для перевірки чи користувач натиснув четверту кнопку, і якщо так, тоді припинявся безкінечний цикл і програма завершувала роботу.

```
6    exit = False
```

Рис 3.13 Змінна для ролі тригера виходу з гри програми

```
316     def change_strategies(self):
317         self.menu_root.destroy()
318 
319     def exit_simulation(self):
320         global exit
321         exit = True
322         self.menu_root.destroy()
```

Рис 3.14 Функції для перезапуску програми та виходу з програми

Врешті-решт, можна поглянути на результат роботи.

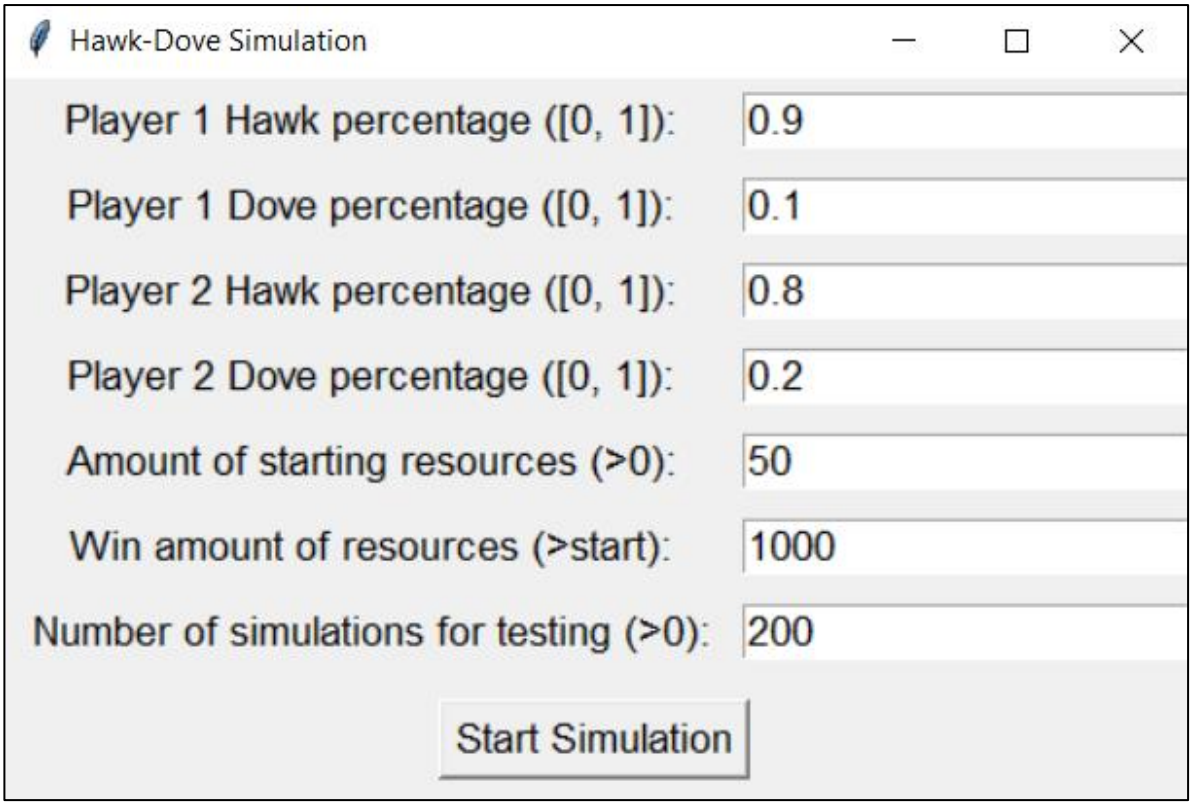


Рис 3.15 Приклад вводу коректних даних користувачем

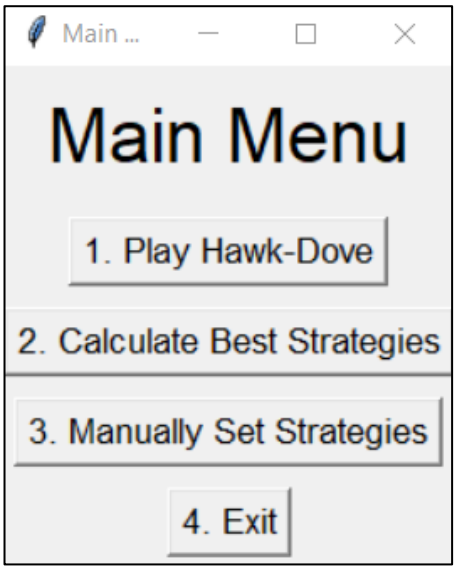


Рис 3.16 Головне меню програми

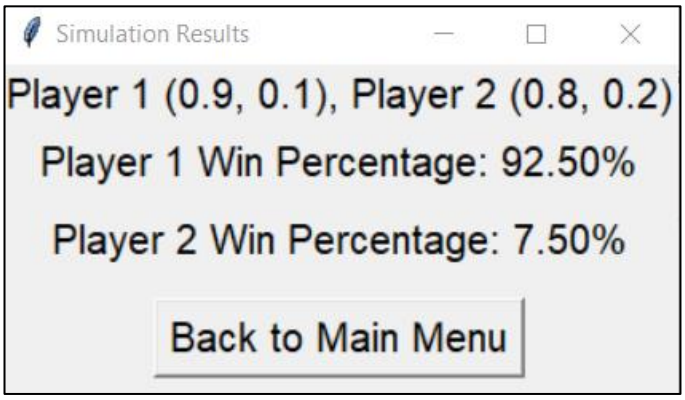


Рис 3.17 Результат виграшів гравців 200 симуляцій ігор “Яструб-Голуб”

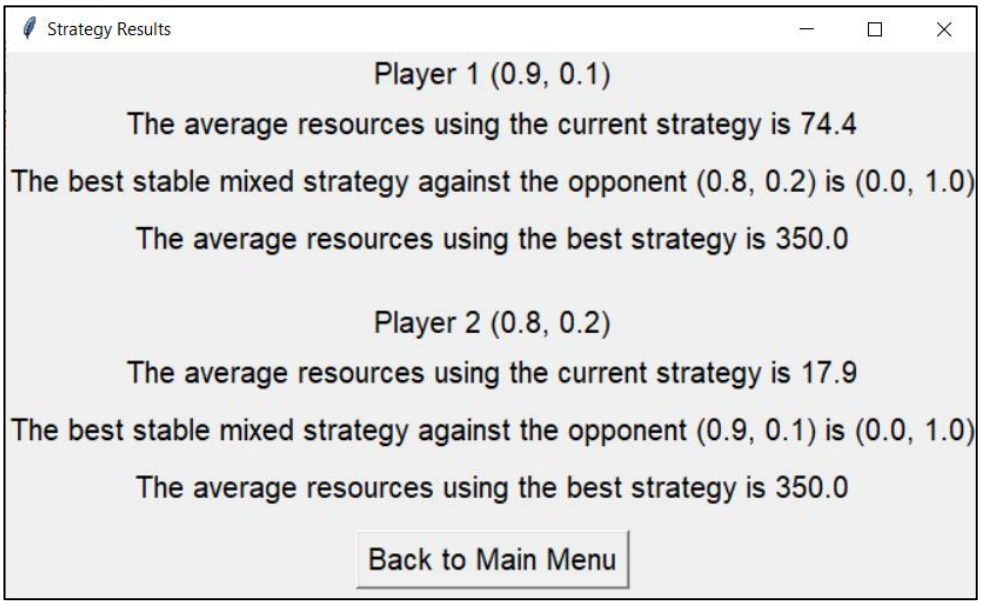


Рис 3.18 Результат вирахування найкращої стратегії для обох гравців шляхом порівняння найкращої з можливих середньої кількості отриманих ресурсів а також відображення середньої кількості ресурсів при поточній стратегії

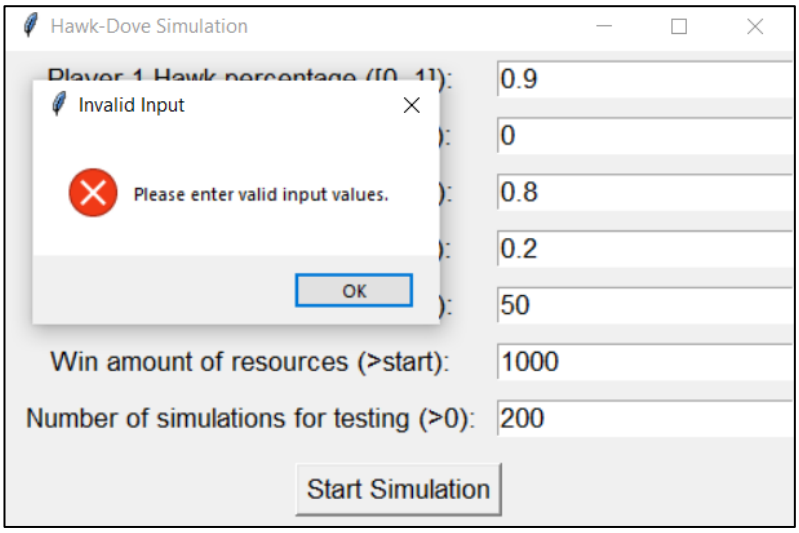


Рис 3.19 Екран повідомлення про помилку у випадку неправильного заданого співвідношення голубів та яструбів (сума співвідношення не є рівна 1).

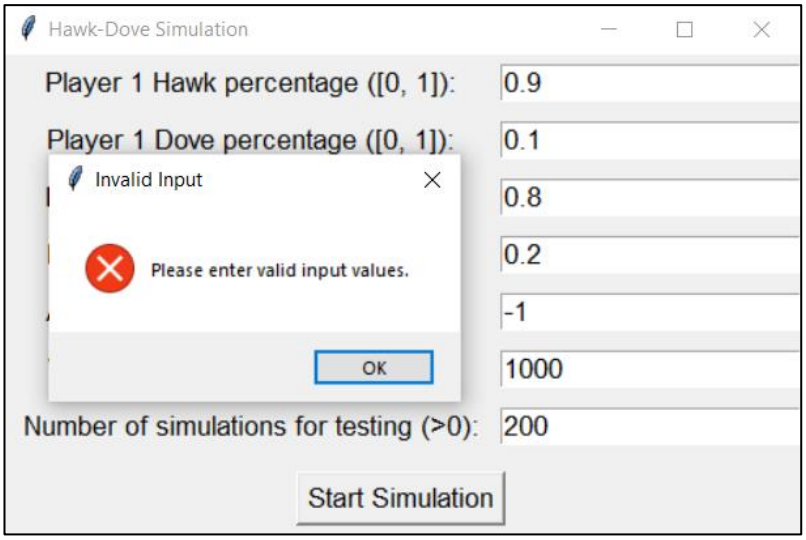


Рис 3.20 Екран повідомлення про помилку у випадку неправильного вказання початкової кількості ресурсів гравців (від’ємна кількість початкових ресурсів)

ВИСНОВОК

Протягом написання дипломної роботи було розглянуто теоретичні поняття еволюційних ігор, в особливості еволюційної гри “Яструб-Голуб”, а також поняття рівноваги Неша та поняття Еволюційно-Стабільної стратегії, у рамках якої було доведено єдину симетричну рівновагу Неша (0.5, 0,5).

Було розроблено програму для симуляції гри “Яструб-Голуб” спочатку у терміналі для перевірки логіки та оформлення програми для графічного інтерфейсу. Також було розроблено автоматизований алгоритм визначення найкращої з усіх можливих стратегій під стратегію противника. Після цього було програму з підтримкою графічного інтерфейсу, написаного за допомогою модулю для Python під назвою tkinter. В межах цього модулю було перенесено усю логіку програми для симуляції гри “Яструб-Голуб”, включно з визначенням найкращої з усіх можливих стратегій та зміни характеристик гравців.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hans Peters – Game Theory: A Multi-Leveled Approach
2. Tim Rees – An Introduction to Evolutionary Game Theory https://www.cs.ubc.ca/~kevinlb/teaching/cs532a%20-%202004-5/Class%20projects/Tim.pdf
3. Harold W.Kuhn Lectures on the Theory of Games / HaroldW. Kuhn – Princeton and Oxford: Published by Princeton University Press, 2003
4. Wikipedia – Bimatrix Game https://en.wikipedia.org/wiki/Bimatrix_game
5. Wikipedia – Nash Equilibrium https://en.wikipedia.org/wiki/Nash_equilibrium
6. Ashley Hodgson – Game Theory/Nash Equilibrium playlist https://www.youtube.com/watch?v=SOVdLjnBHQw https://www.youtube.com/watch?v=-RiZiTz2hWA https://www.youtube.com/watch?v=S49oKtISkqI
7. Erich Prisner Game Theory Through Examples / Erich Prisner – Published and Distributed by The Mathematical Association of America, 2014
8. Wikipedia – Evolutionary Stable Strategy https://en.wikipedia.org/wiki/Evolutionarily_stable_strategy
9. Veritasium – Evolutionarily Stable Strategies ft. Richard Dawkins https://www.youtube.com/watch?v=mUxt--mMjwA
10. Programming with Mosh – Git Tutorial for Beginners: Learn Git in 1 Hour https://www.youtube.com/watch?v=8JJ101D3knE
11. Tech with Tim – Git Tutorial for Beginners - Git & GitHub Fundamentals https://www.youtube.com/watch?v=DVRQoVRzMIY
12. FreeCodeCamp.org – Tkinter Course - Create Graphic User Interfaces in Python Tutorial https://www.youtube.com/watch?v=YXPyB4XeYLA