ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КИЇВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Тарануха Володимир Юрійович

ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ

Лабораторний практикум

Київ – 2026

УДК 004.8(076)
Т 19

Рецензенти: д.ф.-м.н., професор, Марченко О.О., професор факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка
д.ф.-м.н., професор, Провотар О.І., професор факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка

Рекомендовано до друку вченою радою факультету комп'ютерних наук та кібернетики
(протокол № 15 від 27 травня 2026 р.)

Ухвалено науково-методичною комісією факультету комп'ютерних наук та кібернетики
(протокол № 11 від 18 травня 2026 р.)

к.ф.-м.н. Тарануха Володимир Юрійович
Інтелектуальні системи: лабораторний практикум : навчальний посібник / В.Ю.Тарануха. – Київ : 2026. – 52 с.

Анотація. Навчальний посібник «Інтелектуальні системи: лабораторний практикум» присвячено формуванню практичних навичок розв'язування інтелектуальних задач із використанням моделей подання знань, інтелектуального пошуку, евристичних методів, оптимізації. Посібник побудовано у формі лабораторних робіт: кожна з них поєднує коротку теоретичну основу, вимоги до реалізації, експериментальну частину та аналіз отриманих результатів.

Для кого призначене видання. Для здобувачів вищої освіти факультету комп'ютерних наук та кібернетики та інших освітніх програм, у межах яких вивчається дисципліна «Інтелектуальні системи» або споріднені дисципліни з технологій штучного інтелекту.

© Тарануха В.Ю., 2026

Зміст

ВСТУП ... 4

Політика академічної доброчесності ... 4

Лабораторна робота 1. Подання знань і бази знань на Пролозі ... 6

Інструментарій та програмне забезпечення ... 6

Теоретичні відомості ... 7

Заперечення та керування пошуком у Пролозі ... 8

Різниця між *->/2 і , ->/2 у if-then-else ... 9

Заперечення як невдача ... 9

Відтинання ! ... 10

Зелені та червоні відтинання ... 11

Комбінація ! і fail для заперечення ... 13

Практичні поради для програмування ... 13

База знань і онтологічна модель у Пролозі ... 13

Типові онтологічні конструкції в Пролозі ... 14

Закритий світ і перевірка узгодженості ... 14

Не онтологічна база знань на Пролозі - «Навчальний процес», приклад фрагмента програми ... 15

Приклади запитів ... 16

Помилки, яких треба уникати ... 16

Постановка задачі ... 17

Вимоги до звіту ... 18

Контрольні запитання ... 19

Критерії оцінювання ... 20

Лабораторна робота 2. Абстрактні стратегічні ігри 20
Інструментарій та програмне забезпечення 21
Теоретичні відомості 21
Евристики впорядкування ходів 24
Складна оціночна функція 26
«Навчання» оціночної функції та проблема перемінних коефіцієнтів 28
Помилки, яких треба уникати 29
Постановка задачі 29
Вимоги до звіту 30
Контрольні запитання 31
Критерії оцінювання 32
Лабораторна робота 3. Евристичний пошук та інтелектуальні агенти . 33
Умови використання Pac-Man-подібного навчального середовища . 33
Інструментарій та програмне забезпечення 34
Теоретичні відомості 35
Простір станів у грі 35
Евристики 36
Пошук шляху 36
Реактивна та кооперативна поведінка 36
Емерджентна поведінка 37
Вплив на геймплей 37
Помилки, яких треба уникати 38
Постановка задачі 38
Експериментальна частина 42

Вимоги до звіту 42
Контрольні запитання 43
Критерії оцінювання 44
Лабораторна робота 4. Генетичні алгоритми та оптимізація 45
Інструментарій та програмне забезпечення 46
Теоретичні відомості 46
Види кодування та відповідні мутації 47
Типи кросоверу 49
Евристики керування пошуком і популяцією 50
Помилки, яких треба уникати 52
Постановка задачі 53
Вимоги до звіту 54
Контрольні запитання 55
Критерії оцінювання 56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 58

ВСТУП

Цей практикум створено для того, щоб студент працював з інтелектуальними задачами на рівні конкретної формалізації, реалізації та перевірки результату. У межах курсу увага зосереджується на тих підходах, де важливо явно задати структуру задачі: описати сутності й зв’язки, побудувати простір станів, обрати стратегію пошуку, визначити функцію оцінювання або правило висновку, а потім перевірити, як саме це впливає на поведінку системи.

Лабораторні роботи підібрано так, щоб студент послідовно переходив від опису знань до пошуку рішень, поведінки агентів і оптимізації. Тому в посібнику акцент зроблено не на демонстрації готових засобів та бібліотек, а на розумінні внутрішньої логіки методів: чому модель працює саме так, які обмеження вона має і в яких випадках її застосування є виправданим.

Особливістю практикуму є свідомий акцент на не нейромережевих підходах. Такий вибір не заперечує ролі машинного навчання, але дозволяє зосередитися на тих класах задач, де вирішальне значення мають структура знань, пояснюваність рішень, керування пошуком і прозорість алгоритмічної поведінки [1,2].

Окрему подяку автор висловлює своїм вчителям, Глибовцю М.М. та Мєдвєдєву М.Г. Ідеї Глибовця М.М. було покладено в основу практикуму, а перші дві лабораторні є розвитком завдань з лабораторних робіт Мєдвєдєва М.Г.

Політика академічної доброчесності

Під час виконання лабораторного практикуму студент зобов’язаний дотримуватися принципів академічної доброчесності.

Порушенням академічної доброчесності в межах лабораторного практикуму вважаються, зокрема:

– подання чужого коду, тексту, моделі, графіків або результатів як власних,

– копіювання розв’язання іншого студента повністю або з незначними змінами,
– використання готових програмних заготовок, бібліотек, шаблонів, генераторів коду або систем штучного інтелекту без належного зазначення факту і способу їх використання,
– підробка результатів тестування,
– навмисне приховування джерел запозичених рішень.

Допускається використання зовнішніх джерел, довідкових матеріалів, документації, відкритих бібліотек і допоміжних інструментів, зокрема систем штучного інтелекту. Проте у такому разі студент повинен:
– явно зазначити у звіті та під час захисту, які саме зовнішні засоби були використані і для чого,
– розуміти отриманий результат,
– уміти пояснити кожний ключовий фрагмент розв’язання,
– бути здатним продемонструвати розуміння під час перевірки шляхом дописування або переписування фрагментів лабораторної на вимогу викладача під час здачі лабораторної.

У короткому підсумку – студент має розуміти матеріал і володіти матеріалом так, наче він повністю створений студентом самостійно.

Якщо викладач встановлює, що студент не володіє матеріалом, не може пояснити нібито власний розв’язок або подав явно неавторський результат, робота може бути повернена на доопрацювання, оцінена частково або зовсім не зарахована на розсуд викладача.

Лабораторна робота 1. Подання знань і бази знань на Пролозі

Лабораторна робота присвячена базовим засобам подання знань у Пролозі. У межах роботи розглядаються способи подання фактів і правил у вигляді предикатів, організація бази знань, формулювання логічних запитів, а також практичне використання механізму логічного виведення Прологу для отримання нових знань із наявних фактів і правил [3-5].

Мета роботи. Ознайомитися з основними принципами логічного програмування в середовищі Пролог та набути практичних навичок побудови бази знань предметної області на основі фактів, правил і запитів. Навчитися формалізувати знання у вигляді предикатів, перевіряти істинність тверджень за допомогою запитів і аналізувати результати логічного виведення.

У межах цієї лабораторної роботи база знань розглядається як множина тверджень про об'єкти предметної області. Для побудови такої бази знань важливо:

- визначити перелік сутностей,
- виділити ключові властивості та зв'язки між сутностями,
- задати їх у вигляді предикатів,
- сформулювати правила для виведення похідних фактів,
- перевірити коректність знань за допомогою запитів.

Інструментарій та програмне забезпечення

Для виконання лабораторної роботи допускається використання як локального, так і браузерного середовища розробки, як то:

- SWI-Prolog [6] як основне середовище розробки та виконання програм на Пролозі,
- SWISH [7] як онлайн-середовище для написання, запуску та перевірки програм на Пролозі без локального встановлення.

Необхідно враховувати, що SWISH працює на спільному сервері в sandbox-режимі, тому небезпечний код блокується, а частина можливостей

SWI-Prolog підтримуються обмежено, як то введення/виведення, паралельність тощо.

Можна використати інший Пролог на свій розсуд, але тоді студент має обгрунтувати причину цього рішення та отримати дозвіл до початку виконання роботи.

Теоретичні відомості

Пролог є мовою логічного програмування, у якій програма розглядається як сукупність фактів і правил. Основною одиницею подання знань є предикат, що описує об’єкт, його властивість або відношення між об’єктами [3,4].

Особливістю Прологу є те, що програміст описує не покроковий алгоритм, а логічну модель предметної області. Механізм виведення самостійно виконує пошук доведення, використовуючи уніфікацію, резолюцію та повернення назад. При цьому Пролог намагається довести ціль (Goal).

Посібник складено з розрахунку на синтаксис SWI-Prolog у першу чергу, тому позначення часто вживатимуться відповідно до нотації, наведеної у документації до SWI-Prolog [6]. Так, наприклад, при описі предикатів та операцій вживатиметься нотація через */n*, де n вказує на розрядність предиката.

Факт у Пролозі задає твердження, яке вважається істинним у межах бази знань. Наприклад:

student(petro).

course(petro, ai).

Правило задає залежність між твердженнями й дозволяє виводити нові знання. Загальний вигляд правила:

висновок :- умова1, умова2, ..., умоваN.

Наприклад:

studies_ai(X) :- student(X), course(X, ai).

Це правило означає: об’єкт X вивчає штучний інтелект, якщо X є студентом і має курс *ai*.

Якщо записано декілька предикатів з однаковою лівою частиною, то вони працюють як альтернативи. Наприклад:

studies_ai(X) :- student(X), course(X, short_ai).

Це правило означає: об’єкт X вивчає штучний інтелект, якщо X є студентом і має курс *short_ai*.

Читати разом два предикати можна як: об’єкт X вивчає штучний інтелект, якщо X є студентом і має курс *short_ai* або *ai*.

Запит – це звернення до бази знань з метою перевірити, чи можна довести певне твердження. Наприклад:

?- studies_ai(ivan).

Якщо твердження виводиться з наявних фактів і правил, Пролог повертає true, інакше – false.

Коментарі мають такий вигляд:

% однорядковий коментар

та

/ це багаторядковий*
*коментар */*

Заперечення та керування пошуком у Пролозі

У Пролозі, окрім фактів, правил і звичайного механізму повернення з відкотом (*backtracking*), важливу роль відіграють засоби керування пошуком. До них належать заперечення як невдача *\+/1*, відтинання *!/0*, примусовий провал *fail/0*, а також конструкції **->/2* (наслідок), *->/2*(наслідок), *;/2*(або) і предикат *once/1* (один раз і досить). У синтаксисі Прологу диз’юнкції та конструкції if-then-else (через **->/2* і *->/2*) бажано завжди брати в дужки, це відповідає і документації, і типовому стилю запису програм у SWI-Prolog [6].

Різниця між *->/2 і , ->/2 у if-then-else

Загальна структура має вигляд: If -> Then *;* Else або If *-> Then *;* Else і між ними є різниця.

Нехай є приклад, де умова має кілька розв'язків:

*p_soft(X):- (member(X, [a,b]) *-> true ; X = none)*

p_hard(X) :- (member(X, [a,b]) -> true ; X = none).

Отримаємо такі результати при запитах:

?- p_soft(X).

X = a;

X = b.

?- p_hard(X).

X = a.

У випадку -> після першого успіху інші варіанти умови вже не перебираються.

Заперечення як невдача

У SWI-Prolog [6] заперечення \+ *Goal* означає не класичне логічне «не Goal», а твердження «Goal не вдається довести». У SWI-Prolog \+*/1* реалізовано як керувальну конструкцію, тому \+ має і декларативний, і процедурний зміст.

Практично це означає, що \+ найнадійніше використовувати тоді, коли потрібні змінні вже конкретизовані попередніми цілями. Якщо ж викликати заперечення над неінстанційованою змінною, результат може не відповідати інтуїтивному «логічному запереченню». Основне правило: спочатку звузити область пошуку, а вже потім застосовувати \+.

student(sofiya).

student(petro).

submitted(sofiya).

not_submitted(X) :- student(X), \+ submitted(X).

?- not_submitted(X).

X = petro.

Цей варіант коректний у звичному для бази знань режимі: спочатку student(X) перебирає кандидатів, а вже потім перевіряється відсутність факту submitted(X).

not_submitted_bad(X) :- \+ submitted(X), student(X).

?- not_submitted_bad(X).

false.

Тут заперечення стоїть до конкретизації X. Оскільки ціль *submitted(X)* можна довести хоча б для *X* = *sofiya*, вираз \+ *submitted(X)* одразу провалюється, і правильний результат не буде знайдено. Саме тому порядок цілей у Пролозі може змінювати як ефективність, так і сам набір відповідей.

Відтинання !

Відтинання *!* – це спеціальний керувальний предикат, який завжди успішний, але відсікає точки вибору, створені від моменту входу в поточне правило. Інакше кажучи, після проходження через *!* Пролог уже не повертається до альтернатив, що були створені ліворуч у цьому самому правилі, а також фіксує вибір поточної клаузи.

У SWI-Prolog [6] важливо пам’ятати, що *!* має різну поведінку для *call/1* з одного боку та для *;/2*, *->/2* і **->/2* з іншого.

Розглянемо:

p :- (a, !, fail ; b).

p :-d.

Суть: спробувати виконати a. Якщо a успішне, виконати *!*. Відтинання фіксує поточний шлях виконання і прибирає альтернативи, створені ліворуч від відтинання. Оскільки *;/2* є «прозорим» для відтинання, *!* також прибирає альтернативну гілку b. Після цього fail примушує вибрану гілку завершитися невдачею, але Пролог уже не має права перейти до b. Більш того, навіть якщо

d істинне, воно вже не буде перевірене. Відтинання зафіксувало вибір першої клаузи p, і тепер p теж зазнало невдачі.

У той же час *call/1* є «непрозорим» для *!*. Розглянемо:

p:- call((a, !, fail ; b)).

p:-d.

Якщо a успішне, то результат всередині *call/1* той самий, але можна досягти успіху p, якщо d успішне.

Найчастіше *!* застосовують для двох цілей: або щоб прибрати беззмістовний повторний перебір**,** або щоб навмисно зафіксувати процедурний сценарій виконання. Саме звідси походить розрізнення між зеленими і червоними відтинаннями.

Зелені та червоні відтинання

Зелене відтинання (green cut) не змінює логічного змісту програми: воно лише прибирає ті гілки пошуку, які все одно не дали б нових правильних відповідей. Червоне відтинання (red cut), навпаки, є необхідним для правильної роботи конкретного процедурного варіанта програми: якщо прибрати таке !, програма почне повертати інші відповіді або працюватиме неправильно.

Приклад зеленого відтинання:

sign_green(X, positive) :-   X > 0,   !.

sign_green(X, non_positive) :-   X =< 0.

Приклади запитів:

?- sign_green(5, S).

S = positive.

?- sign_green(-2, S).

S = non_positive.

У цьому прикладі дві умови взаємовиключні: якщо X > 0 вже істинне, то немає сенсу повертатися й перевіряти друге правило X =< 0. Відтинання лише

усуває зайвий перебір, але не змінює множину правильних відповідей. Тому це типовий приклад зеленого відтинання.

Приклад червоного відтинання:

sign_red(X, S) :- X > 0, !, S = positive.

sign_red(_X, non_positive).

Приклад запиту:

?- sign_red(5, S).

S = positive.

На перший погляд, програма працює правильно. Але тут *!* уже є не просто оптимізацією, а необхідною частиною процедурної поведінки. Якщо прибрати відтинання, друге правило стане доступним після повернення з відкотом, і для додатного числа програма зможе помилково повернути ще й відповідь non_positive:

sign_red_no_cut(X,S) :- X > 0, S = positive.

sign_red_ no _cut(_X, non_positive).

Тоді запит може дати дві відповіді:

?- sign_red_no_cut(5, S).

S = positive;

S = non_positive.

Саме тому такий варіант належить до червоного відтинання: правильність відповіді залежить від процедурного ефекту *!*.

Ще небезпечніший варіант виглядає так:

sign_bad(X, positive) :- X > 0,!.

sign_bad(_X, non_positive).

На запит:

?- sign_bad(5, non_positive).

true.

Програма повертає неправильний результат: число 5 не є недодатним. Помилка виникає через те, що друге правило стало надто загальним і спрацьовує для будь-якого аргументу, якщо перше правило не було

застосоване через невідповідність голови предиката. Відтинання тут не виправляє логіку предиката, а лише примусово керує переходами між гілками. Такий приклад добре показує, що червоні відтинання можуть маскувати логічні дефекти означення.

Комбінація ! і fail для заперечення

Заперечення можна реалізувати як комбінацію відтинання і примусового провалу:

negation(Goal) :- Goal, !, fail.

negation(_Goal). % можна написати _ замість _Goal, на результат не вплине

Якщо Goal вдалося довести, після *!* Пролог уже не має права шукати альтернативу, а *fail* змушує весь виклик провалитися. Якщо ж *Goal* не доводиться, перше правило не спрацьовує, і друге правило дає успіх. Саме на цій ідеї базується заперечення як невдача, в реальних програмах замість саморобного *neg/1* слід використовувати стандартний оператор \+.

Практичні поради для програмування

- Спочатку слід будувати логічно прозоре означення предиката без відтинань.
- Якщо після вилучення *!* правильні відповіді не змінюються, то відтинання, найімовірніше, є зеленим.
- Якщо без *!* програма починає повертати інші відповіді, таке відтинання є червоним і потребує особливо уважного аналізу.
- Заперечення \+ варто застосовувати після того, як змінні, від яких воно залежить, уже конкретизовані.

База знань і онтологічна модель у Пролозі

У простій базі знань Пролог використовується для зберігання фактів і правил про предметну область. Однак ті самі засоби можна застосувати і для побудови більш структурованої моделі знань, що являє собою онтологію. У

такому разі знання організуються не лише як набір окремих тверджень, а як система понять, екземплярів, властивостей, зв’язків і обмежень [8, 9].

Для такого підходу необхідно явно виділити:

- класи предметної області,
- ієрархії типу підклас–надклас,
- атрибути та онтологічні відношення,
- обмеження на допустимі комбінації фактів і зв’язків,
- екземпляри класів.

Онтологічна модель у Пролозі – це не окремий формат файлу, а дисципліновано побудована логічна модель, у якій структура предметної області задана явно, а частина знань виводиться правилами.

Типові онтологічні конструкції в Пролозі

Для онтологічного моделювання в Пролозі зручно використовувати кілька груп предикатів. Окремо виділяють предикати класів, належності екземпляра до класу, ієрархії класів, зв’язків між сутностями, атрибутів і обмежень. Типовими є предикати *subclass/2*, *superclass/2*, *instance/2*, *part_of/2*, *has_part/2*, *has_attribute/3, domain/2, range/2*. При цьому часто варто виділити два різних види предикатів для відношень між частиною і цілим, оскільки деякі цілі зазвичай лишаються такими самими при вилученні складової, а деякі – змінюють властивості. Наприклад, вилучення зернини з купи зерна лишає купу – купою. Вилучення мотора з автомобіля міняє його властивості докорінно [8,9].

Таке розбиття робить програму зрозумілішою, полегшує перевірку коректності моделі та допомагає відрізняти належність до класу від асоціативного зв’язку або атрибута. Саме ця дисципліна подання знань відрізняє просту базу фактів від онтологічно організованої моделі.

Закритий світ і перевірка узгодженості

Під час моделювання знань у Пролозі важливо пам’ятати про підхід закритого світу: якщо певний факт не задано і не виводиться правилами, то в

межах поточної моделі він вважається недоведеним. Через це необхідно особливо уважно задавати словник понять, типи зв'язків і здійснювати перевірку узгодженості. Для розширеної бази знань доцільно додавати предикати, які виявляють помилки моделювання, наприклад належність одного екземпляра до несумісних класів, порушення допустимого типу аргументів відношення або некоректні зв'язки (наприклад – цикли) у відношеннях частина–ціле [8,9].

Не онтологічна база знань на Пролозі - «Навчальний процес», приклад фрагмента програми

```
student(sofiya).
student(bogdan).
student(maria).

course(sofiya, prolog).
course(sofiya, ai_basics).
course(bogdan, databases).
course(maria, prolog).

teaches(ivanenko, prolog).
teaches(petrenko, ai_basics).
teaches(sydorenko, databases).

mandatory(prolog).
mandatory(ai_basics).

studies_prolog(X) :-
    course(X, prolog).

attends_mandatory(X, C) :-
```

```
    student(X),
    course(X, C),
    mandatory(C).

studies_with_teacher(X, T) :-
    course(X, C),
    teaches(T, C).

has_two_courses(X) :-
    course(X, C1),
    course(X, C2),
    C1 @< C2.
/* таке використання на відміну від C1 \= C2 гарантує, що симетричні пари
(prolog, ai_basics) і (ai_basics, prolog) рахуватимуться у видачі як одна */
```

Приклади запитів

```
?- student(sofiya).
?- studies_prolog(sofiya).
?- studies_prolog(bogdan).
?- attends_mandatory(sofiya, prolog).
?- attends_mandatory(bogdan, databases).
?- studies_with_teacher(sofiya, ivanenko).
?- has_two_courses(sofiya).
?- teaches(T, prolog).
```

Помилки, яких треба уникати

Змішування класів, екземплярів і властивостей. Наприклад, одночасне використання *student/1* як класу, як конкретного студента і як властивості. У результаті база знань втрачає чітку структуру. Правильно розрізняти:

клас: *class(student)*.

екземпляр: *instance(petro, student)*.

властивість: *has_attribute(petro, age, 19)*.

відношення: *studies(petro, prolog)*.

Не обов'язково вводити саме такі предикати в кожній роботі, але логічне розрізнення має бути зрозумілим.

Неправильний порядок цілей у правилі. У Пролозі порядок цілей впливає на ефективність, а іноді й на результат. Особливо це важливо при використанні заперечення, арифметичних порівнянь, генерації кандидатів і фільтрації.

Правильний загальний принцип:

rule(X) :- generate_candidate(X), check_condition(X).

Погано:

rule(X) :- check_condition(X), generate_candidate(X).

Такий порядок є помилковим, якщо check_condition(X) вимагає, щоб X уже був конкретизований. Це стосується і \+ також.

Використання *!* для приховування логічної помилки. Відтинання не має бути способом "змусити програму дати потрібну відповідь". Якщо без *!* предикат повертає неправильні або зайві відповіді, треба спочатку перевірити логіку правил. Іноді помилка там, а не тому що це червоне відтинання.

Постановка задачі

Необхідно розробити базу знань на Пролозі для заданої предметної області. Необхідно:

- виділити поняття-класи та, за можливості, підкласи,
- відокремити екземпляри від класів,
- розрізняти ієрархічні, атрибутивні й асоціативні зв'язки,
- реалізувати принаймні одне правило успадкування або інший приклад неявного висновку,
- передбачити хоча б одну перевірку узгодженості моделі.

База знань повинна містити:

- не менше 15 фактів,
- не менше 5 правил,
- не менше 8 змістовних запитів для перевірки роботи програми.

У межах побудованої бази знань слід:

- описати об’єкти предметної області,
- задати між ними відношення,
- реалізувати правила для логічного виведення нових фактів,
- виконати серію запитів і проаналізувати отримані результати.

Необхідно обрати предметну область, відмінну від області «Навчальний процес», оскільки вона використана як приклад та оформити звіт.

Вимоги до звіту

У звіті мають бути наявні:

- тема, мета та завдання лабораторної роботи,
- короткі теоретичні відомості про факти, правила, запити та логічне виведення в Пролозі,
- опис обраної предметної області,
- перелік використаних предикатів із поясненням їх призначення,
- опис класів, екземплярів, атрибутів і типів зв’язків предметної області,
- приклади похідних знань, отриманих через правила успадкування або інші правила висновку,
- приклади перевірок узгодженості та пояснення виявлених помилок моделювання,
- повний текст програми на Пролозі,
- не менше 8 запитів до бази знань,
- результати виконання кожного запиту,
- аналіз отриманих результатів,
- висновки щодо коректності побудованої бази знань і особливостей логічного виведення.

Бажано також окремо вказати:

- які факти є базовими,
- які знання отримуються лише через правила,
- які помилки виникали під час розроблення та як вони були усунуті.

Контрольні запитання

1. Що таке логічне програмування?
2. Чим Пролог відрізняється від імперативних мов програмування?
3. Що таке факт у Пролозі?
4. Що таке правило у Пролозі?
5. Що таке предикат і що означає його арність?
6. Що таке база знань?
7. Яке призначення запиту в Пролозі?
8. Як працює механізм логічного виведення в Пролозі?
9. Що таке уніфікація?
10. Для чого використовується повернення назад?
11. У чому різниця між базовими фактами та знаннями, виведеними за правилами?
12. Які типові помилки трапляються під час побудови бази знань у Пролозі?
13. Чому важливо правильно обирати імена предикатів і структуру аргументів?
14. Як перевірити коректність бази знань?
15. У яких задачах доцільно використовувати Пролог для подання знань?
16. Чим проста база знань відрізняється від онтологічно організованої моделі?
17. Як у Пролозі можна подати класи, екземпляри та ієрархію понять?
18. У чому різниця між відношеннями instance-of, is-a та part-of?
19. Для чого в онтологічній моделі потрібні обмеження та перевірки узгодженості?
20. Які знання доцільно виводити правилами, а які – задавати безпосередньо фактами?

Критерії оцінювання

Оцінювання лабораторної роботи здійснюється за такими рівнями:

На 100%. База знань побудована коректно. Онтологія змістовно задає предметну область. Факти й правила відповідають предметній області, та детально її описують. Запити сформульовано правильно. Результати логічного виведення інтерпретовано коректно. Звіт повний, структурований і містить змістовні висновки.

На 70%. База знань загалом коректна, але містить незначні недоліки в структурі онтології. Предикати сформульовано правильно. Результати логічного виведення коректні. У звіті наявні всі основні елементи.

На 50%. База знань містить лише мінімально необхідний набір фактів і правил. Частина предикатів сформульована некоректно або не дає очікуваного результату. Звіт містить неповний опис програми та поверховий аналіз.

У разі, коли завдання виконано менш ніж на 50% лабораторна вважається виконаною на 0.

Лабораторна робота 2. Абстрактні стратегічні ігри

Лабораторна робота присвячена ігровому пошуку в детермінованих позиційних двоосібних антагоністичних іграх із повною інформацією. У межах роботи розглядаються подання стану гри, побудова дерева позицій, рекурсивний minimax-пошук, обмеження глибини, евристичне оцінювання позицій та альфа-бета-відтинання як спосіб прискорення minimax без зміни підсумкового вибору ходу [10-12].

Мета роботи. Ознайомитися з принципами ігрового пошуку в детермінованих позиційних двоосібних антагоністичних іграх із повною інформацією; навчитися формалізувати гру як простір станів і переходів, реалізовувати дерево гри, алгоритм minimax і його оптимізацію через альфа-бета-відтинання; дослідити вплив глибини пошуку, функції оцінювання та порядку розгляду ходів на ефективність і якість вибору ходу; навчитися

пояснювати, чому коректне відтинання зменшує обсяг перебору, але не змінює minimax-відповідь.

Інструментарій та програмне забезпечення

Для виконання лабораторної роботи рекомендовано використовувати C++ або C#. Допускаються такі формати реалізації:

- консольна програма та вивід через псевдографіку,
- легка графічна візуалізація.

Для подання та аналізу дерева гри можна використати Graphviz.NetWrapper та ScottPlot, їх аналоги для C++, або власну текстову чи табличну візуалізацію невеликих фрагментів дерева. Обраний інструментарій не повинен приховувати логіку алгоритму: засіб має дати змогу показати порядок обходу, зміну alpha і beta та місця відтинання.

За потреби можна використати іншу мову програмування та/або інші засоби, але тоді студент має обґрунтувати причину цього рішення та отримати дозвіл до початку роботи.

Теоретичні відомості

У цій лабораторній роботі розглядаються детерміновані позиційні двоосібні антагоністичні ігри з повною інформацією. Це означає, що ходи виконуються почергово, обидва гравці бачать поточний стан гри, стан гри може бути описаний за допомогою скінченного опису і кількість описів теж скінченна, а випадковість або взагалі відсутня, або не є частиною моделі. Такі ігри природно описуються через дерево позицій.

Один із гравців розглядається як MAX: він прагне максимізувати значення позиції. Інший розглядається як MIN: він прагне це значення мінімізувати. Якщо на певному рівні дерева хід належить MAX, алгоритм обирає найбільше значення серед дочірніх вузлів, якщо хід належить MIN, то обирається найменше. Саме це чергування і становить зміст minimax-пошуку.

Вузол дерева відповідає певному стану гри, а ребра – допустимим ходам, що переводять гру до наступного стану. Термінальні вузли – це позиції, у яких

гра завершена: виграш, програш, нічия або інший фінальний результат. Проміжні вузли – це позиції, з яких гру ще можна продовжити.

Для термінальних позицій використовують точну функцію корисності або, як її ще називають, функцію виграшу, UF. Вона відбиває реальний підсумок гри. Наприклад, для гравця MAX можна взяти +1 за виграш, 0 за нічию і -1 за програш. Проте, краще коли це буде MAX_INT, 0 та MIN_INT відповідно. Для нетермінальних позицій на обмеженій глибині застосовують статичну оціночну функцію – евристичну оцінку, HEF. Статичною вона називається, бо працює лише з поточним полем, на відміну від динамічного/рекурентного виклику minimax. Вона не є точним підсумком гри, а лише наближенням того, наскільки позиція добра для MAX.

Для коротких пояснювальних прикладів використовуються хрестики-нулики, оскільки дерево позицій залишається наочним. Для практичної реалізації гри у лабораторній цього недостатньо, не треба намагатися здати таку задачу.

Нехай X відповідає гравцеві MAX, а O - гравцеві MIN. Розглянемо позицію на рис.1:

X	O	X
O	X	
		O

Рис.1. Позиція гри в хрестики-нулики.

Тут хід MAX. Позиція ще не є термінальною, але якщо MAX ставить X у нижню ліву клітинку, він одразу завершує діагональ і переходить до термінального вузла з значенням функції виграшу UF = +1. Якщо ж пошук обмежено малою глибиною і гра не розгортається до кінця, потрібна оціночна функція, яка, наприклад, враховує кількість відкритих ліній для X і O, наявність двійок із вільною третьою клітинкою, контроль центру та загрозу негайного виграшу або програшу.

Minimax – це рекурсивний алгоритм, який поширює значення від листків дерева до кореня. Якщо вузол термінальний, повертається значення UF. Якщо досягнуто межу глибини, повертається значення HEF. Якщо вузол належить MAX, повертається максимум серед значень нащадків, якщо вузол належить MIN, повертається мінімум серед значень нащадків. Така рекурсія моделює припущення, що обидва гравці діють раціонально в межах заданої моделі [10,11].

У практичних іграх дерево дуже швидко зростає. Це пов'язано з тим, що зазвичай у однієї позиції багато нащадків, і оцінюється середньою кількістю допустимих ходів у позиції. Якщо на кожному рівні є b ходів, а глибина дорівнює d, то розмір дерева зростає приблизно як b^d. Саме тому в реальних постановках часто вводять фіксовану межу глибини пошуку [10-13].

Обмеження глибини неминуче пов'язане з ефектом обрію. Алгоритм може не бачити подію, що лежить трохи далі за межами поточної глибини, і тому переоцінити або недооцінити позицію. Через це якість HEF істотно впливає на підсумковий вибір ходу.

Альфа-бета-відтинання не є окремим алгоритмом вибору ходу. Це спосіб прискорити minimax без зміни підсумкової відповіді. Параметр alpha - це найкраща гарантована нижня межа для MAX, знайдена на поточному шляху зверху вниз. Параметр beta - це найкраща гарантована верхня межа для MIN, знайдена на тому самому шляху [10,13].

Під час обходу дерева у вузлі MAX значення alpha може зростати, а у вузлі MIN значення beta може зменшуватися. Якщо в певний момент alpha >= beta, подальший перегляд решти дітей цього вузла стає непотрібним: жоден із них уже не може змінити рішення предка. Це і є умова відтинання.

Нехай у вузлі MAX уже знайдено варіант із оцінкою 0,6, тобто alpha = 0,6. Далі розглядається інший дочірній вузол, який належить MIN. Перший уже переглянутий нащадок цього вузла дав оцінку 0,4, отже beta = 0,4. Оскільки MAX ніколи не вибере піддерево, яке в найкращому разі дає лише

0,4, якщо вже має варіант із 0,6, решту дітей цього вузла MIN можна не переглядати. Саме так альфа-бета-відтинання зменшує обсяг перебору.

Ефективність альфа-бета-відтинання дуже залежить від порядку розгляду ходів. Якщо сильні ходи перевіряються рано, хороші межі alpha і beta встановлюються швидко, і кількість відтинань зростає. Якщо ж порядок невдалий, алгоритм може переглянути майже стільки ж вузлів, скільки й звичайний minimax. Тому часто вживаються наведені нижче засоби [11-13].

Ітеративне поглиблення – стратегія, за якої пошук запускають послідовно на глибинах 1, 2, 3, ... , d. Вона корисна тим, що завжди дає поточний найкращий знайдений хід і водночас забезпечує добрий матеріал для кращого порядку ходів на наступних ітераціях [10,12].

Таблиця вже оцінених позицій. Вона потрібна тому, що одна й та сама позиція може бути досягнута різними послідовностями ходів. Збереження вже порахованого результату дозволяє не розгортати однакове піддерево повторно. У цій лабораторній роботі достатньо розуміти ідею та вміти пояснити, для чого така таблиця потрібна [10,12].

Евристики впорядкування ходів

Ефективність alpha-beta-відтинання істотно залежить від порядку, у якому розглядаються ходи або нащадки поточного стану. Якщо сильніші або перспективніші ходи переглядаються першими, межі alpha і beta оновлюються раніше, а отже більша частина дерева може бути відсічена без повного перегляду. Для багатьох ігор уже проста евристика дає відчутний ефект: спочатку перевіряти взяття, ходи з негайною загрозою, просування до дамки чи інші тактично сильні продовження. При цьому у класичному alpha-beta-пошуку впорядкування ходів не змінює результат, якщо всі ходи зрештою розглядаються з тією самою глибиною. Воно змінює лише кількість переглянутих вузлів і час роботи алгоритму.

Повне впорядкування ходів. Полягає в тому, що перед розгортанням вузла всі допустимі ходи оцінюються за допомогою швидкої евристики, після чого сортуються від найперспективнішого до найменш перспективного. Наприклад, у грі типу шахів або шашок першими можна переглядати:

- ходи, що ведуть до виграшу або взяття фігури,
- ходи, що дають шах, загрозу або безпосередню тактичну перевагу,
- ходи, які були найкращими на меншій глибині пошуку,
- ходи, які раніше часто спричиняли відтинання.

Часткове впорядкування ходів. У цьому випадку алгоритм не сортує всі ходи, а намагається швидко знайти лише кілька найперспективніших.

І як один із варіантів – відбір n кращих: спочатку вибираються й упорядковуються тільки n найкращих ходів, а решта або переглядається без сортування, або сортується пізніше. Повне сортування всіх нащадків може бути марним, якщо відтинання станеться рано; тому часткове сортування найкращих n ходів є природною альтернативою, хоча треба окремо вирішувати, як вибрати n.

Мале значення n зменшує витрати на сортування, але може не знайти хід, який дав би швидке відтинання. Велике n наближає метод до повного сортування, але збільшує накладні витрати.

Пізні ходи та їх спрощена обробка. Якщо ходи вже впорядковано, то перші кілька ходів вважаються найперспективнішими. Якщо вони не дали достатньо доброго результату або не спричинили відтинання, пізніші ходи часто вважаються менш перспективними. Ідея евристики полягає в тому, що пізні ходи переглядаються з меншою глибиною. Якщо такий скорочений пошук показує, що хід усе ж може бути важливим, його можна повторно переглянути на повній глибині. У практичних реалізаціях перші ходи зазвичай шукаються на повній глибині, а наступні – зі зменшенням глибини [14].

Важливе застереження щодо точності: потрібно розрізняти два типи модифікацій.

1. Безпечні модифікації не змінюють результат minimax, а лише впливають на швидкість. До них належать:

- повне впорядкування ходів,
- часткове впорядкування, якщо всі ходи зрештою переглядаються,

2. Селективні модифікації можуть змінити результат, бо не всі ходи аналізуються однаково. До них належать:

- перегляд лише k найкращих ходів,
- спрощення пізніх ходів,
- відтинання на основі якоїсь евристичної оцінки.

У навчальній роботі обов'язково треба явно вказати, який тип модифікації використано: точний або наближений.

Складна оціночна функція

Загальну форму оціночної функції зручно задати як різницю між оцінкою позиції для гравця, що максимізує результат, і оцінкою позиції для суперника:

$Eval(s) = Score(MAX, s) - Score(MIN, s)$

Розглянемо **найпростішу статичну оціночну функцію** на прикладі шахів. Вона враховує лише співвідношення сил. Її зручно використовувати як базову версію.

$Eval0 = k_1*DeltaP + k_2*DeltaN + k_3*DeltaB + k_4*DeltaR + k_5*DeltaQ$

Тут DeltaP, DeltaN, DeltaB, DeltaR, DeltaQ – різниці у кількості відповідно пішаків, коней, слонів, тур і ферзів між MAX та MIN. k_1–k_5 – відповідні коефіцієнти. Король у цю суму не входить, оскільки його не розглядають як звичайну фігуру для обміну. Ця функція дуже проста і швидка, але не розрізняє активні й пасивні фігури, безпеку короля, структуру пішаків і тактичні загрози.

Сила фігур і позиційні таблиці фігур. Додається оцінка того, де саме стоїть фігура. Наприклад, кінь у центрі зазвичай кращий за коня на краю

дошки, а пішак поблизу поля перетворення цінніший за пішака на початковій позиції.

Eval = Eval0 + Sum(PiecePosition(piece, square))

Отже: кінь у центрі -> бонус, кінь на краю -> штраф, пішак близько до перетворення -> бонус, тура на відкритій вертикалі -> бонус.

Функція лишається простою, але вже дає позиційно осмисленіші ходи, але статичні таблиці можуть помилятися у нетипових позиціях.

Сила фігур і мобільність фігур вимірює кількість доступних ходів або кількість атакованих клітин. Вона допомагає відрізняти активні позиції від пасивних.

Eval = Eval0 + alpha(MobilityMAX - MobilityMIN)*

Можна також оцінювати мобільність різних типів фігур окремо:

Eval = Eval0

+ k_1*DeltaMobilityKnights

+ k_2*DeltaMobilityBishops

+ k_3*DeltaMobilityRooks

+ k_4*DeltaMobilityQueen

Фігури починають «прагнути» до активності, але для обчислення треба генерувати ходи або атаковані поля, тому функція стає дорожчою.

Сила фігур і структура пішаків. Структура пішаків істотно впливає на позиційну оцінку. Функція може містити штрафи за слабкості та бонуси за перспективні пішаки.

Отже: ізольований пішак -> штраф, здвоєні пішаки -> штраф, відсталий пішак -> штраф, прохідний пішак -> бонус, захищений прохідний пішак -> підвищений бонус, пішак біля перетворення -> додатковий бонус.

Eval = Eval0 + PawnStructureMAX - PawnStructureMIN

Так позиції стають значно осмисленішими, але треба аналізувати вертикалі, сусідні вертикалі, просування пішаків і поля блокування.

Сила фігур і безпека короля. У шахах слабка безпека короля може переважити матеріальну перевагу. Тому доцільно вводити окремий блок оцінювання королівської зони.

Отже: наявність пішакового прикриття перед королем -> бонус, відкриті лінії біля короля -> штраф, кількість атак суперника на зону короля -> штраф, король у центрі в мітельшпілі -> штраф.

$KingSafety = k_1*PawnShield - k_2*OpenFilesNearKing - k_3*EnemyAttacksNearKing$

$Eval = Eval0 + PositionalTerms + KingSafetyMAX - KingSafetyMIN$

Так програма краще уникає небезпечних позицій, але важче коректно реалізувати, особливо без урахування фази партії.

Складові функції можна комбінувати за бажанням доки швидкість обчислення складної статичної оціночної функції переважає швидкість розгортання наступного рівня дерева гри.

«Навчання» оціночної функції та проблема перемінних коефіцієнтів

Оціночну функцію можна налаштувати на оптимальні ваги. Як один з варіантів такого налаштування можна застосувати переоцінювання складових за рахунок порівняння статичної функції з результатом процедури minimax. Після накопичення достатньої кількості прикладів з різних партій та різних етапів партій (від дебюту до ендшпілю), можна побудувати принаймні регресію. Проте, це не є обов’язковим у даній лабораторній.

Проте, на прикладі шахів можна побачити, що безвідносно до того, як будувалася статична оціночна функція є певні проблеми, які неможливо усунути підбором статичних коефіцієнтів. А саме, відомо, що цінність коня на початку партії вища, а у кінці – нижча, в той час як для слона ситуація зворотна.

Будь-яка проста підгонка параметрів страждає від того, що спостережувані кореляції між чисельними значеннями не означають гарантованого причинно-наслідкового зв’язку.

І в той же час, використання перемінних коефіцієнтів може мати негативні наслідки, аналогічно до відкидання хвоста списку ходів при спрощенні обробки пізніх ходів.

Помилки, яких треба уникати

Нечітке визначення стану гри. Стан має містити всю інформацію, потрібну для вибору ходу: позиції фігур або елементів, активного гравця, доступні ходи, умови завершення, оцінку результату. Якщо частина інформації зберігається "десь окремо" або неявно, алгоритм стає важко перевірити.

Неправильне оновлення alpha і beta. Для MAX-вузла оновлюється alpha, для MIN-вузла – beta. Якщо ці значення переплутані, алгоритм може відтинати правильні гілки або не відтинати нічого.

Надмірно складна або недостатньо обґрунтована оціночна функція. Складна функція з багатьма коефіцієнтами не є безумовно кращою. Коли використовуються ваги, потрібно пояснити, чому вони саме такі, або показати експериментальне порівняння різних варіантів.

Ознаки суперечать одна одній. Наприклад, функція одночасно винагороджує агресивний наступ і пасивне збереження позиції так, що алгоритм поводиться нестабільно.

Постановка задачі

Необхідно реалізувати систему ігрового пошуку для детермінованої позиційної двоосібної антагоністичної гри з повною інформацією. Хрестики-нулики – не допускаються. У межах лабораторної роботи потрібно:

- формально описати правила обраної гри,
- подати гру як множину станів, множину допустимих ходів і функцію переходу між станами,
- відокремити термінальні стани від нетермінальних,
- визначити точну функцію корисності для термінальних позицій,
- визначити евристичну функцію оцінювання для обмеженого пошуку,

- реалізувати minimax без відтинання,
- реалізувати minimax з альфа-бета-відтинанням,
- експериментально порівняти обидва варіанти.

Практичний зміст роботи полягає не в тому, щоб виграти кілька мілісекунд, а в тому, щоб побачити механіку пошуку. Необхідно пояснити, як змінюється обсяг перебору і чому коректне альфа-бета-відтинання не змінює minimax-відповідь, а лише відкидає гарантовано непотрібні гілки.

Критично важливо відслідкувати, як поводяться alpha і beta, дуже бажано відслідкувати як відбувається відтинання (повернення евристичної оцінки без повного розгортання піддерева після досягнення обмеження за глибиною). Без спостереження є ризик не зрозуміти як насправді працює альфа-бета-відтинання.

Для практичної реалізації потрібно обрати гру не простішу за шашки. Допускається використання навчальної зменшеної модифікації, наприклад міні-шашок, якщо така модифікація зберігає чергування ходів, захоплення, нетривіальне розгалудження і потребу в оцінюванні позиції на обмеженій глибині.

Вимоги до звіту

У звіті мають бути наявні:

- тема, мета та завдання лабораторної роботи,
- короткі теоретичні відомості про minimax і альфа-бета-відтинання,
- опис обраної гри та формалізація її правил,
- опис структури стану гри,
- опис способу генерації допустимих ходів,
- опис UF та EHF,
- пояснення, як у реалізації подано MAX, MIN, обмеження глибини, alpha і beta,
- фрагменти реалізації основних частин програми,
- не менше одного наочно поданого фрагмента дерева гри,

- явне позначення місць альфа-бета-відтинання хоча б для одного прикладу,
- таблиця або інша форма порівняння minimax без відтинання і minimax з відтинанням,
- порівняння доброго і поганого порядку ходів,
- аналіз впливу глибини пошуку,
- аналіз впливу якості евристичної оцінки,
- підсумкові висновки.

Бажано також окремо подати кількість переглянутих вузлів у кожному експерименті, кількість відтинань, приклади позицій, де HEF давала хибно оптимістичний або хибно песимістичний результат, а також коротке обговорення того, коли ітеративне поглиблення або таблиця вже оцінених позицій можуть бути корисними.

Контрольні запитання

1. Які ігри називають двоосібними антагоністичними іграми з повною інформацією?
2. Що таке дерево гри?
3. Яка роль гравців MAX і MIN у minimax-пошуку?
4. Чим термінальний вузол відрізняється від проміжного?
5. Що таке функція корисності?
6. Чим оціночна функція відрізняється від функції корисності?
7. У чому полягає ідея minimax-рекурсії?
8. Навіщо обмежувати глибину пошуку?
9. Що таке фактор галудження і як він впливає на складність перебору?
10. Що таке ефект обрію?
11. Що означає параметр alpha?
12. Що означає параметр beta?
13. За якої умови відбувається альфа-бета-відтинання?
14. Чому альфа-бета-відтинання не змінює minimax-відповідь?
15. Чому порядок розгляду ходів впливає на кількість відтинань?

16. Що таке впорядкування ходів?
17. Для чого використовується ітераційне поглиблення?
18. У чому ідея таблиці перестановок?
19. Чому недостатньо просто запустити minimax, не аналізуючи структуру дерева?
20. Які характеристики оціночної функції є критичними для практичного пошуку?

Критерії оцінювання

Оцінювання лабораторної роботи здійснюється за такими рівнями:

На 100%. Обрану гру формалізовано коректно, стан гри, генератор ходів і переходи між станами реалізовано правильно, minimax без відтинання та minimax з альфа-бета-відтинанням працюють коректно, наведено переконливе порівняння кількості переглянутих вузлів, для щонайменше однієї позиції побудовано фрагмент дерева гри та правильно пояснено місця відтинання, проаналізовано вплив порядку ходів, обмеження глибини і статичну оціночну функцію, звіт повний, структурований і містить змістовні висновки.

На 70%. Базову реалізацію гри та обидва варіанти пошуку виконано загалом коректно, порівняння без відтинання і з ним наявне, але не всюди достатньо глибоке, дерево гри або приклади відтинання подано спрощено, вплив глибини чи порядку ходів проаналізовано частково, звіт містить основні елементи, але частина пояснень поверхова.

На 50%.Реалізовано лише частину потрібної функціональності, пошук працює нестабільно або не для всіх тестових позицій, альфа-бета-відтинання реалізовано формально, але без переконливого пояснення, порівняльний аналіз мінімальний, звіт неповний і містить лише фрагментарний опис результатів.

У разі, коли завдання виконано менш ніж на 50%, лабораторна робота вважається виконаною на 0.

Лабораторна робота 3. Евристичний пошук та інтелектуальні агенти

Лабораторна робота присвячена простому евристичному пошуку та моделюванню поведінки інтелектуальних агентів у дискретному середовищі на прикладі на прикладі Pac-Man-подібного навчального лабіринтного середовища. У межах роботи розглядаються простір станів, функція переходів, цілі та евристики, а також практична реалізація й порівняння стратегій поведінки агентів-переслідувачів. Окрему увагу приділено дослідженню того, як із простих локальних правил і взаємодії кількох агентів може виникати складна емерджентна поведінка [12, 15, 16].

Мета роботи. Ознайомитися з принципами простого евристичного пошуку та побудови інтелектуальних агентів у дискретному ігровому середовищі; навчитися реалізовувати та порівнювати прості й складні стратегії поведінки ботів; дослідити, як із набору локальних правил і евристик виникає емерджентна групова поведінка агентів; проаналізувати вплив різних стратегій переслідування на результативність вловлювання керованого агента (гравця) та на сприйняття геймплею.

Умови використання Pac-Man-подібного навчального середовища

У цій лабораторній роботі Pac-Man використовується лише як відома навчальна модель лабіринтного багатоагентного середовища для дослідження евристичного пошуку, агентної поведінки, переслідування, перехоплення та емерджентної поведінки. Офіційна гра Pac-Man, назва, персонажі, графіка, звуки, логотипи та інші фірмові елементи належать правовласникам; офіційний сайт позначає PAC-MAN як “PAC-MAN™ & © Bandai Namco Entertainment Inc.”

У межах роботи дозволяється використовувати лише навчальне Pac-Man-подібне середовище, зокрема UC Berkeley Pac-Man Projects, або власну спрощену реалізацію без оригінальних ресурсів комерційної гри. У разі використання матеріалів Berkeley потрібно зберігати атрибуцію, не поширювати й не публікувати розв’язки, не перепаковувати та не

перевидавати матеріали без погодження з авторами проєкту. Berkeley прямо дозволяє використання цих проєктів для освітніх або особистих цілей за таких умов [18].

Студентам забороняється використовувати оригінальні ROM-файли, спрайти, музику, звуки, логотипи, зображення персонажів або інші матеріали офіційної гри Pac-Man. У власній реалізації слід відтворювати тільки абстрактну постановку задачі: лабіринт, керованого агента, агентів-переслідувачів, цілі, перешкоди, винагороди, функцію переходів і правила вибору дій. Ідея гри та методи гри самі по собі не охороняються авторським правом так, як конкретне графічне, текстове чи звукове вираження.

Інструментарій та програмне забезпечення

Для виконання лабораторної роботи студент може використати навчальне Pac-Man-подібне середовище, зокрема UC Berkeley Pac-Man Projects, або власну спрощену реалізацію без оригінальних ресурсів комерційної гри, оскільки головна задача полягає в розробці логіки поведінки агентів-переслідувачів. Але можна самостійно створити спрощене лабіринтне агентне середовище.

Рекомендований інструментарій:

- мова програмування: C++ або C#,
- бібліотеки для 2D-графіки або ігрового циклу: для C++ – SFML, SDL2, Raylib, Allegro або інші бібліотеки для роботи з вікном, подіями, простими геометричними об’єктами й ігровим циклом, для C# – Unity, MonoGame, Godot C#, Raylib-cs, WinForms/WPF з власною реалізацією циклу оновлення або інші аналогічні засоби,
- засіб фіксації результатів: CSV, JSON, текстовий журнал подій, вбудований лог симуляції, серіалізація результатів запуску або табличний процесор для подальшого аналізу,
- засоби побудови графіків і візуалізації результатів: для C++ – Matplot++, gnuplot, matplotlib-cpp або експорт у CSV з подальшою побудовою графіків

в Excel; для C# – ScottPlot або експорт у CSV/JSON з подальшою побудовою графіків в Excel.

За потреби можна використати іншу мову програмування та/або інші засоби, але тоді студент має обґрунтувати причину цього рішення та отримати дозвіл перед початком виконання роботи.

Теоретичні відомості

Інтелектуальний агент – це сутність, що сприймає стан середовища, обирає дію та змінює середовище відповідно до певної мети. У контексті гри агентами можуть бути керований агент і агенти-переслідувачі. Для цієї лабораторної роботи основний об’єкт дослідження – поведінка агенти-переслідувачі як множини агентів у спільному середовищі [12,15]. Простором станів є множина можливих конфігурацій ігрового середовища.

Простір станів у грі

Гра природно задає дискретний простір станів. Стан може включати:

- позицію керованого агента,
- позиції агентів-переслідувачів,
- напрямки їх руху,
- конфігурацію лабіринту,
- наявність або відсутність їжі, бонусів, енергетичних кульок,
- режим гри (звичайний, вразливість агентів-переслідувачів тощо).

Розширений простір станів включає внутрішні (часто – не спостережувані ззовні) стани агентів.

Функція переходів – правила зміни стану внаслідок дій агентів. Перехід між станами здійснюється внаслідок дій агентів: руху вгору, вниз, ліворуч, праворуч або збереження напряму на наступному такті [15,16].

Ціль – бажаний результат поведінки агента (наприклад, пошуку).

Евристики

Евристика – це правило вибору дії, яке не гарантує оптимального результату, але дозволяє ефективно діяти в умовах обмежених ресурсів. У цій лабораторній роботі евристики застосовуються для вибору напряму руху агента-переслідувача. Приклади евристик [12, 15, 16]:

- зменшення мангеттенської відстані до керованого агента,
- вибір маршруту, який перетинає очікуваний шлях керованого агента,
- уникання скупчення агентів-переслідувачів в одному місці,
- блокування шляхів відступу,
- розподіл ролей між агентами-переслідувачами.

Пошук шляху

Для реалізації переслідування можуть застосовуватись алгоритми пошуку на графі: BFS, DFS, greedy best-first search, A*. У контексті гри особливо доречні BFS або A* для знаходження найкоротшого маршруту в лабіринті. Якщо мета полягає не лише в прямому наближенні, а й у перехопленні, тоді цільова клітинка може визначатися евристично, а сам маршрут до неї шукатися стандартним алгоритмом [17].

Реактивна та кооперативна поведінка

Найпростіший агент реагує лише на поточний стан середовища і не враховує інших агентів. Складніша поведінка виникає тоді, коли агент [15, 16]:

- враховує майбутній рух цілі;
- враховує дії інших агентів,
- змінює свою роль залежно від ситуації,
- передає або використовує спільну інформацію.

Саме в цей момент система з кількох простих агентів може демонструвати емерджентні властивості: оточення, витіснення в коридор, пастки, розділення зон контролю, непередбачувано ефективне переслідування без жорстко прописаного глобального сценарію.

Емерджентна поведінка

Емерджентна поведінка – це така поведінка системи, яка виникає як результат взаємодії простих компонентів, але не зводиться прямо до одного окремого правила. У грі вона може проявлятися як [15, 16]:

- спонтанне оточення керованого агента,
- поділ ролей між агентами-переслідувачами без явно запрограмованого "командира",
- блокування проходів,
- загін керованого агента у тупик,
- посилення тиску на гравця навіть за використання простих локальних евристик.

Вплив на геймплей

Зміна поведінки ботів змінює як їхню результативність, так і стиль гри. Просте випадкове блукання створює хаотичний і часто поблажливий режим. Пряме переслідування робить гру передбачуваною. Координована багатоагентна поведінка створює напругу, примушує гравця планувати маршрут, змінює темп руху, частіше провокує ризикові рішення та підсилює відчуття «розумного» супротивника [12, 15, 16].

Хорошим прикладом є різниця у результативності вловлювання керованого агента залежно від набору використаних правил. Нехай є два варіанти правил для агентів-переслідувачів: випадкове блукання та переслідування по найкоротшому шляху та вважається, що керований (або автоматичний) агент завжди діє оптимально. На рис. 2 зображено два стани гри. При використанні лише випадкового блукання у першому стані агенти-переслідувачі можуть впустити керованого агента. Проте, при використанні лише переслідування у другому стані агенти-переслідувачі ніколи не впіймають керованого агента. Оптимальною є комбінація обох правил, по одному на агента-переслідувача, що гарантує вловлювання керованого агента у наведеному лабіринті незалежно від початкового стану, першого чи другого.

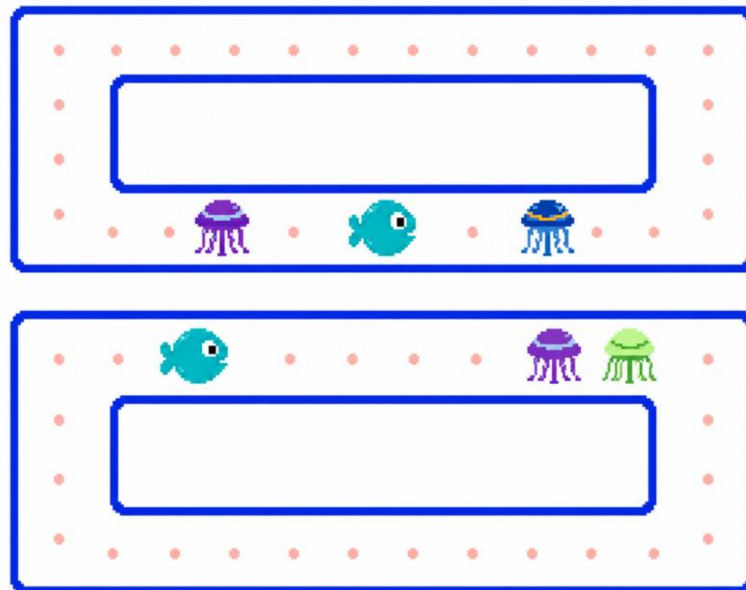

Рис.2 Стани гри для демонстрації емерджентної поведінки.

Помилки, яких треба уникати

Змішування стану і візуального відображення. Карта на екрані – це не те саме, що модель стану. Потрібно мати внутрішнє подання, з яким працює алгоритм: матриця, граф, список вузлів, координати тощо.

Відсутність чіткої функції переходу. Функція переходу має відповідати на питання: який новий стан виникає після певної дії. Якщо перехід реалізовано неявно або розкидано по коду, поведінку агентів складно аналізувати.

Відсутність обробки тупиків і недосяжних цілей. Алгоритм має коректно працювати, якщо шлях заблокований, ціль тимчасово недосяжна або є кілька рівноцінних маршрутів.

Постановка задачі

Необхідно використати навчальне Pac-Man-подібне середовище або самостійно створити спрощене лабіринтне агентне середовище без оригінальних ресурсів комерційної гри та реалізувати систему поведінки кількох агентів-переслідувачів із різними локальними евристиками та ролями. Основна мета – показати, як від найпростіших стратегій до складніших, на

основі відносно простих правил і евристик, виникає складна групова поведінка ботів, що підвищує ефективність вловлювання керованого агента.

Мінімальні функціональні вимоги до програмної реалізації:

- лабіринт із прохідними й непрохідними клітинками,
- керований агент,
- чотири агенти-переслідувачі,
- покроковий або квазібезперервний рух по сітці,
- зіткнення,
- можливість запускати різні режими поведінки агентів-переслідувачів, наприклад, переслідування передбачає використання або самостійну реалізацію алгоритмів пошуку на графі, зокрема BFS та A*,
- можливість збирати статистику за серією запусків.

У межах лабораторної роботи потрібно побудувати чотири етапи ускладнення поведінки агентів-переслідувачів. Ускладнення має починатися зі сценарію, де агенти-переслідувачі не спілкуються між собою і не враховують поведінку один одного, і завершуватися сценарієм, де вони і спілкуються, і слідкують один за одним, тобто використовують спільну інформацію та коригують власні дії залежно від дій інших агентів-переслідувачів. На кожному етапі потрібно реалізувати кілька кроків наростання складності за рахунок порівняно простих евристик. Рекомендується наступна схема.

Етап 1. Незалежні агенти-переслідувачі без координації та без взаємного врахування

На цьому етапі кожен агент-переслідувач діє самостійно й не використовує інформацію про інших агентів-переслідувачів.

Можливі види незалежної поведінки:

- Випадкове блукання: на розвилці агент-переслідувач випадково обирає один із допустимих напрямків.

- Локальне пряме переслідування: агент-переслідувач зменшує відстань до керованого агента за найпростішою евристикою.
- Переслідування через пошук маршруту: агент-переслідувач обирає найкоротший шлях до поточної позиції керованого агента.

Можна запропонувати свою, бажано, просту.

Очікуваний результат: покращення індивідуального переслідування, але відсутність справжньої командної роботи.

Етап 2. Незалежні агенти-переслідувачі без спілкування, але з непрямою спеціалізацією

Агенти-переслідувачі все ще не обмінюються інформацією і не узгоджують план, але отримують різні локальні евристики.

Можливі види поведінки:

- один агент-переслідувач переслідує поточну позицію керованого агента,
- другий намагається рухатися до прогнозованої позиції керованого агента на кілька кроків уперед,
- третій патрулює область перед керованим агентом з урахуванням напряму його руху,
- четвертий обирає маршрут, який потенційно відрізає шлях до найближчого виходу або перехрестя.

Очікуваний результат: без явної координації виникає частковий розподіл ролей і зростає ймовірність перехоплення.

Етап 3. Агент-переслідувачі спілкуються, але ще не моделюють поведінку один одного повноцінно

Агенти-переслідувачі отримують доступ до спільної інформації: поточної позиції керованого агента, прогнозованої цілі, відстаней або пріоритетних зон. Проте вони ще не здійснюють повноцінний взаємний контроль розташування.

Можливі види поведінки:

- Спільна ціль: один модуль визначає цільові точки, а агенти-переслідувачі обирають собі одну з них.
- Розподіл зон переслідування: кожному агенту-переслідувачу задається свій сектор або тип цілі.
- Елементарне уникання дублювання: якщо два агенти-переслідувачі рухаються до однієї точки, один змінює пріоритет.
- Колективне блокування: один агент-переслідувач тисне ззаду, інші займають потенційні шляхи відступу.

Очікуваний результат: виникає виразніша командна поведінка, керований агент (гравець) частіше опиняється під багатостороннім тиском.

Етап 4. Агенти-переслідувачі і спілкуються і слідкують один за одним

На цьому етапі реалізується найскладніша модель. Агенти-переслідувачі використовують спільну інформацію та коригують поведінку залежно від положення, напряму й ролей інших агентів-переслідувачів.

Можливі види поведінки:

- Уникання скупчення: агенти-переслідувачі штрафують вибір маршрутів, які ведуть до надмірної концентрації в одній зоні.
- Динамічний розподіл ролей: ролі переслідувач, перехоплювач, блокувальник, резервний агент змінюються залежно від ситуації.
- Побудова пасток: агенти-переслідувачі узгоджено займають ключові перехрестя та тупики.
- Адаптація до геймплею: поведінка змінюється залежно від стилю руху керованого агента, наприклад переваги коротких маршрутів, уникання центру чи тяжіння до бонусів.

Очікуваний результат: з набору локальних правил виникає складна групова поведінка, яку гравець сприймає як “розумну” та значно небезпечнішу.

Так чи інакше, у грі кожна евристика має бути прив’язана до конкретного механізму: вибір напряму руху, оновлення цільової клітини,

штраф за близькість до інших агентів-переслідувачіи, бонус за перекриття шляхів керованого агента, обмін інформацією про останню відому позицію керованого агента, прогноз руху на k кроків, перемикання ролей тощо.

Експериментальна частина

У лабораторній роботі необхідно показати, як зміна поведінки агентів-переслідувачів впливає на геймплей. Для цього слід провести серію запусків і суб’єктивно оцінити напруженість та передбачуваність гри, оскільки керований агент керується користувачем.

Викладач може вибірково перевірити будь-який із чотирьох етапів або їх комбінацію і студент повинен бути готовий продемонструвати всі чотири етапи та пояснити їх змістовні відмінності

Для кожного етапу, що перевіряється студент повинен бути у змозі коротко пояснити, які саме правила або евристики реалізовано, показати роботу режиму в програмі, пояснити, чим цей етап відрізняється від попереднього у розглянутій реалізації.

Для звіту необхідно перевіряти такі властивості:

- успішність спіймань керованого агента,
- наявність випадків блокування шляху,
- наявність ситуацій, у яких керованого агента загнано в тупик,
- для суб’єктивного аналізу – оцінка передбачуваності та напруженості гри.

Студент повинен бути здатен навести не менше двох конкретних прикладів ситуацій, у яких поведінка агентів-переслідувачів має емерджентний характер. Для кожного прикладу потрібно показати ситуацію на екрані та які локальні правила діяли, пояснити, чому отриманий результат не зводиться до одного простого правила окремого агента-переслідувача.

Вимоги до звіту

У звіті необхідно подати:

- Тему, мету та завдання лабораторної роботи.

- Короткі теоретичні відомості про евристичний пошук, агентів і емерджентну поведінку.
- Короткий опис обраної реалізації гри.
- Опис структури програми та способу подання лабіринту.
- Опис усіх реалізованих стратегій агентів-переслідувачів.
- Окремий опис чотирьох етапів ускладнення поведінки.
- Обґрунтування використаних евристик.
- Скриншоти, фрагменти коду, блок-схеми або псевдокод основних механізмів.
- Таблицю порівняння результатів експериментів.
- Аналіз того, які властивості поведінки можна вважати емерджентними.
- Аналіз впливу реалізованих стратегій на геймплей.
- Підсумкові висновки.

Бажано, щоб у звіті були окремо виділені:

- об'єктивні метрики,
- суб'єктивні враження від гри на кожному етапі,
- приклади ситуацій, де прості правила призвели до несподівано складної поведінки.

Контрольні запитання

1. Що таке інтелектуальний агент?
2. Що розуміють під простором станів?
3. Яку роль виконує евристика в задачах пошуку?
4. Чим евристичний пошук відрізняється від повного перебору?
5. Які алгоритми пошуку шляху доцільно використовувати в лабіринті та чому?
6. Що таке реактивна поведінка агента?
7. Що таке багатоагентна система?
8. Чим відрізняється незалежна поведінка агентів від координованої?
9. Що таке емерджентна поведінка?

10. Які ознаки емерджентної поведінки можна спостерігати у Pac-Man-подібній грі?
11. Чому навіть прості локальні правила можуть породжувати складну групову поведінку?
12. Як впливає обмін інформацією між агентами на ефективність переслідування?
13. Навіщо агентам-переслідувачам враховувати положення один одного?
14. Які метрики можна використати для оцінювання ефективності поведінки агенти-переслідувачі?
15. Як зміна стратегій ботів впливає на геймплей і досвід гравця?
16. Чому пряме переслідування не завжди дає найкращий результат?
17. У чому полягає різниця між переслідуванням і перехопленням?
18. Які переваги й обмеження має модель, у якій агенти-переслідувачі координуються?
19. Як можна ускладнити поведінку агентів-переслідувачів без використання складних алгоритмів машинного навчання?
20. Які висновки про природу інтелектуальної поведінки агентів дозволяє зробити ця лабораторна робота?

Критерії оцінювання

Оцінювання лабораторної роботи здійснюється за такими критеріями:

На 100%. Роботу виконано повністю. Реалізовано базове середовище та всі чотири етапи ускладнення поведінки агентів-переслідувачів. Поведінка агентів помітно змінюється від етапу до етапу. Проведено серію експериментів, наведено метрики, виконано порівняння результатів. У звіті є переконливий аналіз емерджентної поведінки та впливу стратегій на геймплей.

На 80%. Роботу виконано добре, але є окремі недоліки. Реалізовано більшість вимог, проте один із етапів ускладнення подано спрощено або

недостатньо відмежовано від інших. Експериментальна частина наявна, але аналіз результатів не всюди достатньо глибокий.

На 60%. Роботу виконано на базовому рівні. Є працездатна реалізація гри та принаймні кілька режимів поведінки агентів-переслідувачів, але порівняння у звіті між ними поверхове, а різниця в реалізації - мінімальна. Емерджентні властивості лише заявлені, але не доведені прикладами або даними. Звіт неповний.

На 50%. Робота містить лише часткову реалізацію. Базове середовище або логіка агентів працює нестабільно. Лише один режим поведінки. Звіт неповний.

У разі, коли завдання виконано менш ніж на 50% лабораторна вважається виконаною на 0.

Лабораторна робота 4. Генетичні алгоритми та оптимізація

Лабораторна робота присвячена генетичним алгоритмам як засобу еволюційної оптимізації в задачах, де простір пошуку є складним, багатомодальним, негладким, розривним або дискретним. У межах роботи студент реалізує базовий генетичний алгоритм, порівнює його поведінку на послідовності тестових задач, щоб обґрунтовано встановити, у яких випадках популяційний пошук є методично виправданим [19-22].

Мета роботи. Ознайомитися з принципами еволюційної оптимізації та набути практичних навичок реалізації генетичного алгоритму для різних типів цільових функцій. Навчитися задавати кодування особин, функцію пристосованості, оператори селекції, схрещування та мутації, а також аргументовано оцінювати результати роботи генетичного алгоритму залежно від властивостей задачі.

У межах цієї лабораторної роботи важливо:

- зрозуміти, чому генетичні алгоритми особливо доречні для негладких, багатомодальних і дискретних просторів пошуку,
- побудувати й порівняти кілька схем кодування хромосоми,

- навчитися працювати з обмеженнями через штрафні функції та розрізняти жорсткі й м’які обмеження,
- дослідити вплив параметрів генетичного алгоритму на швидкість збіжності, стабільність і якість отриманих розв’язків,
- навчитися інтерпретувати результат не лише як «найкраще знайдене значення», а як наслідок обраного кодування, операторів і властивостей задачі.

Інструментарій та програмне забезпечення

Для виконання лабораторної роботи рекомендовано використати таку базову збірку програмних засобів:

- C++ або C#,
- GeneticSharp як фреймворк для швидкого прототипування генетичних алгоритмів у середовищі C# або відповідні аналоги для C++,
- C# зручним варіантом є ScottPlot, у C++ можна застосувати його аналоги, експорт даних у CSV з подальшою побудовою графіків у табличному процесорі.

За потреби можна використати іншу мову програмування та/або інші засоби, але тоді студент має обгрунтувати причину цього рішення та отримати дозвіл.

Теоретичні відомості

Генетичний алгоритм є популяційним методом пошуку, у якому замість одного поточного наближення одночасно розглядається множина кандидатів-розв’язків. Ефективність генетичного алгоритму істотно залежить від задачі. Його сильні сторони проявляються тоді, коли простір пошуку має багато локальних мінімумів, містить розриви, є дискретним або включає складні поєднання обмежень, через що градієнтні підходи втрачають природність або застосовність [19-21].

Кожен розв’язок-кандидат подається як хромосома, що складається з генів і кодує значення змінних. Сукупність особин утворює популяцію, якість

кожної особини оцінюється функцією пристосованості, а поточний розв'язок – як найкраща пара із хромосоми та її значення функції пристосованості.

У класичній схемі роботи генетичного алгоритму популяція проходить через послідовність поколінь. На кожному кроці виконується оцінювання пристосованості, селекція батьківських особин, схрещування, мутація й формування нового покоління. Селекція віддає перевагу кращим особинам. Схрещування комбінує фрагменти батьківських рішень, а мутація вносить випадкові зміни, що підтримують дослідження простору пошуку. Критерії зупинки зазвичай пов'язують із досягненням заданої кількості поколінь, відсутністю помітного покращення або досягненням цільового значення функції [19-21].

Окрему увагу слід приділити штрафним функціям для роботи з обмеженнями. Якщо задача містить недопустимі конфігурації, то замість жорсткого виключення всіх таких особин іноді доцільно збільшувати значення функції пристосованості на величину штрафу. У найпростішому випадку загальна оцінка має вигляд суми якості розв'язку та штрафів за порушення обмежень. При цьому жорсткі обмеження мають отримувати великі штрафи, щоб алгоритм у першу чергу шукав допустимі рішення, а м'які – менші штрафи, які дозволяють відрізняти «кращі» і «гірші» допустимі або майже допустимі варіанти [19-23].

Необхідно зазначити, що в цій роботі задача відбору ознак або генів не розглядається поглиблено, бо вже зафіксовано задачі та ключові характеристики. Див. Постановка задачі. Проте, в загальному випадку від її розв'язку залежить ефективність роботи ГА. Також конкретний вибір залежить від набору даних. Він має бути методично обґрунтованим до того, як буде вжито ГА [22,23].

Види кодування та відповідні мутації

Бінарне кодування. Хромосома – послідовність 0 і 1. Приклад: 10110010. Зручне для задач вибору, відбору ознак, логічних рішень "так/ні".

Просте для мутації й кросоверу, але не завжди природне для числових задач. Має бітову мутацію (bit-flip). Для бінарної хромосоми окремі біти змінюються з 0 на 1 або з 1 на 0. Іноді можуть бути вжиті мутації як для перестановкового кодування [20,21].

Цілочисельне кодування. Хромосома складається з цілих чисел. Приклад: [3, 1, 5, 2]. Підходить для розкладів, призначень, кластеризації, коли ген має набувати значень із дискретної множини. Мутація працює як випадкове перевстановлення значення: ген отримує нове допустиме значення з домену. Також вживається крокова мутація. Значення змінюється на малий інкремент, а не на довільне нове число. Приклад: $x = 4$ переходить у $x = 5$ або $x = 3$. Вона корисна для дискретизованих параметрів, коли потрібне акуратне локальне доопрацювання.

Дійсновекторне кодування. Хромосома – вектор дійсних чисел. Приклад: [1.25, -0.7, 4.9]. Зручне для оптимізації числових параметрів і неперервних функцій. Зазвичай краще за бінарне, коли змінні природно є числовими. Для нього часто вживається гаусова мутація. До дійсного гена додається випадковий шум, зазвичай з нормального розподілу. Приклад: вектор параметрів [0.8, -1.2] може перейти в [0.73, -1.05].

Перестановкове кодування. Хромосома – перестановка елементів без повторів. Приклад: [4, 2, 1, 5, 3]. Використовується в задачах маршрутизації, порядку виконання, комівояжера, розкладу. Має декілька варіантів мутації:

- Обмін двох позицій (swap mutation). Два гени міняються місцями. Приклад: у перестановці [A, B, C, D, E, F] після мутації можна отримати [A, C, B, D, E, F]. Причому мінятися місцями можуть і не сусіди. Такий тип природний для маршрутів і впорядкувань.
- Інверсія фрагмента (inversion mutation). Частина хромосоми розвертається у зворотному порядку. Приклад: [A, B, C, D, E, F] при інверсії від A до E дає [E, D, C, B, A, F]. Це корисно, коли відносний порядок елементів важливий.

- Перемішування фрагмента (scramble mutation). Усередині вибраного підрядка порядок елементів випадково перемішують. Приклад: [A, B, C, D, E, F] може перейти в [E, C, B, D, A, F]. На відміну від інверсії, новий порядок не є дзеркальним.

Деревоподібне кодування. Хромосома має структуру дерева, а не вектора. Приклад: дерево арифметичного виразу. Типове для генетичного програмування, де еволюціонують формули, правила або програми. Мутація полягає в застосуванні операцій зміни дерева [19,21]. В даній лабораторній бажано не використовувати.

Змішане кодування. Кожна ознака або ген може мати своє кодування. У загальному випадку це означає потребу згрупувати однаково закодовані гени у групи, і вживати кросовер у межах груп, можливо навіть різний кросовер.

Найпростіше правило вибору таке: форма хромосоми має бути максимально природною для задачі.

Типи кросоверу

Кросовер може виконуватися як у межах хромосоми, так і у межах гена, якщо той досить складний [20,21].

Одноточковий кросовер. У двох батьків вибирається одна точка розрізу, після чого хвости обмінюються. Приклад: 110|010 та 001|111 дають 110111 і 001010. Це базовий варіант для бінарного або цілочисельного кодування.

Двоточковий кросовер. Обмінюється сегмент між двома точками. Приклад: 11|001|0 та 00|111|1 дають 111110 і 000010. Такий кросовер краще зберігає зовнішні частини хромосоми.

Однорідний кросовер. Для кожного гена окремо вирішується, від якого з батьків він успадковується. Приклад: за маскою 10110 із батьків 11001 і 00111 можна отримати 11111. Він посилює перемішування, але сильніше руйнує зв’язані блоки генів.

Арифметичний кросовер. Для дійсних векторів нащадок утворюється як середнє або зважена комбінація батьків. Приклад: для батьків x = [2.0, 5.0] і y = [4.0, 1.0] нащадок при зважуванні $\alpha = 0.25$ дорівнює [3.5, 2.0]. Цей вид має використовуватися обмежено, оскільки може загнати розв'язок близько до центру області визначення, особливо за слабкої мутації.

Інтервальний кросовер (з параметром α). Для кожного гена нове значення вибирають з інтервалу, трохи ширшого за відрізок між батьками. Приклад: якщо батьківські значення гена 3 і 6, то при $\alpha = 2.5$ допустимий інтервал становить [0.5; 8.5]. Це дає як змішування, а і керовану екстраполяцію.

Імітація бінарного. Імітує поведінку одноточкового кросоверу для дійсних чисел і часто використовується у неперервній оптимізації. Практично він породжує нащадків, що зазвичай лежать близько до батьків, але інколи виходять далі для посилення дослідження простору.

Частково відображений кросовер. Призначений для перестановок і зберігає унікальність елементів. Приклад: для маршрутів або порядків робіт одна й та сама вершина після кросоверу не повинна з'явитися двічі.

Порядковий кросовер. Нащадок успадковує підрядок одного з батьків, а решта елементів заповнюється в порядку появи з другого. Це корисно, коли важливий саме відносний порядок, а не значення позицій як такі.

Як і у випадку мутації, кросовер слід узгоджувати з поданням особини. Бінарним хромосомам природні одноточковий, двоточковий і однорідний кросовер; дійсновекторним – арифметичний, інтервальний кросовер з параметром або імітація бінарного; перестановкам – порядковий кросовер та частково відображений кросовер. Неправильний вибір частіше створює формально некоректних нащадків або руйнує цінні структурні блоки [20, 21].

Евристики керування пошуком і популяцією

Пріоритет допустимості. Спочатку порівнюються особини за кількістю і вагою порушених жорстких обмежень, а вже потім – за цільовою якістю.

Приклад: у задачі розкладу розклад без конфліктів викладачів слід вважати кращим за розклад із меншою кількістю «вікон», але з конфліктами аудиторій [21-23].

Адаптивний штраф. Вага штрафів змінюється в ході еволюції. Якщо популяція довго не входить у допустиму область, штрафи жорстких обмежень збільшують, якщо допустимі особини вже переважають, можна посилити вагу м'яких обмежень.

Евристика «ремонт-після-мутації». Після схрещування або мутації невдалу особину не відкидають одразу, а локально ремонтують. Приклад: якщо два заняття однієї групи потрапили в один слот, декодер переносить одне з них у найближчий допустимий час. Така евристика особливо корисна для розкладу й інших задач із жорсткими обмеженнями.

Локальний прогноз корисності гена. Для бінарного відбору ознак можна вводити додаткове правило: рідко вимикати ознаки, які стабільно входять до хороших рішень, і частіше вмикати ознаки, що вже давали приріст якості в попередніх поколіннях.

Перезапуск після стагнації. Якщо найкраще значення не змінюється протягом заданого числа поколінь, частину популяції ініціалізують заново

Меметична домішка локального пошуку. Після глобального пошуку ГА до найкращих особин застосовують просту локальну доробку [21].

Елітизм. Найкращі особини переносяться в нове покоління без змін. Елітизм пришвидшує збіжність, але в надлишку підсилює передчасне виродження [19-21].

Проріджування згущень. Якщо в малому околі простору накопичується забагато дуже схожих особин, залишають одного або кількох представників околу, а іншим зменшують шанси вижити. Корисно у випадку обмеженої пам'яті та функції з багатьма екстремумами. Є декілька варіацій [21]:

– Детерміноване проріджування. Нащадок змагається не з усією популяцією, а з батьком або сусідом вибраним за якимось правилом. Така схема добре зберігає локальні ніші.

– Послаблене проріджування. Близькі розв'язки не зобов'язані одразу виштовхувати один одного, дозволяється короткочасне співіснування кількох дуже схожих особин у малій околиці. Така схема корисна тоді, коли функція пристосованості має різкі перепади на малих змінах генів. Приклад: для бінарних ознакових векторів [1,1,1,0,0] і [1,1,1,1,0] одна додаткова ознака може різко змінювати якість класифікації, якщо одразу «прорідити» околицю занадто агресивно, алгоритм втратить можливість акуратно дослідити локальний розрив. Іноді це супроводжують усередненням значень функції пристосованості.

Вікова заміна. На противагу чистому елітизму, частина популяції оновлюється за віком. Приклад: особини, що прожили 10 поколінь, видаляються навіть за прийнятної якості, щоб не блокувати нові лінії пошуку.

Острівна модель. Популяцію розбивають на кілька субпопуляцій з рідкісною міграцією. Приклад: один «острів» працює з підвищеною мутацією, інший – з сильнішим елітизмом, а кожні N поколінь найкращі дві особини обмінюються між островами. Це простий спосіб підтримувати різні режими пошуку [21].

Помилки, яких треба уникати

Мутація руйнує структуру розв'язку. Наприклад, для перестановкового кодування не можна застосовувати довільну заміну гена, якщо вона створює дублікати або втрату елементів. Для кожного типу кодування потрібні відповідні оператори мутації.

Неправильний масштаб штрафів. Якщо штраф за дрібне порушення більший, ніж за критичне, алгоритм оптимізує не те, що потрібно. Наприклад, порушення аудиторії або одночасного перебування викладача у двох місцях має штрафуватися суттєвіше, ніж небажаний час заняття.

Помилки налаштування генетичного алгоритму.

- Передчасна збіжність. Популяція швидко стає одноманітною, і алгоритм перестає знаходити нові розв'язки. Причини: надто сильна селекція, мала популяція, низька мутація, надмірний елітизм.
- Незбіжність. Якщо мутація занадто сильна, алгоритм перетворюється на випадковий пошук і не накопичує корисні часткові рішення.

Постановка задачі

Необхідно реалізувати генетичний алгоритм і дослідити його поведінку. Метою є як отримання робочої реалізації, так і експериментальне обґрунтування того, що популяційний пошук має найбільшу цінність тоді, коли задача погано узгоджується з класичною гладкою оптимізацією. Необхідно продемонструвати контраст.

Необхідно реалізувати генетичний алгоритм для двох класів цільових функцій і задач:

- розривна або негладка функція кількох змінних, для якої класичний градієнтний підхід працює незручно або нестабільно (на вибір студента),
- дискретна задача з природним бінарним, цілочисельним або перестановковим кодуванням (у цій лабораторній – розклад).

Для кожної тестової задачі слід:

- задати обране кодування хромосоми,
- визначити функцію пристосованості,
- реалізувати оператори селекції, схрещування та мутації,
- реалізувати жадібний варіант для порівняння з дискретною задачею та варіант, що використовує градієнтний спуск для дійсної задачі,
- провести серію запусків і порівняти швидкість збіжності, стабільність результату та чутливість до параметрів.

Основною прикладною задачею лабораторної роботи є складання розкладу. У межах цієї задачі в хромосомі потрібно кодувати призначення

занять, викладачів, аудиторій і часових слотів або іншу еквівалентну структуру, яка однозначно декодується в розклад. Обов'язково потрібно розвести жорсткі та м'які обмеження. До жорстких обмежень належать конфлікти аудиторій, викладачів і груп, порушення обов'язкових часових заборон, недопустимі типи аудиторій тощо. До м'яких обмежень можуть належати небажані «вікна», надмірна концентрація занять у певні дні, нерівномірний розподіл навантаження та інші фактори якості розкладу.

Для задачі розкладу функція пристосованості має бути складеною й містити великий штраф за жорсткі обмеження та менші штрафи за м'які обмеження. У звіті необхідно подати і найкращу особину, і розшифровку штрафів за кожний клас обмежень. Бажано також показати, як змінювалася частка допустимих особин у популяції впродовж еволюції.

У межах лабораторної роботи необхідно порівняти щонайменше три конфігурації параметрів генетичного алгоритму та три набори вхідних даних. Доцільно змінювати розмір популяції, імовірність схрещування, імовірність мутації, перелік використаних евристик та число поколінь. Порівняння має бути однаковим для всіх варіантів вхідних даних задачі, щоб можна оцінити як абсолютний результат, так і чутливість алгоритму до налаштувань.

Вимоги до звіту

У звіті мають бути наявні:

- тема, мета та завдання лабораторної роботи,
- короткі теоретичні відомості про генетичні алгоритми,
- опис постановки задачі складання розкладу, перелік жорстких і м'яких обмежень, а також структуру штрафної функції,
- опис використаного інструментарію та обґрунтування вибору бібліотек,
- опис реалізації генетичного алгоритму, у тому числі схема побудови інструментарію, вибрані оператори й логіка запуску експериментів,
- опис усіх варіантів функції якості використаних у лабораторній роботі,
- обґрунтування обраного кодування хромосом,

- опис щонайменше трьох конфігурацій параметрів генетичного алгоритму та трьох наборів вхідних даних,
- результати серії запусків для тестових функцій, подані у вигляді таблиць і графіків збіжності,
- приклади найкращого знайденого розкладу (для кожного набору даних) та розшифровку штрафів за кожним класом обмежень,
- порівняння результатів між різними конфігураціями параметрів генетичного алгоритму,

Бажано також окремо подати:

- псевдокод або блок-схему загального циклу генетичного алгоритму,
- короткий аналіз типових помилок реалізації, наприклад передчасної збіжності, виродження популяції або некоректного масштабу штрафів,
- приклади невдалих налаштувань, якщо вони були зафіксовані під час експериментів.

Контрольні запитання

1. Що таке генетичний алгоритм і чим він відрізняється від локального пошуку?
2. Що таке хромосома, ген і популяція в контексті еволюційної оптимізації?
3. Яку роль виконує функція пристосованості?
4. Які типи кодування хромосоми використовують для неперервних і дискретних задач?
5. У чому полягає призначення селекції?
6. Чим схрещування відрізняється від мутації?
7. Для чого в генетичних алгоритмах потрібен елітизм?
8. Які критерії зупинки можна використати в ГА?
9. Чому генетичний алгоритм не завжди є найкращим вибором для гладкої опуклої функції?

10. Чому багатомодальні функції є складними для багатьох класичних методів оптимізації?

11. Які труднощі створюють розриви, негладкість або дискретність простору пошуку?

12. Що таке жорсткі та м'які обмеження?

13. Для чого використовується штрафна функція?

14. Чому масштаб штрафів у задачі з обмеженнями потрібно добирати обережно?

15. Як можна подати задачу складання розкладу у вигляді хромосоми?

16. Чому результати генетичного алгоритму слід порівнювати за кількома запусками, а не за одним?

17. Які параметри генетичного алгоритму найбільше впливають на його поведінку у наведених реалізаціях?

18. У яких випадках спеціалізовані підходи можуть виявитися кращими за генетичний алгоритм?

Критерії оцінювання

Оцінювання лабораторної роботи здійснюється за такими критеріями:

На 100%. Реалізовано генетичний алгоритм. Коректно задано кодування хромосоми та функції пристосованості. Обґрунтовано вибрано оператори селекції, схрещування, мутації та параметри алгоритму. Виконано порівняння щонайменше трьох конфігурацій параметрів та для трьох різних наборів вхідних даних. Реалізовано задачу складання розкладу з чітким розділенням жорстких і м'яких обмежень. Надано розшифровку штрафів для найкращого знайденого розкладу. Звіт містить повний аналіз збіжності, стабільності та придатності генетичного алгоритму для різних типів задач. Висновки є змістовними, аргументованими й узгодженими з експериментами.

На 70%. Генетичний алгоритм загалом реалізовано коректно, але аналіз проведено не повністю. Прикладні задачі реалізовано без порівняння впливу параметрів. У задачі розкладу наявні значні спрощення. Звіт містить основні

результати, але аналіз збіжності, штрафів способів кодування тощо надано не повністю.

На 50%. Реалізовано лише базову версію генетичного алгоритму без достатнього порівняльного аналізу. Тестові задачі охоплено мінімально (один набір параметрів та один набір даних). Кодування хромосоми чи функція пристосованості задані спрощені і не повністю обґрунтовані. Звіт містить лише частину обов’язкових елементів і поверховий аналіз результатів.

У разі, коли завдання виконано менш ніж на 50%, лабораторна робота вважається виконаною на 0.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Литвин В. В., Пасічник В. В., Яцишин Ю. В. Інтелектуальні системи : підручник. Львів : Новий Світ – 2000, 2020. 406 с. URL: https://ns2000.com.ua/wp-content/uploads/2019/10/Intelektual_system.pdf (дата звернення: 10.04.2026).

2. Удовик І. М., Коротенко Г. М., Коротенко Л. М., Трусов В. О., Харь А. Т. Методи та системи штучного інтелекту : навч. посіб. Дніпро : ДВНЗ «НГУ», 2017. 112 с. URL: https://it.nmu.org.ua/ua/library/Metody_ta_systemy_shtuchnogo_intelektu.pdf (дата звернення: 10.04.2026).

3. Bratko I. Prolog Programming for Artificial Intelligence. 4th ed. Harlow : Pearson Education, 2012. 673 p.

4. Sterling L., Shapiro E. The Art of Prolog: Advanced Programming Techniques. 2nd ed. Cambridge, MA : MIT Press, 1994. 509 p.

5. Новожилова М. В., Петрова О. О. Використання мови логічного програмування Visual Prolog для розробки експертних систем : навч. посіб. Харків : ХНУМГ ім. О. М. Бекетова, 2019. 89 с. URL: https://eprints.kname.edu.ua/55864/ (дата звернення: 10.04.2026).

6. SWI-Prolog reference manual**.** SWI-Prolog. [Електронний ресурс] URL: https://www.swi-prolog.org/pldoc/doc_for?object=manual (дата звернення: 01.05.2026).

7. SWISH – SWI-Prolog for SHaring**.** [Електронний ресурс] URL: https://swish.swi-prolog.org/ (дата звернення: 01.05.2026).

8. Brachman R. J., Levesque H. J. Knowledge Representation and Reasoning. Amsterdam ; Boston : Morgan Kaufmann, 2004. 381 p.

9. Довгий С. О. та ін. Комп'ютерні онтології та їх використання у навчальному процесі. Теорія і практика : монографія. Київ : Інститут обдарованої дитини НАПН України, 2013. 310 с. URL: https://core.ac.uk/download/pdf/32309159.pdf (дата звернення: 10.04.2026).

10. Russell S. J., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Hoboken : Pearson, 2021. 1168 p.

11. Knuth D. E., Moore R. W. An Analysis of Alpha-Beta Pruning. Artificial Intelligence. 1975. Vol. 6, No. 4. P. 293–326.

12. Millington I. AI for Games. 3rd ed. Boca Raton : CRC Press, 2019. 1010 p. DOI: 10.1201/9781351053303.

13. Мероник Т. Ю. Ігровий штучний інтелект – підходи до побудови : текстова частина до курсової роботи. Київ : НаУКМА, 2020. 43 с. URL: https://ekmair.ukma.edu.ua/bitstreams/f382dedc-4cb2-4901-8c07-da56abd71fbc/download (дата звернення: 10.04.2026).

14. Hoki K., Muramatsu M. Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move pruning, and Late Move Reduction (LMR). Entertainment Computing. 2012. Vol. 3, No. 3. P. 51–57. DOI: 10.1016/j.entcom.2011.11.003.

15. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. 2nd ed. Cham : Springer, 2025. DOI: 10.1007/978-3-031-83347-2.

16. Novikov O. A., Yanovsky V. V. Analysis of Search and Multi-Agent Algorithms in the Pac-Man Game. Control Systems and Computers. 2024. No. 4. P. 19–33. DOI: 10.15407/csc.2024.04.019.

17. Hart P. E., Nilsson N. J., Raphael B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics. 1968. Vol. 4, No. 2. P. 100–107. DOI: 10.1109/TSSC.1968.300136.

18. The Pac-Man Projects [Електронний ресурс] // Model AI Assignments / UC Berkeley CS188. URL: https://modelai.gettysburg.edu/2010/pacman/pacman.html

19. Holland J. H. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Cambridge, MA : MIT Press, 1992.

20. Goldberg D. E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA : Addison-Wesley, 1989. 412 p.

21. Eiben A. E., Smith J. E. Introduction to Evolutionary Computing. 2nd ed. Berlin ; Heidelberg : Springer, 2015. 287 p. DOI: 10.1007/978-3-662-44874-8.

22. Савченко І. О. Еволюційні методи оптимізації. Комп'ютерний практикум [Електронний ресурс] : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2025. 47 с. URL: https://ela.kpi.ua/items/6958480f-1387-4c99-ad25-35a3e0c12a95 (дата звернення: 10.04.2026).

23. Бугаєва І. Г. Аналіз параметрів генетичного алгоритму розв'язання двовимірної задачі упаковки. Вісник Херсонського національного технічного університету. 2025. No 2(93), ч. 2. С. 53-59. DOI: 10.35546/kntu2078-4481.2025.2.2.6.

Тарануха Володимир Юрійович

Інтелектуальні системи

Лабораторний практикум