УДК 004.8 http://doi.org/10.15407/pp2023.04.076

Я. В. Омельяненко

МОДЕЛЮВАННЯ АВТОНОМНОГО ПРОХОДЖЕННЯ ЛАБІРИНТУ З ВИКОРИСТАННЯМ АЛГОРИТМУ NEAT

Автономне проходження лабіринту є класичним завданням прикладної математики, що стосується галузі автономної навігації. У цій роботі ми розглянемо, як можна використовувати методи нейроеволюції для вирішення задач проходження лабіринту. Також тут пояснимо, як визначити функцію пристосованості з використанням оцінки пристосованості агента-навігатора, розрахованої як похідної від відстані агента до кінцевої мети. Ми деталізуємо основи навчання автономного навігаційного агента з використанням методів нейроеволюції і на їх базі далі зможемо створювати більш розвинутий вирішувач для лабіринтів. Під час роботи здійснено розробку симулятора автономного робота, керованого штучною нейронною мережею, продукованою за допомогою алгоритму NEAT. Також розроблені нові методи візуалізації, які полегшують розуміння результатів виконання алгоритму. Всі розробки здійсненні з використанням мови програмування Python.

Ключові слова: генетичні алгоритми, нейроеволюція наростаючих топологій, автономне проходження лабіринту, NEAT, симулятор автономного робота.

Вступ

Завдання проходження лабіринту є класичною проблемою прикладної математики, яка тісно пов'язана зі створенням автономних агентів навігації, здатних знайти шлях у неоднозначних середовищах [1]. Навколишнє середовище у вигляді лабіринту – класична ілюстрація цілого класу проблем, які мають оманливий ландшафт пристосованості [2, 3, 8]. Це означає, що цілеорієнтована функція пристосованості (goal-oriented fitness function) може мати круті градієнти показників пристосованості в глухих кутах лабіринту, близьких до кінцевої мети. Такі зони лабіринту стають локальними оптимумами для алгоритмів пошуку на основі близькості до мети, які можуть сходитися в цих зонах. Коли алгоритм пошуку застряє у такому оманливому локальному оптимумі, він не може знайти адекватного агента, здатного пройти лабіринт.

На рис. 1 зображено двомірний лабіринт, в якому затемнені локально оптимальні глухі кути.

Конфігурація лабіринту на цьому рисунку демонструє ландшафт оманливих показників пристосованості, зосереджених у локально оптимальних глухих кутах (помічених як зафарбовані сегменти). Агент-вирішувач лабіринту, що переміщається від початкової точки (нижнє коло) до точки виходу (верхнє коло) та навчений на основі критерію близькості до мети, буде схильний застрягати в локальних глухих кутах. Крім того, подібна оманлива оцінка пристосованості може завадити алгоритму навчання на основі близькості до мети знайти успішний вирішувач лабіринтів.

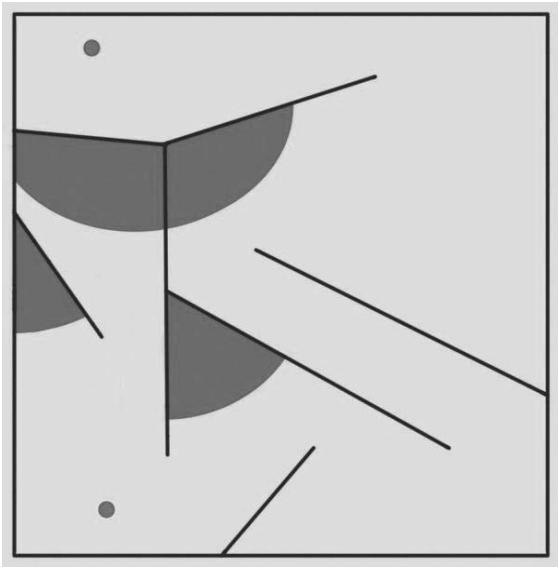

Рис. 1. Двовимірний лабіринт із локально оптимальними глухими кутами (затемнені)

У цій статті ми розглянемо, як алгоритм нейроеволюції NEAT [4, 7, 9] може бути використаний для навчання контролюючої штучної нейронної мережі (ШНМ), яка дасть змогу агенту-вирішувачу пройти лабіринт. Крім того,

будуть розглянуті нові методи, розроблені для візуалізації роботи алгоритму, що дозволяють наочно оцінювати якість його роботи. Розглянуті методи навчання контролюючої ШНМ можуть бути використані для широкого кола задач, пов'язаних із навчанням роботизованих агентів.

1. Огляд аналогів

Як було зазначено вище, задача проходження лабіринту є класичною проблемою прикладної математики, яка використовується для бенчмаркінгу навігаційних алгоритмів.

Для вирішення подібних задач у [5] автори пропонують використовувати еволюційні алгоритми, що фокусуються на навчанні агентів-вирішувачів за принципом максимальної розбіжності знайдених рішень. Але водночас поштовх до розбіжності рішень балансується цілеорієнтованою функцією для звуження простору пошуку. Як результат, продукується процес навчання під назвою *різноманітність якості* (quality diversity, QD). Відомими представниками цього класу алгоритмів є *пошук новизни з локальною конкуренцією* (novelty search with local competition, NSLC) та *багатовимірний архів фенотипових еліт* (Multi-dimensional Archive of Phenotypic Elites, MAPE).

Інший підхід для вирішення задачі проходження лабіринту представлений у [6], де автори пропонують використовувати принципи необмеженої еволюції, яка базується на коеволюції популяцій агентів-вирішувачів та популяцій ШНМ продукуючих конфігурації лабіринтів. Ця ідея базується на засадах збереження *мінімального критерію коеволюції* (minimal criterion coevolution, MCC). Разом з тим мінімальними критеріями, обмежуючими репродуктивну здатність організмів у популяції, є наступне: кожен лабіринт має бути розв'язаний принаймні одним агентом-вирішувачем та кожен агент-вирішувач має розв'язати принаймні один лабіринт. Кожен організм з обох родоводів, що відповідає цьому обмеженню, може розмножуватись. Автори показали, шо цей метод може бути ефективно використаний для пошуку універсальних агентів-вирішувачів.

Недоліками розглянутих підходів є необхідність у значних обчислювальних потужностях, а також тривалий час пошуку рішень.

Стаття пропонує алгоритм, який дозволяє навчати агента, що вирішує задачу лабіринту, за допомогою алгоритму NEAT та мови програмування Python. Розроблений алгоритм навчання може працювати на звичайному комп'ютері та швидко знаходити рішення.

У наступному розділі ми розглянемо основні особливості алгоритму NEAT.

2. Особливості алгоритму NEAT

Найважливішою особливістю алгоритму NEAT, яка визначає його потенціал, є здатність розвивати топологію ШНМ під час процесу навчання.

Важливість поступового нарощування складності топології ШНМ стає очевидною, якщо розглянути, як це реалізовано в алгоритмі NEAT.

Досліджуючи простір пошуку рішення, алгоритм NEAT починає з найпростішої базової топології мережі, яка включає лише вхідні та вихідні вузли ШНМ розв'язувача. Після цього з кожною епохою еволюції топологія вирішувачів ускладнюється, та найскладніші багатовимірні топології оцінюються лише на завершальних етапах еволюційного процесу. До того ж, найбільш підходящі топологічні структури, знайдені під час еволюції, зберігаються алгоритмом NEAT через усі покоління популяції та є предметом наступних оцінок придатності в майбутніх поколіннях. Тобто в процесі еволюції виживуть лише корисні топологічні структури, складність яких завжди виправдана цільовою функцією.

Як бачимо, за рахунок *наростаючого ускладнення топології ШНМ* досягається значне підвищення продуктивності алгоритму NEAT порівнянно з іншими еволюційними алгоритмами. Це досягається за рахунок значного звуження простору пошуку рішення під час навчання з одночасним збереженням різноманітності популяції рішень.

Різноманітність популяції рішень досягається за рахунок іншої суттєвої осо-

бливості алгоритму – *видоутворення* (speciation). Видоутворення дозволяє зберігати корисні мутації за рахунок обмеження репродуктивної конкуренції рамками окремої видової ніші у популяції. Розподіл організмів на види відбувається відповідно до генетичної відстані, тобто подібності між геномами організмів. Як результат, у процесі видоутворення захищаються потенційно корисні інновації та зберігається різноманітність популяції. Це відбувається за рахунок запобігання ситуації, коли найефективніші на даний час рішення витісняють всі інші рішення в межах популяції в цілому. Іншими словами, видоутворення зменшує негативний еволюційний тиск на молоді організми, які були щойно утворенні, та мають більш складну топологію за рахунок додавання нового вузла чи зв'язку між існуючими вузлами. Подібне ускладнення топології може призвести до тимчасового зниження придатності молодого організму. Але водночас подібна інновація може ввести новий вектор для дослідження простору пошуку рішень, що зрештою приведе до винайдення ефективного остаточного рішення у майбутніх поколіннях.

Пошук інновацій разом із запобіганням зупинці у локальних мінімумах цільової функції досягається за рахунок застосування *оператора мутації* до хромосом організмів під час репродукції. Цей оператор змінює один чи декілька генів у геномі організму відповідно до ймовірності мутації, що задається експериментатором як гіперпараметер. За рахунок випадкових змін, що вносяться у геном розв'язувача, досягається збереження генетичної різноманітності популяції та розширення простору пошуку можливих рішень. Оператор мутації є спільною рисою майже всіх генетичних алгоритмів. Але відмінною рисою реалізації його алгоритмом NEAT є те, що змінюються не лише вагові коефіцієнти зв'язків між вузлами ШНМ, а й сама структура топології ШНМ.

Для збереження інновацій та оптимізації обчислень упродовж кросинговеру між організмами популяції, алгоритм NEAT вводить поняття *історичних позначок* (innovation number). Це є однією з найвидатніших особливостей алгоритму NEAT, яка дозволяє оптимально вирішувати проблему пошуку перетину між геномами зі схожими топологіями без потреби в складних обчисленнях на основі графів.

В основу еволюційного процесу під керуванням алгоритму NEAT покладено процес *кросинговеру* (рекомбінації) геномів під час репродукції наприкінці кожної епохи еволюції. Саме на цьому етапі набувають значення історичні позначки, що дозволяють ефективно ідентифікувати гени, які перекриваються (overlapping/matching), а також роз'єднані (disjoint) та надлишкові (excess) гени. Гени, що перекриваються, мають однакові історичні позначки в геномах обох батьків і кодують однакову топологічну структуру незалежно від значень інших параметрів (коефіцієнт ваги з'єднання тощо). В процесі кросинговеру гени, що перекриваються, вибираються випадково у кожного із батьків для наслідування нащадками. Разом з тим, гени, які представлені лише у геномі одного із батьків розрізнюються за положенням їхніх історичних позначок у геномі іншого батька. Якщо історична позначка гена знаходиться в межах діапазону історичних позначок іншого з батьків, то він буде належати до роз'єднаних (disjoint), а якщо виходить за межі - надлишкових (excess) генів. Під час кросинговеру роз'єднані та/або надлишкові гени вибираються випадково від найефективнішого із батьків, або випадково від кожного із батьків (залежно від типу кросинговеру).

Згадані особливості алгоритму NEAT роблять його одним із найбільш вживаних та потужних з-поміж сімейства еволюційних алгоритмів, що дозволяють використовувати його для розв'язання широкого кола прикладних задач. Далі ми перейдемо до розгляду застосування алгоритму NEAT для пошуку ефективного рішення проблеми навігації у лабіринті.

3. Середовище моделювання задачі лабіринту

У рамках цієї статті було розроблено *новітнє програмне забезпечення* для моделювання задачі проходження лабірин-

ту мовою програмування *Python*. Воно складається з трьох основних компонентів, які реалізовані у вигляді окремих класів:

– *Agent* – клас, який зберігає інформацію, пов'язану з агентом навігатора лабіринту, задіяного в симуляції.

– *AgentRecordStore* - клас, який управляє зберіганням записів, що належать до оцінок усіх вирішальних агентів під час еволюційного процесу. Зібрані записи можна використовувати для аналізу еволюційного процесу після його завершення.

– *MazeEnvironment* – клас, який містить інформацію про середовище моделювання лабіринту. Цей клас також надає методи, які керують середовищем моделювання, керують положенням вирішального агента, виявляють зіткнення зі стінами лабіринту та генерують вхідні дані для датчиків агента.

Далі ми розглянемо кожний компонент середовища моделювання лабіринту докладніше.

Агент-вирішувач лабіринту. Агент, що переміщується лабіринтом, являє собою симуляцію робота, обладнаного набором датчиків, що дозволяють йому виявляти прилеглі перешкоди і визначати напрямок виходу з лабіринту. Переміщення робота здійснюється двома приводами, що впливають на лінійний та кутовий рух корпусу робота. Приводи робота управляються нейромережею, яка отримує дані від датчиків і видає два управляючі сигнали на приводи.

Наразі ми розглянемо завдання проходження двовимірного лабіринту. Це завдання легко візуалізувати, і відносно легко написати симулятор робота-навігатора для двомірного лабіринту. Основна мета робота – прохід лабіринтом до певної мети за вказану кількість кроків симуляції. Роботом управляє нейромережа, яка розвивається у процесі нейроеволюції.

Алгоритм нейроеволюції починається з дуже простої початкової конфігурації нейромережі, яка має тільки вхідні вузли для датчиків і вихідні вузли для приводів і поступово стає все складнішим, поки не буде знайдено успішного вирішувача лабіринтів. Це завдання ускладняється особливою конфігурацією лабіринту, в якій є кілька глухих кутів, що заважають знайти шлях до мети і заманюють агента в локальні оптимуми ландшафту пристосованості, як згадувалося раніше.

На рис. 2 представлено схематичне зображення агента-вирішувача, виконаного у вигляді робота та задіяного у моделюванні вирішення задачі лабіринту. На цій схемі зафарбоване коло позначає твердий корпус робота. Стрілка всередині зафарбованого кола показує напрямок руху робота. Шість стрілок навколо зафарбованого кола представляють шість далекомірних датчиків, які вказують відстань до найближчої перешкоди у заданому напрямку. Чотири сегменти зовнішнього кола позначають чотири радарних датчика із секторним оглядом, які діють як компас, що вказує напрямок до мети (виходу з лабіринту).

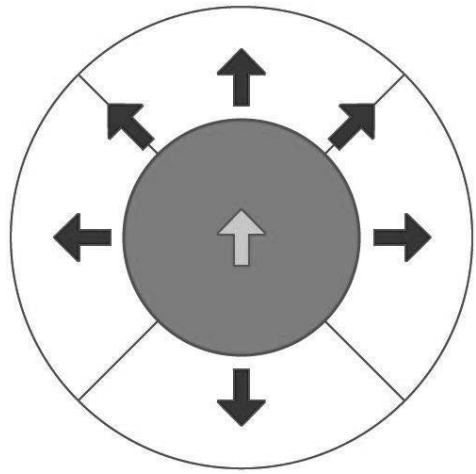

Рис. 2. Схематичне зображення навігаційного агента

Спеціальний радарний датчик активується, коли лінія від точки цілі до центру робота потрапляє у його поле зору (field of view, FOV). Дальність виявлення датчика обмежена зоною лабіринту, яка потрапляє в його поле зору. Таким чином, у будь-який момент часу активований один з чотирьох радарних датчиків, що вказує напрямок виходу з лабіринту.

Радарні датчики мають такі зони огляду щодо курсу робота:

Таблиця 1

Датчик	Поле огляду, градуси
Передній	315,0 ~ 405,0
Лівий	45,0 ~ 135,0
Задний	135,0 ~ 225,0
Правий	225.0 ~ 315.0

Далекомірний датчик – це стежачий промінь, спрямований від центру робота у певному напрямі. Активується під час перетину з будь-якою перешкодою та повертає відстань до виявленої перешкоди. Дальність виявлення цього датчика визначається конкретним параметром конфігурації.

Далекомірні датчики робота відстежують наступні напрямки щодо напрямку руху:

Таблиця 2

Датчик	Напрям, градуси
Правий	-90,0
Передній правий	-45,0
Передній	0,0
Передній лівий	45,0
Лівий	90,0
Задній	-180,0

Рух робота контролюється двома приводами, що прикладають сили, які повертають та/або рухають корпус робота, тобто змінюють його лінійну та/або кутову швидкість.

У реалізації робота-навігатора мовою Python є кілька полів для зберігання його поточного стану та для підтримки станів активності його датчиків:

```
def __init__(self, location, heading=0,
speed=0, angular_vel=0, radius=8.0,
range_finder_range=100.0):
        self.heading = heading
        self.speed = speed
        self.angular_vel = angular_vel
        self.radius = radius
        self.range_finder_range =
range_finder_range
        self.location = location
        # Визначаємо датчики дальноміру
        self.range_finder_angles = [-90.0, -
45.0, 0.0, 45.0, 90.0, -180.0]
        # Визначаємо радарні датчики
        self.radar_angles = [(315.0, 405.0),
(45.0, 135.0), (135.0, 225.0), (225.0, 315.0)]
        # Список станів активності
дальномірів
        self.range_finders = [None] *
len(self.range_finder_angles)
        # Список станів активності
секторних радарів
        self.radar = [None] *
len(self.radar_angles)
```

Цей блок коду демонструє, як створити клас *Agent* зі стандартним конструктором, який задає всі атрибути агенту. Ці атрибути будуть використовуватися симулятором середовища лабіринту для відстеження поточного положення агента на кожній ітерації симуляції.

Реалізація середовища моделювання лабіринту. Щоб змоделювати поведінку агента-вирішувача, що досліджує лабіринт, нам потрібно визначити середовище, яке керує конфігурацією лабіринту, відстежує положення агента-вирішувача і забезпечує вхідні дані для даних датчика навігаційного робота.

Всі ці функції поміщаються в один логічний блок, інкапсульований в клас *MazeEnvironment*, що має наступні поля:

```
def __init__(self, agent, walls, exit_point,
exit_range=5.0):
        self.walls = walls
        self.exit_point = exit_point
        self.exit_range = exit_range
        # Агент-вирішувач лабіринта
        self.agent = agent
        # Прапор індикації про те, що вихід
        знайдено
        self.exit_found = False
        # Початкова відстань від агента до
        виходу
```

```
self.initial_distance =
self.agent_distance_to_exit()
```

У попередньому коді показано конструктор класу *MazeEnvironment* з ініціалізацією всіх його полів:

– конфігурація лабіринту визначається списком стін і точки виходу. Стіни (*walls*) – це списки відрізків; кожен відрізок лінії представляє певну стіну в лабіринті, а точка виходу (*exit_point*) - місце розташування виходу з лабіринту;

– у полі *exit_range* зберігається значення відстані до точки виходу, що визначає зону виходу. Ми вважаємо, що агент успішно пройшов лабіринт, коли його позиція знаходиться у зоні виходу;

– поле *agent* містить посилання на ініціалізований клас агента, описаний раніше, який визначає початкове місце розташування агента в лабіринті та інші поля агента;

– поле *initial_distance* зберігає відстань від початкової позиції агента до точки виходу із лабіринту. Це значення буде пізніше використано для розрахунку показника пристосованості агента.

Генерація сенсорних сигналів. Агент, що проходить лабіринт, управляється нейромережею, якій необхідно мати дані датчиків на вході для формування відповідних керуючих сигналів на виході. Як ми вже згадували, робот-навігатор оснащений масивом двох типів датчиків:

– шість далекомірних датчиків для запобігання зіткненням зі стінами лабіринту, які показують відстань до найближчої перешкоди у певному напрямку;

– чотири секторні радарні датчики, які вказують напрямок до точки виходу з лабіринту з будь-якого його місця.

Показники датчиків необхідно оновлювати на кожному кроці моделювання, а клас *MazeEnvironment* надає два посилальних методи, які оновлюють датчики обох типів.

Масив далекомірних датчиків оновлюється в такий спосіб:

```
for i, angle in enumera-
te(self.agent.range_finder_angles):
        rad = geometry.deg_to_rad(angle)
        projection_point = geometry.Point(
                x = self.agent.location.x +
math.cos(rad) *
self.agent.range_finder_range,
                y = self.agent.location.y +
math.sin(rad) * self.agent.range_finder_range
        )
        projec-
tion_point.rotate(self.agent.heading,
self.agent.location)
        projection_line = geometry.Line(a =
self.agent.location,  b= projection_point)
        min_range =
self.agent.range_finder_range
        for wall in self.walls:
                found, intersection =
wall.intersection(projection_line)
                if found:
                        found_range = intersec-
tion.distance(self.agent.location)
                        if found_range <
min_range:
                                min_range =
found_range
        # Зберігаємо відстань до
найближчої перешкоди
        self.agent.range_finders[i] =
min_range
```

Цей код перераховує всі напрямки далекомірних датчиків, які визначаються кутами напрямку. Потім для кожного напряму вибудовується лінія проекції, починаючи з поточної позиції робота і з довжиною, що дорівнює дальності виявлення далекоміра. Після цього лінія перевіряється на предмет її перетину зі стінами лабіринту. Якщо виявлено кілька перетинів, відстань до найближчої стіни зберігається як поточне значення для конкретного далекоміра. В іншому випадку як поточне значення буде збережена величина максимальної дальності виявлення.

Масив секторних радарних датчиків оновлюється за допомогою коду в класі *MazeEnvironment*:

```
def update_radars(self):
        target = geometry.Point(self.exit_point.x, self.exit_point.y)
        target.rotate(self.agent.heading, self.agent.location)
        target.x -= self.agent.location.x
        target.y -= self.agent.location.y
        angle = target.angle()
        for i, r_angles in enumerate(self.agent.radar_angles):
                self.agent.radar[i] = 0.0
                if (angle >= r_angles[0] and angle < r_angles[1]) or (angle + 360 >= r_angles[0] and angle + 360 < r_angles[1]):
                        # Тригер радар
                        self.agent.radar[i] = 1.0
```

Цей код створює копію точки виходу з лабіринту і повертає її щодо курсу та положення агента в глобальній системі координат. Потім цільова точка транслюється в локальну систему координат агента, що досліджує лабіринт; агент перебуває на початку координат. Після цього ми обчислюємо кут вектору, утвореного від початку координат до цільової точки в локальній системі координат агента. Цей кут є азимутом до точки виходу з лабіринту поточної позиції агента. Коли азимутальний кут знайдено, ми перераховуємо зареєстровані радарні датчики, щоб знайти той, у якого поточний азимутальний кут потрапляє в поле зору (FOV). Відповідний радарний датчик активується шляхом встановлення його значення в 1.0, тоді як інші радарні датчики деактивуються через обнулення їхніх значень.

Оновлення позиції агента. Положення агента в лабіринті необхідно оновлювати на кожному етапі моделювання після отримання відповідних сигналів управління від ШНМ контролера. Для оновлення позиції агента виконується наступний код:

```
def update(self, control_signals):
        if self.exit_found:
                return True # Вихід знайдено
        self.apply_control_signals(control_signals)
        vx = math.cos(geometry.deg_to_rad(self.agent.heading)) * self.agent.speed
        vy = math.sin(geometry.deg_to_rad(self.agent.heading)) * self.agent.speed
        self.agent.heading += self.agent.angular_vel
        if self.agent.heading > 360:
                self.agent.heading -= 360
        elif self.agent.heading < 0:
                self.agent.heading += 360
        new_loc = geometry.Point(
                x = self.agent.location.x + vx,
                y = self.agent.location.y + vy)
        if not self.test_wall_collision(new_loc):
                self.agent.location = new_loc
        self.update_rangefinder_sensors()
        self.update_radars()
        distance = self.agent_distance_to_exit()
        self.exit_found = (distance < self.exit_range)
        return self.exit_found
```

Функція *update(self, control_signals)* визначена у класі *MazeEnvironment* викликається на кожному кроці моделювання. Вона отримує список з керуючими сигналами як вхідні дані і повертає логічне значення, що вказує, чи досяг агент зони виходу після оновлення своєї позиції.

Код на початку цієї функції застосовує отримані сигнали, що управляють, до поточних значень кутової та лінійної швидкостей агента наступним чином:

```
self.agent.angular_vel += (
                        control_signals[0] - 0.5)
self.agent.speed += (control_signals[1] - 0.5)
```

Потім обчислюються компоненти швидкості *x* і *y* разом із напрямом умовної передньої частини агента та використовуються для оцінки його нового положення в лабіринті. Якщо ця нова позиція не стикається ні з однією зі стін лабіринту, то вона призначається агенту і стає його поточною позицією:

```
vx = math.cos(geometry.deg_to_rad(
```

```
        self.agent.heading)) * self.agent.speed
vy = math.sin(geometry.deg_to_rad(
        self.agent.heading)) * self.agent.speed
self.agent.heading += self.agent.angular_vel
if self.agent.heading > 360:
        self.agent.heading -= 360
elif self.agent.heading < 0:
        self.agent.heading += 360
new_loc = geometry.Point(
        x = self.agent.location.x + vx,
        y = self.agent.location.y + vy)
if not self.test_wall_collision(new_loc):
        self.agent.location = new_loc
```

Далі нова позиція агента використовується у функціях, які оновлюють далекомірні та радарні датчики, отримуючи значення нових входів датчиків для наступного тимчасового кроку:

```
self.update_rangefinder_sensors()
self.update_radars()
```

Нарешті наступний блок коду перевіряє, чи досяг агент виходу з лабіринту, що визначається круглою зоною навколо точки виходу з радіусом, рівним значенню поля *exit_range*:

```
distance = self.agent_distance_to_exit()
self.exit_found = (distance < self.exit_range)
return self.exit_found
```

Якщо вихід з лабіринту було досягнуто, значення поля *exit_found* встановлюється рівним *True*, щоб повідомити про успішне вирішення завдання, і це значення повертається з виклику функції.

Зберігання записів агента. Після завершення експерименту нам знадобиться оцінка та візуалізація того, як кожен окремий агент-вирішувач працював протягом еволюційного процесу в усіх поколіннях. Для цього ми збираємо додаткові статистичні дані про кожного агента після запуску моделі проходження лабіринту протягом певної кількості тимчасових кроків. Це досягається шляхом збору додаткових статистичних даних про кожного агента після запуску задачі моделювання вирішення лабіринту протягом визначеної кількості часових кроків.

Колекція записів агента опосередковується двома класами: *AgentRecord* та *AgentRecordStore*.

Клас *AgentRecord* складається з декількох полів даних, як видно з конструктора класу:

```
def __init__(self, generation, agent_id):
        self.generation = generation
        self.agent_id = agent_id
        self.x = -1
        self.y = -1
        self.fitness = -1
        self.hit_exit = False
        self.species_id = -1
        self.species_age = -1
```

Поля мають таке призначення:

- generation містить ідентифікатор покоління, коли було створено запис агента;
- *agent_id* – унікальний ідентифікатор агента;
- *x* та *y* – позиція агента в лабіринті після завершення симуляції;
- *fitness* – підсумкова оцінка пристосованості агента;
- *hit_exit* – це прапор, який вказує, чи досяг агент ділянки виходу з лабіринту;
- *species_id* й *species_age* - ідентифікатор і вік виду, до якого належить агент.

Клас *AgentRecordStore* містить список *AgentRecord* та надає функції для збереження зібраних записів у певний файл та читання з нього. Записи, зібрані під час експерименту, будуть збережені у файлі *data.pickle* в каталозі *output* і можуть бути далі використані для візуалізації працездатності всіх оцінених агентів.

Візуалізація записів агента. Після того як в ході нейроеволюційного процесу будуть зібрані всі оціночні записи всіх агентів, нам буде корисно візуалізувати записані дані, щоб отримати уявлення про стан справ у нашій еволюції. Візуалізація повинна включати кінцеві позиції всіх ви-

рішальних агентів і дозволяти встановлювати порогове значення пристосованості виду для контролю за тим, які види будуть додані до відповідної ділянки. Ми вирішили подати зібрані записи агентів на двох графіках, поданих один над одним. Верхній графік призначений для записів агентів, які належать до видів, у яких показник пристосованості більший або дорівнює вказаному граничному значенню, а нижній графік – для інших записів (дивись рис. 7).

4. Визначення цільової функції з використанням показника пристосованості.

Наразі розглянемо створення успішних агентів, які проходять лабіринт, використовуючи цілеорієнтовану цільову функцію (*goal-oriented objective function*) для управління еволюційним процесом. Цільова функція цього типу заснована на оцінці показника пристосованості вирішувача лабіринту шляхом вимірювання відстані між кінцевим положенням і метою (виходом з лабіринту) після виконання 400 кроків моделювання. Іншими словами, цілеспрямована цільова функція орієнтована на конкретну мету і залежить від кінцевої мети експерименту – досягнення ділянки виходу з лабіринту.

Цілеспрямована цільова функція, задіяна в цьому експерименті, визначається наступним чином. По-перше, нам потрібно визначити функцію помилки як евклідову відстань між кінцевою позицією агента в кінці симуляції та позицією виходу з лабіринту:

$$L = \sqrt{\sum_{i=1}^{2}(a_i - b_i)^2}\,, (1)$$

Де L – функція помилки, a – координати кінцевої позиції агента та b – координати виходу з лабіринту. У цьому експерименті ми використовуємо двовимірний лабіринт, тому координати мають два значення, по одному для кожного вимірювання.

Маючи функцію помилки, ми можемо визначити функцію пристосованості:

$$F = \begin{cases} 1.0 & L \le R_{exit} \\ F_n & otherwise \end{cases}, (2)$$

Де R_{exit} – радіус зони виходу навколо точки виходу з лабіринту, а Fn – нормалізований показник пристосованості, який визначається так:

$$F_n = \frac{D_{init} - L}{D_{init}}, (3)$$

Де D_{init} - це початкова відстань від вирішального агента до виходу з лабіринту на початку навігаційного моделювання.

Рівняння (3) нормалізує показник пристосованості, щоб він розташовувався в діапазоні (0, 1], але може повернути негативні значення в тих поодиноких випадках, коли кінцева позиція агента знаходиться далеко і від його початкової позиції, і від виходу з лабіринту. Щоб уникнути негативних значень, ми будемо застосовувати до нормалізованого показника пристосованості такі поправки:

$$F_n = \begin{cases} 0.01 & F_n \le 0 \\ F_n & otherwise \end{cases}, (4)$$

Коли показник пристосованості менше або дорівнює 0, йому буде присвоєно мінімальне підтримуване значення 0,01; в іншому випадку він залишиться як є. Ми обрали мінімальний показник пристосованості вищий за нуль, щоб надати кожному геному шанс на розмноження.

Наступний код *Python* реалізує описану вище цілеспрямовану цільову функцію:

```
fitness = env.agent_distance_to_exit()
if fitness <= self.exit_range:
        fitness = 1.0
else:
        fitness = (env.initial_distance - fitness) / env.initial_distance
        if fitness <= 0.01:
                fitness = 0.01
```

Далі ми розглянемо проведений експеримент та проаналізуємо отримані результати.

5. Експеримент із простою конфігурацією лабіринту

Отже, ми починаємо наш експеримент зі створення успішного навігаційного агента, який досліджує просту конфігурацію лабіринту. Подібна конфігурація лабіринту, не зважаючи на згадані раніше глухі кути з оманливими локальними оптимумами, має відносно прямий шлях від початкової точки до точки виходу.

На рис. 3 зображена конфігурація простого лабіринту, задіяного в поточному експерименті.

Лабіринт має дві фіксовані позиції, подзначені забарвленими кругами. Верхнє ліве коло позначає початкову позицію агента-вирішувача лабіринту. Нижнє праве коло позначає точне місце виходу з лабіринту, яке має бути знайдене вирішувачем. Для виконання завдання, розв'язувач лабіринту повинен досягти поблизу точки виходу з лабіринту, позначеної спеціальною зоною виходу навколо нього.

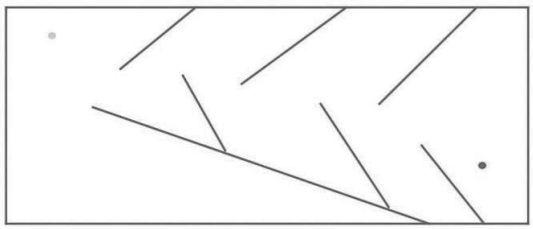

Рис. 3. Конфігурація простого лабіринту

Вибір гіперпараметрів. Згідно з визначенням цільової функції (формула 3), максимальне значення показника пристосованості агента, яке може бути отримано при досягненні заданого околу виходу з лабіринту, становить 1.0. Завдання пошуку рішення для задачі навігації лабіринту є доволі складним, тому для успішного пошуку рішення необхідно використовувати широку зону пошуку у просторі рішень. Для цього, методом спроб і помилок ми виявили, що чисельність популяції може бути рівною 250. Це дає гарний результат, одночасно маючи адекватний час на завершення алгоритму нейроеволюції.

Початкова конфігурація фенотипу ШНМ має 10 вхідних вузлів, 2 вихідних вузли та 1 прихований вузол. Вузли входу та виходу відповідають вхідним датчикам та виходам керуючого сигналу. Прихований вузол спрямований на введення нелінійності від початку нейроеволюційного процесу та економію часу еволюції на додавання цього вузла.

Для розширення зони пошуку рішень нам також необхідно збільшити видове розмаїття популяції, щоб спробувати різні зміни геному протягом обмеженого числа поколінь. Цього можна досягти зменшенням порогу сумісності, або через збільшення значень коефіцієнтів, які використовуються для розрахунку показників сумісності геному. У цьому експерименті ми використовували обидві поправки, тому що ландшафт функції пристосованості оманливий, і нам потрібно підкреслити навіть крихітні зміни в конфігурації геному, щоб створити новий вид. На це впливають такі параметри конфігурації:

```
[NEAT]
compatibility_disjoint_coefficient = 1.1
[DefaultSpeciesSet]
compatibility_threshold = 3.0
```

Ми особливо зацікавлені у створенні оптимальної конфігурації ШНМ, яка має мінімальну кількість прихованих вузлів та зв'язків. Оптимальна конфігурація менш затратна в обчислювальному відношенні під час навчання нейроеволюційним методом, а також під час фази виведення у симуляторі проходження лабіринту. Оптимальну конфігурацію нейромережі можна отримати, зменшивши можливість додавання нових вузлів, як показано в наступному фрагменті з файлу конфігурації NEAT:

```
node_add_prob = 0.1
node_delete_prob = 0.1
```

Нарешті ми дозволяємо нейроеволюційному процесу використовувати не тільки нейромережу з прямим зв'язком, а й рекурентні нейромережі. Дозволяючи рекурентні з'єднання, ми даємо можливість нейромережі мати пам'ять і стати кінцевим автоматом. Це корисно для еволюційного процесу. Наступний гіперпараметр конфігурації впливає на цей фактор:

```
feed_forward = False
```

Гіперпараметри, наведені вище, виявилися корисними для алгоритму NEAT, який використовується в експерименті для навчання протягом обмеженої кількості поколінь успішного агента, що проходить лабіринт.

Результати експерименту. Експеримент був проведений з використанням Python 3.10 та бібліотеки NEAT-Python версії 0.92 (10). Робоча станція, що використовувалась для цього, має наступні параметри: CPU 2,3 GHz 8-Core Intel Core i9, 16 GB 2667 MHz DDR4, macOS 13.5.2.

Успішний агент-вирішувач був знайдений після 144 поколінь нейроеволюції та має наступну конфігурацію геному:

Nodes:

0 DefaultNodeGene(key=0, bias=5.534849614521037, response=1.0, activation=sigmoid, aggregation=sum)

1 DefaultNodeGene(key=1, bias=1.8031133229851957, response=1.0, activation=sigmoid, aggregation=sum)

158 DefaultNodeGene(key=158, bias=-1.3550878188609456, response=1.0, activation=sigmoid, aggregation=sum)

Connections:

DefaultConnectionGene(key=(-10, 158), weight=-1.6144052085440168, enabled=True)

DefaultConnectionGene(key=(-8, 158), weight=-1.1842193888036392, enabled=True)

DefaultConnectionGene(key=(-7, 0), weight=-0.3263706518456319, enabled=True)

DefaultConnectionGene(key=(-7, 1), weight=1.3186165993348418, enabled=True)

DefaultConnectionGene(key=(-6, 0), weight=2.0778575294986945, enabled=True)

DefaultConnectionGene(key=(-6, 1), weight=-2.9478037554862824, enabled=True)

DefaultConnectionGene(key=(-6, 158), weight=0.6930281879212032, enabled=True)

DefaultConnectionGene(key=(-4, 1), weight=-1.9583885391583729, enabled=True)

DefaultConnectionGene(key=(-3, 1), weight=5.5239054588484775, enabled=True)

DefaultConnectionGene(key=(-1, 0), weight=0.04865917999517305, enabled=True)

DefaultConnectionGene(key=(158, 0), weight=0.6973191076874032, enabled=True)

Успішний агент-вирішувач зміг досягти зони виходу з лабіринту за 388 кроків з відведених 400. Конфігурація нейромережі успішного вирішувача містить 2 вихідні вузли та 1 прихований вузол, з 11 зв'язками між цими вузлами та входами. Остаточна конфігурація нейромережі показана на рис. 4.

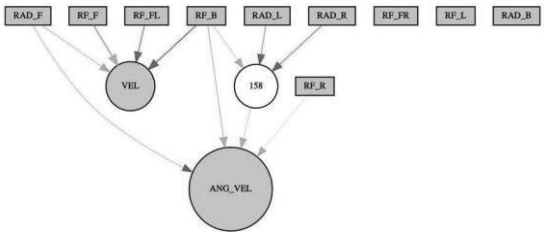

Рис. 4. Конфігурація нейромережі, що відповідає успішному вирішувачу простого лабіринту

Цікаво поглянути на граф нейромережі з точки зору впливу різних входів датчиків на вихідні керуючі сигнали. Ми можемо бачити, що конфігурація нейромережі повністю ігнорує вхідні сигнали від переднього і лівого далекомірних датчиків (*RF_FR* і *RF_L*) і від секторного радарного датчика *RAD_B* робота. Водночас лінійні та кутові швидкості робота залежать від унікальних комбінацій інших датчиків.

Крім того, ми можемо бачити агрегацію лівого та правого радарних датчиків (*RAD_L* і *RAD_R*) з далекоміром *RF_B* через прихований вузол, який потім ретранслює агрегований сигнал вузлу, що управляє кутовою швидкістю. Якщо ми подивимося зображення простої конфігурації лабіринту (рис. 3), то подібна агрегація виглядає досить природною. Вона дозволяє

роботу розвернутися і продовжити досліджувати лабіринт, коли він застряг в одному з глухих кутів, де розташовані локальні оптимуми.

Оцінку пристосованості агентів за поколіннями показано на рис. 5. На цьому графіку ми можемо бачити, що еволюційний процес зміг продукувати досить успішних агентів-вирішувачів у поколінні 44 з оцінкою пристосованості 0,96738. Але знадобилося ще 100 поколінь, аби розвинути геном, який кодує нейромережу успішного агента.

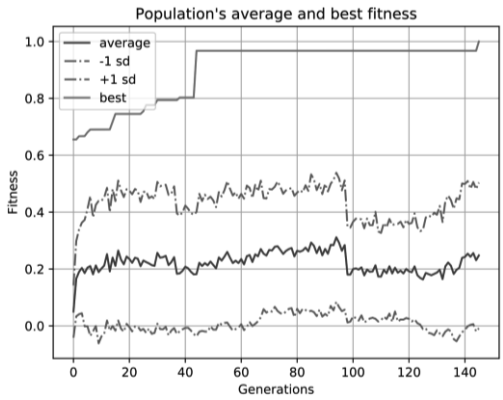

Рис. 5. Середні оцінки пристосованості агентів за поколіннями

Крім того, цікаво відзначити, що підвищення продуктивності в поколінні 44 генерується видами з ID 1, але геном успішного вирішувача належить виду з ID 7, який навіть не був відомий під час першого сплеску. Види, що породили чемпіона, з'явилися через 12 поколінь і залишалися в популяції до кінця, зберігаючи корисну мутацію і вдосконалюючись на її основі.

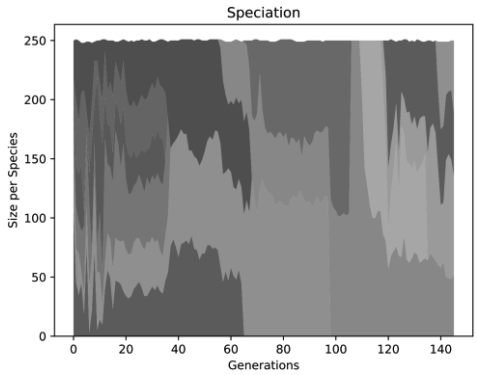

Рис. 6. Видоутворення за поколіннями

Видоутворення протягом кількох поколінь показано на рис. 6. На цьому графіку ми також спостерігаємо вид з ID 7. Цей вид у кінцевому підсумку породив геном успішного вирішувача за час еволюційного процесу. Розмір виду 7 значно варіюється протягом всього його життя, і свого часу він був єдиним видом у всій популяції упродовж декількох поколінь (від 105 до 108).

Візуалізація запису агента. У цьому експерименті представлено новий метод візуалізації, який дозволяє візуально розрізняти ефективність різних видів в еволюційному процесі. Візуалізація базується на збережених даних щодо проходження лабіринту кожним з агентів-вирішувачів протягом усіх епох нейроеволюції.

Візуалізатор малює кінцеві позиції агентів на карті лабіринту в кінці симуляції проходження. Кінцева позиція кожного агента представлена у вигляді кольорового кола. Колір кола означає вид, до якого належить конкретний агент. Кожен вид, отриманий у процесі еволюції, має унікальний колірний код. Результати цієї візуалізації показано на рис. 7.

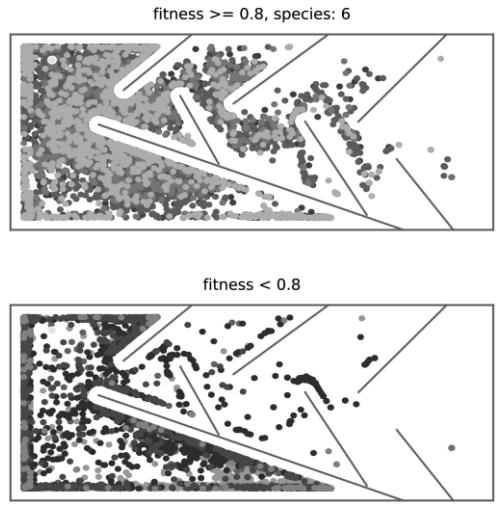

Рис. 7. Візуалізація оцінки агентів-вирішувачів

Для більшої інформативності візуалізації введено поріг пристосування для фільтрації найбільш ефективних видів. Верхня половина рисунка показує кінцеві

позиції вирішальних агентів, що належать до видів-чемпіонів (показник пристосованості вище 0,8). Як можна побачити, організми, що належать до цих шести видів, є активними дослідниками, у яких є гени, що провокують пошук у невідомих місцях лабіринту. Їхні кінцеві розташування майже рівномірно розподілені ділянками лабіринту навколо початкової точки, мають низьку щільність у глухих кутах локальних оптимумів.

Водночас на нижній половині рисунка можна побачити, що еволюційні невдахи демонструють консервативнішу поведінку, концентруючись в основному біля стін у стартовій зоні і в найбільш вираженій ділянці локального оптимуму - найбільшому глухому куті, який знаходиться в нижній частині лабіринту.

Висновки

Досліджено можливість навчання ефективних агентів-розв'язувачів задачі навігації лабіринтом за допомогою алгоритму NEAT. Надано математичний аналіз цільової функції, яка підходить для оптимізації процесу навчання агентів-розв'язувачів під час нейроеволюції. За допомогою запропонованої цільової функції було створено програмний продукт для управління нейроеволюційним процесом.

Було розроблено систему моделювання поведінки робота, який може автономно знайти вихід з лабіринту, використовуючи сигнали від різних типів вхідних датчиків. За допомогою розробленої моделюючої системи, було здійснено низку експериментів для встановлення оптимальних значень гіперпараметрів для ефективного навчання керуючої ШНМ методом нейроеволюції.

Було створено спеціалізоване програмне рішення для відображення процесу навчання агентів-розв'язувачів. Розроблені методи відображення суттєво сприяють пошуку оптимальних параметрів алгоритму NEAT за рахунок наочної демонстрації впливу зміни того чи іншого параметру на процес навчання.

У майбутньому отримані дані можуть сприяти створенню енергозберігаючих керуючих ШНМ для регулювання роботи фізичних роботів. Крім того, розроблена система комп'ютерної імітації, дозволяє виконувати велику кількість досліджень у стислі терміни та з мінімальними витратами.

Все це обумовлює актуальність досліджень щодо використання алгоритму NEAT для вирішення широкого кола прикладних задач та проблем моделювання автономних агентів.

References

1. Jean-Baptiste Mouret and Stephane Doncieux. 2012. Encouraging behavioral diversity in evolutionary robotics: An empirical study. Evolutionary computation 20, 1 (2012), 91–133.
2. Joel Lehman and Kenneth O Stanley. 2010. Revising the evolutionary computation abstraction: minimal criteria novelty search. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2010). ACM, 103–110.
3. Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. Evolutionary computation 19, 2 (2011), 189–223.
4. Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. Evolutionary computation 10, 2 (2002), 99–127.
5. Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. Frontiers in Robotics and AI 3 (2016), 40.
6. Jonathan C. Brant and Kenneth O. Stanley. 2017. Minimal criterion coevolution: a new approach to open-ended search. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17). Association for Computing Machinery, New York, NY, USA, 67–74.
7. Iaroslav Omelianenko. Hands-On Neuroevolution with Python: Build high-performing artificial neural network architectures using neuroevolution-based algorithms. Birmingham, UK: Packt Publishing Ltd, 2019. ISBN: 9781838824914, 368 pp.
8. Iaroslav Omelianenko. Creation of Autonomous Artificial Intelligent Agents Using Novelty Search Method of Fitness Function Optimization. NewGround LLC, Sept. 2018, https://hal.science/hal-01868756.
9. Iaroslav Omelianenko. “Autonomous Artificial Intelligent Agents”. In: Machine Learning

and the City. John Wiley Sons, Ltd, 2022. Chap. 12, pp. 263–285. ISBN: 9781119815075, DOI: 10.1002/9781119815075.ch21, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119815075.ch21, SCOPUS: 2-s2.0-85147956837, https://www.scopus.com/record/display.uri?eid=2-s2.0-85147956837&origin=inward&txGid=b119d677e846feb8f319ba241f759c75

10. McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C., & Netto, M. L. neat-python [Computer software], https://github.com/CodeReclaimers/neat-python

Одержано: 28.09.2023

Про автора:

Омельяненко Ярослав Вікторович,
молодший науковий співробітник.
Кількість зарубіжних публікацій – 7.
Кількість патентів - 2.
https://orcid.org/0000-0002-2190-5664

Місце роботи автора:

Інститут Програмних Систем
НАН України,
03187, м. Київ-187,
проспект Академіка Глушкова, 40.
Тел.: (044) 526 3559.
E-mail: yaric@newground.com.ua