МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ

“До захисту допущена”
Зав. кафедри комп'ютерних наук
та прикладної математики
д.т.н., професор Турбал Ю. В.
«___» _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

«Генетичні алгоритми у криптоарифметичних задачах»

Виконала: Пристопчук Яна Юріївна
студентка навчально-наукового інституту автоматики, кібернетики та обчислювальної техніки
група ІПЗ-41

_____
(підпис)

Керівник: зав. кафедри комп’ютерних наук та прикладної математики, д.т.н., професор Турбал Юрій Васильович

_____
(підпис)

Рівне – 2023

РЕФЕРАТ

Об’єкт дослідження: аналітичні методи криптографії.

Предмет дослідження: генетичні алгоритми в криптографії.

Мета роботи: дослідження використання генетичних алгоритмів у криптографії, зокрема, для вирішення криптоарифметичних задач.

Актуальність: Генетичні алгоритми — це частина ширшої області, яка називається еволюційними обчисленнями. Однією з областей еволюційних обчислень, що тісно пов'язана з генетичними алгоритмами, є генетичне програмування , при якому програми і задіюють операції відбору, кросинговера та мутації, щоб, змінюючи себе, знайти неочевидні рішення задач програмування. Генетичне програмування — технологія, що не дуже широко використовується, але уявіть собі майбутнє, в якому програми пишуть себе самі.

Перевага генетичних алгоритмів полягає в тому, що вони легко піддаються розпаралелюванню. У найбільш очевидній формі кожна популяція може бути змодельована на окремому процесорі. У більш детальному варіанті для кожного індивіда виділяється особливий потік, в якому він може мутувати і брати участь у кросинговері і де обчислюється його життєздатність. Існує також безліч проміжних можливостей.

Ключові слова: алгоритм, захищеність, криптографія, криптоарифметика.

Зміст

Вступ

Генетичні алгоритми найчастіше використовуються тоді, коли традиційні алгоритмічні підходи не підходять для вирішення проблеми у розумні терміни. Іншими словами, генетичні алгоритми зазвичай застосовуються для складних завдань, які не мають простих рішень. Їх часто використовують для вирішення складних завдань, які не вимагають абсолютно оптимальних рішень, таких як багатокритеріальні завдання з обмеженнями. Одним із прикладів таких завдань є складні проблеми планування.

Генетичні алгоритми знайшли широке застосування у обчислювальній біології. Ці алгоритми використовуються у фармацевтичних дослідженнях та для того, щоб краще зрозуміти механізми природних явищ. У комп'ютерному мистецтві генетичні алгоритми іноді використовуються для імітації фотографій за допомогою стохастичних методів.

Генетичні алгоритми — це частина ширшої області, яка називається еволюційними обчисленнями. Однією з областей еволюційних обчислень, що тісно пов'язана з генетичними алгоритмами, є генетичне програмування , при якому програми і задіюють операції відбору, кроссовера та мутації, щоб, змінюючи себе, знайти неочевидні рішення задач програмування. Генетичне програмування — технологія, що не дуже широко використовується, але уявіть собі майбутнє, в якому програми пишуть себе самі.

Перевага генетичних алгоритмів полягає в тому, що вони легко піддаються розпаралелюванню. У найбільш очевидній формі кожна популяція може бути змодельована на окремому процесорі. У більш детальному варіанті для кожного індивіда виділяється особливий потік, в якому він може мутувати і брати участь у кросинговері і де обчислюється його життєздатність. Існує також безліч проміжних можливостей.

Мета роботи: дослідження використання генетичних алгоритмів у криптографії, зокрема, для вирішення криптоарифметичних задач.

Об’єкт дослідження: аналітичні методи криптографії.

Предмет дослідження: генетичні алгоритми в криптографії.

Розділ 1. Криптоарифметичні задачі та їх специфіка.

1.1 Поняття криптоарифметичних задач

Криптарифметика — це клас проблем, що відносяться до так званих задач задоволення обмежень, які включають створення математичних співвідношень між значущими словами за допомогою простих арифметичних операторів.

Назва «криптарифм» була придумана Міносом у травневому випуску бельгійського журналу з математики «Сфінкс» за 1931 рік і була перекладена як «криптарифметика» Морісом Крайтчиком у 1942 році. У 1955 році Дж. А. Х. Хантер ввів слово «алфавітний» для позначення криптарифмів, таких як код Дудені, літери якого утворюють значущі слова або фрази. Криптарифметична задача - це своєрідна головоломка, що складається з арифметичного завдання, в якому цифри замінено буквами деякого алфавіта. Мета полягає в тому, щоб розшифрувати літери використовуючи обмеження, надані арифметичною задачею та додатковим обмеженням, що немає двох різних символів які можуть мати однакове числове значення.

Одну з найпростіших відомих криптарифметичних задач Генрі Дадені показано на рис. 1.1.

Рис.1.1. Задача Генрі Дадені

Розвязком задачі є наступні значення. O=0, M=1, Y=2, E=5, N=6, D=7, R=8 і S=9. Обмеження для даної криптарифметичної задачі є наступними: арифметичні операції є десятковими; отже повинно бути максимум десять різних літер в рядках, які використовуються. Всі однакові літери повинні бути прив'язані до унікальної цифри, а дві різні літери не можуть позначати ту ж саму цифру. Оскільки слова будуть позначати числа, то

перша буква не може бути 0. Отримані числа повинні задовольняти задачу, це означає, що результат двох перших чисел (операндів) під вказаною арифметичною операцією (оператор плюс) має бути третім числом.

Виділяють наступні типи криптарифмів: альфетичний, дигіметичний, скелетний та зворотній.

Альфетичний - тип криптарифма, в якому набір слів записується у вигляді довгої суми додавання або будь-якої іншої математичної задачі. Ціль полягає в тому, щоб замінити літери алфавіту десятковими цифрами для отримання дійсної арифметичної суми.

Дигіметичний - криптарифм, у якому цифри використовуються для позначення інших позначень.

Скелетний поділ - поділ, у якому більшість цифр замінюються символами (зазвичай зірочками) для формування криптарифма.

Зворотний криптарифм - рідкісний варіант, коли формується деяка формула, а рішенням є відповідний криптарифм, рішенням якого є наведена формула.

При узагальненні, проблема визначення того, чи криптарифм має розв'язок, є NP-повною.

1.2 Загальна задача виконання обмежень CSP

1.2.1 Постановка проблеми

Багато проблем реального світу такі як розподіл ресурсів, часу, планування діяльності генерують новий клас задач - проблеми виконання обмежень (CSP). Багато теоретичних задач і багато логічних ігор також є природними прикладами CSP: розфарбовування графів, зіставлення графів, криптоарифметика, N-ферми, латинські квадрати, судоку та його незліченна кількість варіантів, футошікі, какуро та багато інших математичних ігор (або логічних головоломок).

За останні десятиліття вивчення таких проблем перетворилося на основну підгалузь штучного інтелекту (ШІ) з власними спеціалізованими

методами. Дослідження були зосереджені на пошуку ефективних алгоритмів, що було необхідно для роботи з великомасштабними програмами.

CSP формально описуються множиною рішень і множиною обмежень на комбінації рішень.

Рішення описуються в термінах змінних, кожній з яких може бути присвоєне значення з області її значень. Обмеження описуються в термінах відношень, що встановлюють, які з комбінацій значень змінних є істинними.

Існує два підходи для представлення задач у вигляді CSP: встановлення стартових параметрів для кожної задачі таким чином, щоб виконувались всі часові та ресурсні обмеження; встановлення обмежень впорядкування на задачі таким чином, щоб виконувались всі часові та ресурсні обмеження.

Задача CSP визначається через мережу обмежень , яка складається з кінцевого набору змінних, кожна з яких пов'язана з областю значень, і набору обмежень. Обмеження вказує для певної підмножини змінних набір несумісних комбінацій значень для цих змінних. Рішення CSP — це присвоєння значення кожній змінній значення із її домену таким чином, щоб усі обмеження були задоволені . CSP є узгодженою , якщо вона має принаймні одне рішення; в іншому випадку задача є надмірно обмеженою, або нерозвязною. Наприклад, Max-CSP — це проблема визначення присвоєння значення кожній змінній із її домену таким чином, щоб було задоволено якомога більше обмежень. Задоволення обмежень забезпечує зручний спосіб представлення та вирішення проблем, у яких взаємно сумісні значення повинні бути призначені заздалегідь визначеній кількості змінних за набором обмежень. Такі проблеми виникають у різноманітних областях, включаючи компютерний зір, теорію графів, проектування схем та діагностичне міркування .

Добре відомим прикладом проблеми задоволення обмежень є задача *k* -розмальовування , де завдання полягає в тому, щоб розфарбувати даний граф G не більше ніж у *k* кольорів так, щоб будь-які дві сусідні вершини мали різні кольори. Якщо таке забарвлення існує, то G називається *k*-розфарбованим .

Формулювання задоволення обмежень цієї проблеми пов'язує вершини графа зі змінними, набір можливих кольорів $\{1,\ldots, k\}$ є областю визначення кожної змінної, а обмеження нерівності між суміжними вершинами є обмеженнями проблеми. Для даного CSP часткове визначення – це призначення значення з його домену, тобто області допустимих значень для кожної змінної окремо, деяким змінним, але не обов'язково всім. Коли всі змінні отримують значення, присвоєння вважається завершеним.

Підмножина S обмежень CSP є незадовільною, якщо жодне повне призначення не задовольняє всім обмеженням у S одночасно. Згідно з термінологією, підмножина обмежень називається незвідною незадовільною множиною (IIS, Irreducible Infeasible Set) обмежень, якщо вона не має розв'язку, але стає розв'язною, коли хоча б одне довільне обмеження буде видалено. Подібним чином підмножина V змінних CSP є незадовільною, якщо немає законного часткового призначення змінних у V. Ми визначаємо незвідний незадовільний набір (IIS) змінних як будь-яку підмножину змінних, яка є незадовільною, але стає розв'язною, коли будь-яка змінна із набору змінних видаляється.

1.2.2 Проблема k -розфарбування

Розглянемо граф, представлений на Рис.1.а. CSP, що відповідає задачі 2-розфарбування на цьому графі, має $V = \{v_1,\ldots, v_7\}$ як набір змінних (набір вершин). З кожним ребром (v_i, v_j) ми пов'язуємо обмеження, позначене (i, j), яке накладає на v_i та v_j різні кольори. Отже, набір обмежень C є $\{(1,2),(1,3),(1,7),(2,3),(3,4),(4,5),(5,6),(5,7),(6,7)\}$. Цей CSP має чотири IIS обмежень, $\{(1,2),(1,3),(2,3)\}$, $\{(5,6),(5,7),(6,7)\}$, $\{(1,3),(1,7),(3,4),(4,5),(5,7)\}$ і $\{(1,2),(1,7),(2,3),(3,4),(4,5),(5,6),(6,7)\}$ і три IIS змінних, $\{v_1, v_2, v_3\}$ $\{v_5, v_6, v_7\}$ і $\{v_1, v_3, v_4, v_5, v_7\}$.

Рис.1.2. Розфарбування графа

Якщо змінні CSP визначають IIS змінних, то обмеження, які включають ці змінні, не обов'язково визначають IIS обмежень. Як приклад розглянемо CSP, пов'язаний із проблемою 3-кольорового розфарбування для графа на Рис.1.b. Набір вершин V графа є IIS змінних, оскільки граф не є 3-кольоровим, тоді як видалення будь-якої вершини створює 3-кольоровий граф. Однак набір ребер не є IIS обмеженям, оскільки видалення ребра, що зв'язує v_1 з v_2, дає граф, який все ще не є 3-кольоровим.

Відображення IIS обмежень або змінних може бути дуже корисним на практиці, особливо IIS невеликого розміру. Наприклад, при вирішенні задачі складання розкладу часто буває, що немає рішення, яке б задовольняло всі обмеження. IIS представляє частину проблеми, яка дає часткове пояснення цієї проблеми.

Визначення IIS обмежень або змінних також може бути дуже корисним для доведення невідповідності CSP. Дійсно, IIS зазвичай містять невелику кількість обмежень і змінних у порівнянні з вихідною проблемою, тому доведення невідповідності легше отримати в IIS, ніж у вихідній проблемі. Щоб проілюструвати це, розглянемо ще раз проблему k -розфарбування.

Припустимо, що жоден евристичний алгоритм не здатний визначити k -розфарбовування розглянутого графа G. Тоді можна припустити, що G не є k -розфарбованим. Щоб довести це, достатньо показати частковий підграф G' (отриманий видаленням ребер) або індукований підграф G" (отриманий

видаленням вершин і всіх ребер, інцидентних цим вершинам), який не є k -розфарбованим, але стає k - розфарбовуваним, як тільки будь-яке ребро G' або будь-яка вершина G" буде видалено. Ребра часткового підграфа G' відповідають IIS обмеженням, тоді як вершини індукованого підграфа G" формують IIS змінних. Якщо G' і G" мають менше ребер і вершин, ніж G, тоді замість того, щоб доводити, що G не є k -розфарбованим, легше довести, що G' або G" не є k -розфарбованим.

Кроуфорд і Мазур та ін. розробили алгоритми для визначення нерозвязних підмножин обмежень для проблеми задоволення обмежень.

Проблеми пошуку IIS обмежень і змінних у неузгодженому CSP є двома окремими випадками LSCP. Дійсно, враховуючи суперечливий CSP, визначте як елемент E будь-яке повне визначення.

До кожного обмеження c_i (i =1,…, m) CSP поставимо у відповідність підмножину E_i з E , що містить усі визначення, які порушують c_i . Набір обмежень є нерозвязним тоді і тільки тоді, коли він покриває E. Отже, пошук IIS обмежень еквівалентний розв'язанню LSCP.

Процедура IS-ELEMENT (e, i) просто визначає, чи повне призначення e порушує обмеження i. Процедура MIN_WEIGHT (ω) повертає повне призначення e , яке мінімізує суму ваг обмежень, порушених у e . У випадку проблеми k -розфарбування визначають конфліктне ребро як ребро, що має обидві кінцеві точки однакового кольору. Процедура MIN_WEIGHT (ω) повертає забарвлення, яке мінімізує суму ваг конфліктуючих ребер. Задача, розв'язана процедурою MIN_WEIGHT, відповідає Max-CSP, коли всі ваги дорівнюють 1.

З кожною змінною v_i CSP пов'яжемо підмножину E_i з E, що містить усі допустимі часткові призначення, у яких v_i не має значення (тобто v_i не створено). Підмножина змінних є нездійсненною тоді і тільки тоді, коли вона покриває E. Отже, тут знову пошук IIS змінних еквівалентний розв'язанню LSCP. Процедура IS-ELEMENT (e, i) повертає «true» тоді і тільки тоді, коли змінна i не створена в частковому призначенні e. Процедура MIN_WEIGHT (ω)

повертає законне часткове призначення *e*, яке мінімізує суму ваг змінних, які не створені в *e*. У випадку проблеми *k*-розмальовування процедура MIN_WEIGHT (ω) повертає часткове забарвлення без конфліктуючих ребер, що мінімізує суму ваг незабарвлених вершин.

На цьому етапі важливо зауважити, що процедура MIN_WEIGHT зазвичай вирішує NP-складну проблему як у випадку пошуку IIS обмежень, так і IIS змінних.

1.2.3 Інтерактивна проблема підтримки прийняття рішень

Є деякі випадки, коли MIN_WEIGHT можна реалізувати по-іншому. Для ілюстрації розглянемо CSP, який з'являється в контексті інтерактивної проблеми підтримки прийняття рішень. Розглядається послідовний CSP з набором *C* обмежень, де рішення представляють каталог компанії. Під час конфігурації продукту користувач системи підтримки прийняття рішень вказує ряд додаткових обмежень C_1, $C_{2,...}$ щодо особливостей продукту, який його цікавить. На деякій ітерації *p* цей набір додаткових обмежень може стати нездійсненним (тобто $C \cup \{C_1, ..., C_{p-1}\}$ є можливим, тоді як $C \cup \{C_1, ..., C_p\}$ ні). Щоб керувати користувачем, система повинна забезпечити мінімальну підмножину S $\{C_1, ..., C_p\}$ таку, що $C \cup S$ вже є нездійсненною. Набір *S* називається поясненням, оскільки він пояснює, чому обмеження C_p породжує невідповідність. Завдяки складним структурам даних Amilhaste et al. розробили ефективну процедуру, яка реалізує процедуру MIN_WEIGHT : процедура забезпечує рішення, яке не порушує жодних обмежень у $C \cup \{C_p\}$ і мінімально зважену підмножину обмежень у $\{C_1, ..., C_{p-1}\}$ для будь-якого зважування обмежень. Процедура використовується для визначення максимальної підмножини F $\{C_1, ..., C_{p-1}\}$ такої, що $C \cup F \cup \{C_p\}$ є можливим.

Деякі аспекти пошуку розв'язку задач CSP

Існує дві категорії алгоритмів розв'язання ЗЗО: конструктивний пошук; локальний пошук.

При конструктивному пошуці циклічно виконується вибір і встановлення значення змінної; перевірка обмежень; відступ назад при порушенні обмежень.

При локальному пошуці спочатку виконується встановлення значень всіх змінних, а потім робиться спроба “відремонтувати” порушені обмеженні шляхом зміни значення змінної в одному з обмежень.

Для конструктивного пошуку характерно наступне. Розглянемо простий алгоритм конструктивного пошуку для розв’язання CSP.

Ф(CSP):

1. Якщо порушене будь-яке обмеження, то НЕВДАЧА(CSP);
2. Якщо всі змінні встановлені, то ВИХІД(CSP);
3. Вибрати невстановлену змінну V;
4. Вибрати значення V_i для V;
5. CSP’ = Розповсюдити(CSP ∪ $V=V_i$);
6. **Ф**(CSP’).

На кожному рівні алгоритму виконується вибір та встановлення значення змінної і рекурсивний виклик самого себе на конкретизованій версії CSP.

Якщо вибране значення змінної несумісне, то процедура зазнає невдачі і виконується хронологічний відступ назад. Пунктами відступу назад є оператори Вибрати.

Цей алгоритм складається з декількох компонентів:

1 впорядкування змінних;
2 впорядкування значень змінних;
3 стратегія розповсюдження обмежень;
4 стратегія відступу назад.

У впорядкуванні змінних найбільш відомою евристикою є мінімум значень, що залишилися (МЗЗ), яка обирає змінну з найменшим числом серед значень, що залишилися не обрані.

Більш ефективними для розглядуваних нами задач є евристики, які впорядковують змінні у відповідності з шириною стартового вікна задачі та перетином між часовими інтервалами.

Загальною стратегією для впорядкування значень змінних є вибір значень, які спричиняють найменше обмеження областей неозначенних змінних.

Механізм розповсюдження обмежень виводить нові обмеження в результаті встановлення значення змінної. В діапозон стратегій розповсюдження обмежень входять як прості стратегії швидкої перевірки сумісності, подібні “перевірці вперед”, так і потужні, але вартісні стратегії К-сумісності, які можуть виводити нові К-місні обмеження серед залишившихся змінних.

Алгоритм конструктивного пошуку виконує хронологічний відступ назад, коли поточне встановлення змінної приводить до невдачі.

Існує багато інших алгоритмів конструктивного пошуку для CSP, які ідентифікують змінні, що відповідають за невдачу і відступають назад на відповідний рівень.

Локальний пошук звичайно починається з повного встановлення значень змінних, навіть незважаючи на те, що деякі обмеження можуть бути порушені. Поступово, модифікуючи встановлення шляхом зміни деяких, або всіх значень змінних досягається розв’язання CSP.

В процесі пошуку генерується “сусідство” нових встановлень, виконується їх ранжування і вибір найкращої. Кожна з цих дій називається кроком.

Оскільки ефективність цього підходу сильно залежить від вихідного встановлення, після відповідного числа кроків і невдачі знайти розв’язок, генерується нове вихідне встановлення і т.д. Кожний цикл, який включає генерацію вихідного встановлення і виконання ряду кроків, називається спробою. Після достатньої кількості невдачних спроб процес завершується з визнанням загальної невдачі.

Алгоритм локального пошуку зображено наступному рисунку.

ЛП(CSP)

1. ПОВТОРИТИ ДЛЯ СПРОБ;
2. Генерувати вихідне встановлення A для змінних в CSP;
3. Повторити для кроків;
4. Якщо обмеження не порушені, ВИХІД(A);
5. Відібрати сусіднє встановлення A ';
6. Встановити A ← A '.

Алгоритм описує сімейство алгоритмів локального пошуку. Для того, щоб конкретизувати алгоритм, необхідно специфікувати процедури генерувати сусідів, ранжувати сусідів, відібрати сусіда. Ефективність алгоритму ЛП залежить від цих процедур, які визначаються для кожної задачі окремо.

1.3 Евристичні стратегії в криптоарифметиці

Евристики – широке «сімейство» різнопланових прийомів мислення від найпростішої можливості перервати в якийсь момент хід рішення до найскладніших форм інтелектуальної рефлексії – призначені для ефективного самонаведення на розв'язання задачі чи проблеми. Найбільш відома функція евристик полягає в тому, що вони дозволяють уникнути повного перебору варіантів, спрямованих на пошук рішення.

Евристичні прийоми нерідко визначаються як засоби активації мислення для досягнення тієї чи іншої мети. Сам термін «евристика» ввів К.Дункер (1926), намагаючись з його допомогою відповісти на питання про рушійні сили продуктивного розумового процесу (про реальні механізми переходу від будь-якої стадії розв'язання задачі до наступної). Зазвичай евристики різко протиставляються алгоритму, тобто строгій послідовності операцій, точне виконання яких гарантовано призводить до досягнення правильного результату.

Для криптоарифметичних задач, які є предметом дослідження даної роботи, важливими є методи, розроблені А. Ньюеллом і Г. Саймоном.

Перший з цих методів – метод зменшення відмінності – є евристичним методом, який має стратегію, що передбачає послідовне зменшення відмінності (розриву) між поточним та цільовим станом. Часто при вирішенні проблем, особливо у незнайомих областях, вибирають оператори (способи дій), які перетворять поточний стан у новий, що зменшує різницю між ними по певних критеріях. Зменшення відмінності іноді називається сходженням на пагорб. Якщо уявити мету у вигляді найвищої точки на поверхні землі, один із способів спробувати досягти її полягає в тому, щоб завжди йти «вгору».

Однак у ряді випадків принцип зменшення відмінностей не спрацьовує. Ця евристика іноді заважає тимчасово відхилитися від цілі, не враховуючи, що саме таке відхилення дозволяє вирішити завдання ефективніше.

Для виправлення недоліків методу зменшення відмінностей, А.Ньюелл і Г.Саймон (1961) створили евристику під назвою «аналіз коштів та цілей», яка пропонує більш загальний підхід до вирішення проблем. Ця евристика є основою в розвязанні цілого класу криптоарифметичних задач. А.Ньюелл і Г.Саймон використали цю евристику в комп'ютерній програмі під назвою «Універсальний вирішувач завдань» (УРЗ), яка моделює вирішення проблем людьми.

Цей вид аналізу – класифікація речей у термінах їх функцій та зіставлення цілей, необхідних функцій та засобів їх реалізації – утворює основу УРЗ. Аналіз засобів та цілей може розглядатися як складніша версія принципу зменшення відмінності. Загальна особливість цієї евристики полягає в тому, що вона розбиває велику мету на підцілі.

Аналіз коштів та цілей виявився дуже загальним та потужним методом вирішення проблем. А.Ньюелл і Г.Саймон обговорювали можливість його застосування в різних областях (у шахах, логіці, доказі теорем, у криптоарифметиці).

УРЗ поступово був витіснений іншими методами, але він був визнаний як відкриття в області вирішення задач комп'ютерами шляхом розкладання складного завдання на підзавдання, які вирішити значно простіше.

Розділ 2 Генетичні алгоритми та їх особливості

2.1 Поняття генетичних алгоритмів

Генетичний алгоритм (англ. genetic algorithm) - це евристичний алгоритм пошуку, який використовується для вирішення задач оптимізації та моделювання шляхом послідовного підбору, комбінування та варіації шуканих параметрів з використанням механізмів, що нагадують біологічну еволюцію. Є різновидом еволюційних обчислень (англ. evolutionary computation). Відмінною особливістю генетичного алгоритму є акцент на використання оператора «схрещування», який здійснює операцію рекомбінації рішень-кандидатів, роль якої аналогічна ролі схрещування у живій природі. "Батьком-засновником" генетичних алгоритмів вважається Джон Холланд (англ. John Holland), книга якого "Адаптація в природних і штучних системах" (англ. Adaptation in Natural and Artificial Systems) є основною працею в цій галузі досліджень.

При описі генетичних алгоритмів використовують визначення, запозичені з генетики. Наприклад, йдеться про популяцію особин, а як базові поняття застосовуються ген, хромосому, генотип, фенотип, аллель. Також використовуються відповідні цим термінам визначення з технічного лексикону, зокрема, ланцюг, двійкова послідовність, структура.

Дуже важливим поняттям генетичних алгоритмах вважається функція пристосованості (fitness function), інакше звана функцією оцінки. Вона є мірою пристосованості цієї особи в популяції. Ця функція відіграє найважливішу роль, оскільки дозволяє оцінити ступінь пристосованості конкретних особин у популяції і вибрати з них найбільш пристосовані (тобто такі, що мають найбільші значення функції пристосованості) відповідно до еволюційного принципу виживання «найсильніших» (найкраще пристосованих). Функція пристосованості також одержала свою назву безпосередньо з генетики. Вона сильно впливає на функціонування генетичних алгоритмів і повинна мати точне і коректне визначення. У задачах оптимізації функція пристосованості, як правило, оптимізується (точніше кажучи, максимізується) і називається

цільовою функцією. У завдання мінімізації цільова функція перетворюється, і проблема зводиться до максимізації. Теоретично управління функція пристосованості може набувати вигляду функції похибки, а теорії ігор - вартісної функції. На кожній ітерації генетичного алгоритму пристосованість кожної особини цієї популяції оцінюється за допомогою функції пристосованості, і на цій основі створюється наступна популяція особин, що становлять безліч потенційних рішень проблеми, наприклад, завдання оптимізації.

Чергова популяція в генетичному алгоритмі називається поколінням, а до новоствореної популяції особин застосовується термін "нове покоління" або "покоління нащадків".

Біологічна теорія пояснює, як генетична мутація в поєднанні з обмеженнями навколишнього середовища з перебігом часу призводить до змін в організмі, включаючи видоутворення - створення нових видів. Механізм, що зумовлює те, що добре адаптовані організми процвітають, а гірше адаптовані гинуть, відомий як природний відбір . У кожне покоління певного виду входять особини з різними, а іноді новими рисами, які виникають в результаті генетичної мутації. Усі особини змагаються за обмежені ресурси щоб вижити, і оскільки живих особин зазвичай більше, ніж ресурсів, виживають кращі.

Якщо особина має мутацію, яка робить її краще пристосованою до навколишнього середовища, то у неї вища ймовірність виживання та відтворення. З часом у особин, добре адаптованих до навколишнього середовищі, буде більше нащадків, які успадкують цю мутацію. Отже, мутація, сприятлива виживанню, швидше за все, поширіться на всю популяцію.

Наприклад, якщо бактерії гинуть від певного антибіотика, але одна з них в популяції має мутацію в гені, яка робить її більше стійкою до цього антибіотика, то ця бактерія з більшою ймовірністю виживе і розмножиться. Якщо антибіотик постійно застосовується протягом тривалого часу, то нащадки цієї бактерії, що успадкували її ген стійкості до антибіотиків, також з більшою ймовірністю будуть розмножуватися і мати власних нащадків. У кінці

кінців ця мутація може поширитися на всю популяцію бактерій, оскільки тривалий вплив антибіотика буде вбивати усі бактерії поза мутацією. Антибіотик не викликає розвиток мутації, але призводить до розмноженню особин, що мають таку мутацію.

В інформатиці генетичні алгоритми — це моделювання природного відбору для вирішення обчислювальних завдань.

Генетичний алгоритм має на увазі наявність популяції (групи) особин, відомих як хромосоми . Хромосоми, кожна з яких складається з генів , що визначають її властивості, конкурують за вирішення якоїсь проблеми. Те, наскільки успішно хромосома вирішує проблему, визначається функцією життєздатності .

Генетичний алгоритм обробляє покоління. У кожному поколінні з більшою ймовірністю будуть відібрані для розмноження найбільш придатні хромосоми . Також у кожному поколінні існує ймовірність того, що якісь дві хромосоми об'єднають свої гени. І нарешті, у кожному поколінні існує важлива можливість того, що якийсь з генів хромосоми може мутувати - змінитися випадковим чином. Якщо функція життєздатності будь-якої з особин популяції перетинає певний поріг чи алгоритм проходить деяке зазначене максимальне число поколінь, повертається найкраща особина - та, яка набрала найбільшу кількість балів в функції життєздатності.

Генетичні алгоритми залежать від трьох частково або повністю стохастичних (випадково визначених) операцій: відбору, кросинговера та мутації, тому можуть не знайти оптимального рішення в розумні терміни. Є задачі, для яких немає швидких детермінованих алгоритмів розвязку. Для такий задач генетичні алгоритми є дуже хорошим вибором.

В загальному роботу генетичного алгоритму можна описати основними етапами :

1. Генерація початкової популяції, що складається з особин, вибраних випадковим чином.

2. Для кожної особини алгоритм обчислює ймовірність Ps(i), яка дорівнює відношенню її пристосованості до сумарної пристосованості популяції:

$$P_s(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)}$$

3. Відповідно до величини Ps(i) відбувається відбір найсильніших n особин для подальшої генетичної обробки. Члени популяції з високою пристосованістю будуть вибиратися з більшою імовірністю, ніж особини з низькою пристосованістю.

4. Після відбору, n обраних особин випадковим чином поділяються на n/2 пари, які будуть схрещуватися між собою.

5. Для кожної пари застосовується схрещування (кросинговер). Кросинговер - це операція, при якій із двох хромосом породжується кілька нових хромосом.

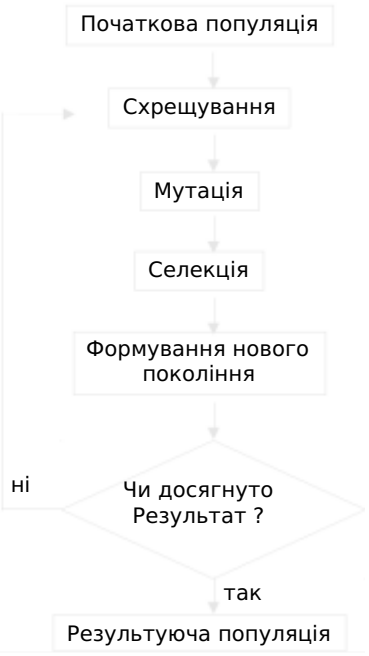

Рис. 2.1. Генетичний алгоритм

Для одноточкового кросинговеру випадковим чином вибирається точка розриву (ділянка між сусідніми ланками в рядку). Обидві батьківські структури розриваються на два сегменти в цій точці. Потім, відповідні сегменти різних батьків склеюються і виходять два генотипи нащадків

6. Відповідно до алгоритму випадково обрані особини піддаються мутації. Мутація - це перетворення хромосоми, яке випадково змінює одну чи декілька її позицій (генів). Поширеним видом мутацій є випадкова зміна лише одного гену з хромосоми. В особині, що піддається мутації, випадковий біт з імовірністю P_m змінюється на протилежний.

7. В утвореній популяції відбувається обчислення пристосованості особин (п.1) і обираються найсильніші. Отримана популяція заміняє поточну і цикл одного покоління завершується.

Наступні покоління обробляються подібним чином: відбір, кросинговер і мутація. В кожному наступному поколінні продукуються нові рішення, серед яких будуть як погані, так і хороші, але завдяки відбору число прийнятних рішень буде зростати.

Робота генетичного алгоритму є ітераційним процесом, що продовжується, доки не виконається задане число поколінь або інший критерій зупинки. Таким критерієм може бути:

- Знаходження глобального або кращого локального рішення.
- Вичерпання числа поколінь, що відведено на еволюцію.
- Вичерпання часу, що відведено на еволюцію.

2.2 Розробка універсального генетичного алгоритму UGA

2.2.1 Клас Chromosome

Генетичні алгоритми часто є вузькоспеціалізованими та розраховані на конкретне застосування. Побудуємо узагальнений генетичний алгоритм. Такий алгоритм можна застосовувати до цілого класу задач. На його основі ми запропонуємо рішення криптоарифметичних задач.

Насамперед визначимо інтерфейс для особин, з якими може працювати універсальний алгоритм. Анотація клас Chromosome визначає чотири основні властивості. Хромосома повинна вміти: визначати власну життєздатність; створювати екземпляр із випадково обраними генами (для використання при заповненні першого покоління); реалізовувати кросинговер (об'єднуватися з іншим об'єктом того ж типу, щоб створювати нащадків), іншими словами, схрещуватися з іншою хромосомою; мутувати - вносити в себе невелику випадкову зміну.

Розглянемо код – Лістинг 2.1 Chromosome, в якому реалізовані ці чотири властивості (листинг 2.1).

Лістинг 2.1. chromosome.py

```
from  future  import annotations
from typing import TypeVar , Tuple, Type
from abc import ABC, abstractmethod
T = TypeVar ( 'T' , bound = 'Chromosome' ) # щоб повертати self
# Базовий клас для всіх хромосом
class Chromosome( ABC): @abstractmethod
def fitness(self) -> float:
   pass
@classmethod @abstractmethod
def random_instance ( cls : Type[T]) -> T:
   pass
@abstractmethod
def crossover( self: T, other: T) -> Tuple[T, T]:
   pass
@abstractmethod
def mutate(self) -> None :
   pass
```

В конструкторі цього класу TypeVar T пов'язаний з хромосомою. Це означає, що все, що заповнює змінну типу T, повинно бути екземпляром класу Chromosome або підкласу Chromosome.

Ми реалізуємо сам алгоритм (код, який маніпулюватиме хромосомами) як параметризований клас, відкритий для створення підкласів для майбутніх спеціалізованих додатків.

Кроки узагальненого генетичного алгоритма будуть наступні.

1. Створити початкову популяцію випадкових хромосом першого покоління алгоритму.

2. Виміряти життєздатність кожної хромосоми в цьому поколінні

популяції. Якщо життєздатність якоїсь із них перевищує порогове значення, то повернути його та закінчити роботу алгоритму.

3. Вибрати для розмноження кілька особин. З самої високою ймовірністю для розмноження вибираються найбільш життєздатні особини.

4. Схрестити (об'єднати) з деякою ймовірністю частину з обраних хромосом, щоб створити нащадків, які репрезентують популяцію наступного покоління.

5. Виконати мутацію: мутують, як правило, з низькою ймовірністю деякі з хромосом. На цьому формування популяції нового покоління завершено і вона замінює популяцію попереднього покоління.

6. Якщо максимально допустима кількість поколінь не отримана, то повернутися до кроку 2. Якщо максимально допустима кількість поколінь отримана, повернути найкращу хромосому.

2.2.2 Особливості задання параметрів

Усі параметри для алгоритму, а саме скільки хромосом повинно бути в популяції, який поріг зупиняє алгоритм, як вибирати хромосоми для розмноження, як вони мають схрещуватися і з якою ймовірністю, з якою ймовірністю мають відбуватися мутації, скільки поколінь потрібно створити будуть налаштовуватись у класі GeneticAlgorithm (Листинг 5.2).

Лістинг 2.2. genetic_algorithm.py

```
from  future  import annotations
from typing import TypeVar , Generic, List, Tuple, Callable
from enum import Enum
from random import choices, random
from heapq import nlargest
from statistics import mean
from chromosome import Chromosome

C = TypeVar ( 'C' , bound=Chromosome) # тип хромосом
```

```
class GeneticAlgorithm ( Generic[C]):
SelectionType = Enum ( " SelectionType " , "ROULETTE TOURNAMENT" )
```


Рис . 2.2. Узагальнена схема генетичного алгоритму

2.2.3 Методи відбору

GeneticAlgorithm приймає параметризований тип, який відповідає Chromosome і називається C. Перелік SelectionType є внутрішнім типом, який використовується для визначення методу відбору, що застосовується в алгоритмі. Існує два найбільш поширені методи відбору для генетичного алгоритму - відбір методом рулетки (іноді званий пропорційним відбором по життєздатності) і турнірний відбір . Перший дає кожній хромосомі шанс бути відібраною пропорційно її життєздатності. При турнірному відборі певна кількість випадково вибраних хромосом бореться один з одним і вибираються такі, що мають кращу життєздатність (листинг 2.3).

Лістинг 2.3. genetic_algorithm.py (продовження)

```
def init (self, initial_population : List[C], threshold: float, max_generations : int =
100, mutation_chance : float = 0.01, crossover_chance : float = 0.7,
selection_type : SelectionType = SelectionType.TOURNAMENT ) -> None :
  self._population : List[C] = initial_population
```

```
self._threshold : float = threshold
self._max_generations : int = max_generations
self._mutation_chance : float = mutation_chance
self._crossover_chance : float = crossover_chance
self._selection_type : GeneticAlgorithm.SelectionType = selection_type
self._fitness_key : Callable = type( self._population [0]).fitness
```

У наведеному коді представлені всі властивості генетичного алгоритму, які будуть налаштовані в момент створення у вигляді __init__(). initial_ population, - хромосоми першого покоління алгоритму.

threshold - поріг життєздатності , який вказує на те, що розв'язання задачі, над яким працює генетичний алгоритм, знайдено;

max_generations — максимальна кількість поколінь, яку можна пройти. Якщо ми пройшли певну кількість поколінь і не було знайдено рішення з рівнем життєздатності, що перевищує threshold, то буде повернено найкраще з знайдених рішень;

mutation_chance - це ймовірність мутації кожної хромосоми у кожному поколінні;

crossover_chance - ймовірність того, що у двох батьків, відібраних для розмноження, з'являться нащадки, які представляють собою суміш батьківських генів, в протилежному випадку вони будуть просто дублікатами батьків. Зрештою, selection_type - це тип методу відбору, описаний в enum SelectionType.

Описаний раніше метод init приймає довгий список параметрів, більшість яких мають значення за замовчуванням. Вони встановлюють версії екземплярів налаштовуваних властивостей. У наших прикладах _population ініціалізується випадковим набором хромосом за допомогою методу random_instance() класу Chromosome. Іншими словами, перше покоління хромосом, що складається із випадкових особин. Це точка потенційної оптимізації для складнішого генетичного алгоритму. Замість того, щоб починати з суто випадкових особин, перше покоління завдяки деякому знанню

конкретного завдання може утримувати особин, що знаходяться ближче до необхідного. Таке явище називається посівом .

_fitness_key - посилання на метод, який ми будемо використовувати в GeneticAlgorithm для розрахунку життєздатності хромосоми. Цей клас повинен працювати з будь-яким підкласом. Отже, функції _fitness_key будуть різними в різних підкласах. Щоб отримати їх, скористаємося методом type() для посилання на конкретний підклас Chromosome, для якого ми хочемо визначити життєздатність.

Тепер розглянемо два методу відбору, які підтримує наш клас (листинг 5.4).

Лістинг 2.4. genetic_algorithm.py (продовження)

```
# Використовуємо метод рулетки з нормальним розподілом ,
# щоб вибрати двох батьків
# не працює при негативних значеннях життєздатності
def _pick_roulette ( self, wheel: list[float]) -> tuple[C, C]:
  return tuple( choices( self._population , weights=wheel, k=2))
```

Вибір методу рулетки заснований на відношенні життєздатності кожної хромосоми до сумарної життєздатності всіх хромосом даного покоління. Хромосоми з найвищою життєздатністю мають більше шансів бути відібраними. Значення, які відповідають життєздатності хромосоми, вказано у параметрі wheel. Реальний вибір зручно виконувати за допомогою функції choices() з модуля random стандартної бібліотеки Python. Ця функція приймає список елементів, з яких ми хочемо зробити вибір, список такої ж довжини, що містить ваги всіх елементів першого списку, і кількість елементів, що вибираються.

Також можна розрахувати частку загальної життєздатності для кожного елемента (пропорційно життєздатності), представлену значеннями з плаваючою комою від 0 до 1. Випадкове число (pick) від 0 до 1 можна використовувати для обчислення того, яку хромосому відібрати. Алгоритм буде працювати, послідовно зменшуючи pick на величину пропорційну життєздатності кожної хромосоми. Коли pick стане менше 0, це буде

хромосома для відбору. При цьому кожна хромосома вибирається пропорційно до її життєздатності.

Найпростіша форма турнірного відбору простіша, ніж вибір методом рулетки. Замість того, щоб обчислювати пропорції, ми просто вибираємо випадковим чином k хромосом з всієї популяції. У відборі перемагають дві хромосоми з найкращою життєздатністю із випадково обраної групи (лістинг 5.5).

Лістинг 2.5. genetic_algorithm.py (продовження)

```
# Вибираємо випадковим чином num_participants та беремо з них дві
найкращих
def _pick_tournament ( self, num_participants : int) -> tuple [C, C]:
  participants: list[C] = choices( self._population , k= num_participants )
  return tuple( nlargest (2, participants, key=self._fitness_key ))
```

У коді для _pick_tournament() спочатку використовується choices(), щоб випадково вибрати num_participants з _population. Потім за допомогою функції nlargest() з модуля heapq знаходимо двох самих великих індивідуумів по _fitness_key. Як і у багатьох інших параметрах генетичного алгоритму, найкращим способом визначення num_participants може бути метод проб і помилок. Слід мати на увазі, що чим більше учасників турніру, тим менше різноманітність в популяції, тому що хромосоми з поганою життєздатністю з більшою ймовірністю загинуть в процесі поєдинків між особинами.

Більш складні форми турнірного відбору можуть вибирати особин, які є не найкращими, а лише другими або третіми по життєздатності, грунтуючись на деякій моделі спадної ймовірності.

Рис . 2.3. Приклад відбору методом рулетки

Табл. 2.2. Частотні характеристики

Хромосома	Життєздатність	Ймовірність	Частка
1	54.5	25 %	0.250
2	17.44	8 %	0.080
3	37.06	17 %	0.170
4	27.25	13 %	0.125
5	81.75	38 %	0.375
Усього		100 %	

Методи _ pick_ roulette ( ) і _ pick_tournament () використовуються для відбору, що виконується в процесі розмноження . Відтворення реалізовано в методі_reproduce_and_replace(), який відповідає за те, щоб на зміну хромосомам останнього покоління прийшла нова популяція з тією ж кількістю хромосом (лістинг 2.6).

Лістинг 2.6. genetic_algorithm.py (продовження)

```
# Заміна популяції новим поколінням особин
import random
```

```
def _reproduce_and_replace(self)-> None:
    new_population: list[C] = []
    # продовжуємо , поки не заповнимо особинами Усе нове покоління
    while len(new_population) < len(self._population ):
    # вибір двох батьків
        if self._selection_type == GeneticAlgorithm.SelectionType.ROULETTE:
            parents: tuple[C, C] = self._
            pick_roulette([x.fitness() for x in self._population])
        else:
            parents = self._
            pick_tournament(len(self._population) // 2)
    # потенційне схрещування двох батьків
        if random.Random() < self._crossover_chance:
            new_population.extend(parents[0].crossover(parents[1]))
        else:
            new_population.extend(parents)
    # якщо число непарне , то один зайвий , тому видаляємо його
    if len(new_population) > len(self._population):
        new_population.pop()
    self._
    population = new_population  # замінюємо посилання
```

У _ reproduce_and_ replace ( ) виконуються наступні основні операції . Дві хромосоми, яких називають батьками (parents), відбираються для відтворення за допомогою одного з двох методів відбору. При турнірному відборі завжди проводиться турнір серед половини популяції.

Існує ймовірність _crossover_chance того, що для отримання двох нових хромосом два батьки будуть об'єднані. У цьому випадку вони додаються до нової популяції (New_population). Якщо нащадків немає, то два батька просто додаються до new_population.

Якщо нова популяція new_population містить стільки ж хромосом, як і стара _population, вона її замінює. Інакше повертаємось до кроку 1.

Метод _mutate(), який реалізує мутацію показаний в наступному коді (лістинг 2.7).

Лістинг 2.7. genetic_algorithm.py (продовження)

```
# Кожна особина мутує з ймовірністю _ mutation_chance
import random
def _mutate(self) -> None:
   for individual in self._population:
   if random.Random() < self._mutation_chance:
      individual.mutate()
```

Тепер у нас є всі будівельні блоки, необхідні для запуску генетичного алгоритму. Метод run() координує етапи вимірювань, відтворення (включаючи відбір) і мутації, в процесі яких одне покоління популяції замінюється іншим. Цей метод також відстежує найкращі, найбільш життєздатні хромосоми, виявлені на будь-якому етапі пошуку (лістинг 5.8).

Лістинг 2.8. genetic_algorithm.py (продовження)

```
# Виконання генетичного алгоритму для max_generations ітерацій
# та повернення кращою з знайдених особин
def run(self) -> C:
   best: C = max(self._population, key=self._fitness_key )
   for generation in range(self._max_generations ):
   # ранній вихід , якщо перевищено поріг
   if best.fitness() >= self._threshold:
      return best
   print(f"Generation {generation} Best { best.fitness ()}
Avg{mean(map(self._fitness_key, self._population ))}" )
   self._reproduce_and_replace()
   self._mutate()
   highest: C = max(self._population, key=self._fitness_key )
```

```
    if highest.fitness() > best.fitness():
        best = highest  # знайдено новий найкращий результат
    return best  # кращий знайдений результат з _ max_generations
```

Значення best дозволяє відстежувати найкращу зі знайдених досі хромосом . Основний цикл виконується _max_generations разів. Якщо якась хромосома перевищує поріг життєздатності, то вона повертається і метод закінчується. У протилежному випадку метод викликає _reproduce_and_replace() і _mutate() для створення наступного покоління та повторного запуску циклу. Якщо досягається значення _max_generations, то повертається найкраща хромосома з знайдених до сих пір.

2.2.4 Приклад застосування

Параметризований генетичний алгоритм GeneticAlgorithm буде працювати з будь-яким типом, який реалізує Хромосом. Як тест виконаємо просте завдання, яке легко вирішити традиційними методами. Ми спробуємо знайти максимальні значення для рівняння 6 x – x 2 + 4 y – y 2 — тобто значення x і y , при яких результат цього рівняння буде найбільшим. Результатом буде x = 3 і y = 2. Покажемо, що алгоритм працює.

Лістинг 2.9. simple_equation.py

```
from genetic_algorithm import GeneticAlgorithm
from random import randrange, random
from copy import deepcopy
class simpleEquation(Chromosome):
    def init(self, x: int, y: int) -> None:
        self.x: int = x
        self.y: int = y


    def fitness(self) -> float:  # 6x - x^2 + 4y - y^2
        return 6 * self.x - self.x * self.x + 4 * self.y - self.y * self.y
```

```
    @classmethod
    def random_instance(cls) -> SimpleEquation:
        return SimpleEquation(randrange(100), randrange(100))
    def crossover(self, other: SimpleEquation) ->
Tuple[SimpleEquation,SimpleEquation]:
        child1: SimpleEquation = deepcopy(self)
        child2: SimpleEquation = deepcopy(other)
        child1.y = other.y
        child2.y = self.y
        return child1, child2

    def mutate(self) -> None:
        if random() > 0.5:  # мутація x
            if random() > 0.5:
                self.x += 1
            else:
                self.x = 1
        else:  # інакше - мутація y
            if random() > 0.5:
                self.y += 1
            else:
                self.y = 1
    def str(self) -> str:
        return f"X : {self.x} Y: {self.y} Fitness: {self.fitness()}"
```

SimpleEquation відповідає Chromosome. Гени хромосоми SimpleEquation можна розглядати як x та y .

Метод fitness() оцінює життєздатність x і y з допомогою рівняння 6 x - x 2 + 4 y - y 2 . Чим більше значення, тим вище життєздатність даної хромосоми _ відповідно з GeneticAlgorithm. Для x і y спочатку присвоюються випадкові цілі числа з діапазону 0… 100, тому random_instance() потрібно тільки щоб

створити новий екземпляр SimpleEquation з цими значеннями. Щоб схрестити два SimpleEquation у crossover(), значення у цих екземплярів просто змінюються місцями для створення двох дочірніх елементів. mutate() випадковим чином збільшує або зменшує x або y.

Оскільки SimpleEquation відповідає Chromosome, то ми відразу можемо підключити його до GeneticAlgorithm (Літинг 5.10).

Лістинг 2.10. simple_equation.py (продовження)

```
if name == "main":
  _initial_population: List[SimpleEquation] = [SimpleEquation.random
_instance() for _ in range(20)]
  ga: GeneticAlgorithm[SimpleEquation] = GeneticAlgorithm(initial_population
= initial_population, threshold = 13.0, max_generations = 100, mutation_chance =
0.1, crossover_chance = 0.7)
  result: SimpleEquation = ga.run()
  print(result)
```

Використані тут параметри були отримані методом припущення та перевірки. Значення threshold було встановлено рівним 13. При x = 3 і y = 2 значення рівняння дорівнює 13.

У цьому випадку можна встановити поріг рівним довільному великому числу. Оскільки генетичні алгоритми є стохастичними, усі проходи алгоритму будуть різними.

Ось приклад вихідних даних одного з проходів, при якому генетичний алгоритм вирішив рівняння за дев'ять поколінь:

Generation 0 Best 349 Avg 6112.3

Generation 1 Best 4 Avg 1306.7

Generation 2 Best 9 Avg 288.25

Generation 3 Best 9 Avg 7.35

Generation 4 Best 12 Avg 7.25

Generation 5 Best 12 Avg 8.5

Generation 6 Best 12 Avg 9.65

Generation 7 Best 12 Avg 11.7

Generation 8 Best 12 Avg 11.6

X: 3 Y: 2 Fitness: 13

Майже в кожному поколінні алгоритм постійно наближався до правильної відповіді.

Слід зважити, що для пошуку рішення генетичний алгоритм потребує більше обчислювальних ресурсів, ніж інші методи. Тому застосування генетичного алгоритму для дуже простого завдання не є доцільним. Але цієї простої реалізації достатньо, щоб показати що наш генетичний алгоритм працює.

2.3 Деякі проблеми генетичних алгоритмів

Для будь-якого завдання, для якого існує швидкий детермінований алгоритм, підхід з використанням генетичного алгоритму не має сенсу. Стохастична природа генетичних алгоритмів не дозволяє передбачити час виконання. Щоб вирішити цю проблему, можна переривати роботу алгоритму через певну кількість поколінь. Але тоді залишається незрозумілим чи знайдено дійсно оптимальне рішення.

Недоліки у порівнянні з іншими методами оптимізації:

- Пошук оптимального рішення для складної задачі високої розмірності часто вимагає великих витрат. В реальних задачах обчислення можуть тривати від кількох годин до кількох днів.
- Генетичні алгоритми погано масштабуються під складність вирішуваної проблеми. При збільшенні області пошуку рішень збільшується число елементів, що піддаються до обробки.
- Умови зупинки алгоритму є різними для кожної проблеми.
- В багатьох завданнях генетичні алгоритми мають тенденцію сходитися до локального оптимуму замість глобального оптимуму для даної задачі.

2.4 Бінарне представлення

2.4.1 Поняття схеми Холланда

Сформулюємо основну теорему, що відноситься до генетичних алгоритмів і називається теоремою про схеми.

Поняття схема було введено Холландом та використовується для аналізу роботи ГА. Зокрема розглядаються процеси конструювання та руйнування певної схеми протягом розвитку популяції (schema dynamics).

Схемою називається рядок виду (a_1, a_2, ..., a_i, ..., a_l), a_i ¿ {0, 1, *}.

Символом "*" в деякому розряді позначається факт, що там може бути як 1, так і 0. Наприклад, для двох бінарних рядків "111000111000" та "110011001100" схема виглядатиме таким чином: "11*0****1*00". Тобто за допомогою схем можна виділяти загальні ділянки двійкових рядків і маскувати відмінності. Маючи у складі схем m символів "*" можна закодувати (узагальнити) 2m двійкових рядків. Так, наприклад, схема "01 * 0 * 1" описує набір рядків

{"010001", "010011", "011001", "011011"}.

Випереджувальною довжиною схеми (schema defining length) називається відстань між двома крайніми символами "0" та/або "1". Для схеми "01*0*1" визначальна довжина дорівнює 5, а для схеми "**0**1*" визначальна довжина дорівнює 3. Порядок схеми (schema order) - це ще одна характеристика схеми і дорівнює числу фіксованих позицій в рядку, тобто. загальному числу "0" та "1". Для схем "01*0*1" та "**0**1*" порядки дорівнюють 4 і 2 відповідно.

2.4.2 Базові представлення хромосоми

Хромосома, по суті, є двійковим рядком. У той же час особині, якій належить хромосома, що містить набір генів-параметрів завдання, поставлена у відповідність величина, що характеризує її пристосованість. Оскільки схема є узагальненням кількох бінарних рядків (хромосом), можна говорити, що особини, які мають хромосоми, які відповідають одній схемі більш пристосовані, а особини з хромосомами, відповідними іншій схемі - менш пристосовані.

Можна сказати, що сенс роботи ГА полягає в пошуку двійкового рядка

певного виду з множини бінарних рядків. Простір пошуку становить 2L рядків, вимірність якого дорівнює L (L-мірний простір), де L - довжина хромосоми. Схема відповідає певній гіперплощині у цьому просторі. Дане твердження можна проілюструвати в такий спосіб. Нехай розрядність хромосоми дорівнює 3, тоді можна закодувати всього $2^3=8$ рядків. Представимо куб у 3-мірному просторі. Позначимо вершини цього куба 3-розрядними бінарними рядками так, щоб мітки сусідніх вершин відрізнялися рівно на один розряд, причому вершина з міткою "000" була б на початку координат. Варіант позначення зображено на рис.

Рис.2.4. Чотиримірний куб

Якщо взяти схему виду "**0", вона опише ліву грань куба, а схема "*10" - верхнє ребро цієї грані. Очевидно, що схема "***" відповідає всьому простору. Якщо взяти двійкові рядки довжиною 4 розряду, то розбиття простору схемами можна зобразити з прикладу 4-мірного куба з іменованими вершинами (рис.2.4).

Рис.2.5. 4-мірний куб

Тут схемі "*1**" відповідає гіперплощина, що включає задні грані зовнішнього та внутрішнього куба, а схемі "**10" - гіперплощина з верхніми ребрами лівих граней обох кубів.

Розбиття простору пошуку можна уявити і інакше. Представимо координатну площину, в якій по одній осі ми відкладатимемо значення двійкових рядків, а по іншій - значення цільової функції (рис. 2.5.).

Рис. 2.6. Розбиття простору

Ділянки простору, заштриховані різним стилем, відповідають різним схемам. Число до правої частини горизонтальної осі відповідає максимальному значенню бінарного рядка - "111...111". З малюнка видно, що схема "0***...*" покриває всю ліву половину відрізка, схема "**1*...*" - 4

ділянки завширшки одну восьму частину, а схема "0*10*. ..*" - ліві половини ділянок, які перебувають на перетині перших двох схем.

Поняття схема було запроваджено як визначення множини хромосом, які мають деякі загальні властивості, тобто подібних один до одного. Якщо алелі приймають значення 0 або 1 (розглядаються хромосоми з двійковим алфавітом), то схема є множиною хромосом, що містять нулі та одиниці на деяких заздалегідь визначених позиціях. При розгляді схем зручно використовувати розширений алфавіт {0, 1, *}, в який крім 0 і 1 введено додатковий символ *, що означає будь-яке допустиме значення, тобто 0 або 1; символ * у конкретній позиції означає "все одно" (don't care). Наприклад,

10*1 ={1001, 1011}

*01*10 = {001010, 001110, 101010,101110}

Вважається, що хромосома належить до цієї схеми, якщо кожній j-й позиції (локуса), j = 1, 2, ..., L, де L - довжина хромосоми; символ, що займає j-ю позицію хромосоми, відповідає символу, що займає j-ту позицію схеми, причому символу відповідають як 0, так і 1. Те ж саме означають твердження про те, що хромосома відповідає схемі і хромосома представляє схему. Відзначимо, що якщо у схемі присутній символів *, то ця схема містить 2^r хромосом. Крім того, кожна хромосома (ланцюжок) довжиною L належить до 2^L схемам. У таблицях 2 та 3 представлені схеми, до яких належать ланцюжки довжиною 2 та 3 відповідно.

Таблиця 2.3. Схеми, до яких належать ланцюжки довжиною 2

Ланцюжки	1 схема	2 схема	3 схема	4 схема
00	**	*0	0*	00
01	**	*1	0*	01
10	**	*0	1*	10
11	**	*1	1*	11

Таблица 2.4. Схеми, до яких належать ланцюжки довжиною 3

Ланцюжки	Схеми							
	1	2	3	4	5	6	7	8
000	***	**0	*0*	0**	*00	0*0	00*	000
001	***	**1	*0*	0**	*01	0*1	00*	001
010	***	**0	*1*	0**	*10	0*0	01*	010
011	***	**1	*1*	0**	*11	0*1	01*	011
100	***	**0	*0*	1**	*00	1*0	10*	100
101	***	**1	*0*	1**	*01	1*1	10*	101
110	***	**0	*1*	1**	*10	1*0	11*	110
111	***	**1	*1*	1**	*11	1*1	11*	111

Ланцюжки довжиною 2 відповідають чотирьом різними схемам, а ланцюжки довжиною 3 - восьми схемам.

Генетичний алгоритм базується на принципі трансформації найбільш пристосованих особин (хромосом). Нехай P(0) означає вихідну популяцію особин, а P(k) - поточну популяцію (k-й ітерації алгоритму). З кожної популяції P(k), k = 0, 1, ... методом селекції вибираються хромосоми з найбільшою пристосованістю, які включаються в так званий батьківський пул (mating pool) M(k). Далі, в результаті об'єднання особин з популяції M(k) у батьківські пари та виконання операції схрещування з ймовірністю p_c, а також операції мутації з ймовірністю $p_т$ формується нова популяція P(k+1), до якої входять нащадки особин із популяції M(k). Тобто для будь-якої схеми було б бажаним, щоб кількість хромосом, що відповідають цій схемі, зростала зі збільшенням кількості ітерацій k.

На відповідне перетворення схем у генетичному алгоритмі впливають 3 фактори: селекція хромосом, схрещування та мутація. Проаналізуємо вплив кожного з них з точки зору збільшення очікуваної кількості представників окремо взятої схеми.

Позначимо схему, що розглядається, символом *S*, а кількість хромосом популяції *P(k),* відповідних цій схемі - *c(S, k)*. Отже, *c(S, k)* можна вважати кількістю елементів (тобто потужністю) множини P*(k)* ¿ S.

2.4.3 Деякі аспекти селекції

Дослідимо вплив селекції. За виконання цієї операції хромосоми з популяції *P(k)* копіюються в батьківський пул M*(k)* з ймовірністю, що визначається виразом (3.3). Нехай *F(S, k)* позначає середнє значення функції пристосованості хромосом із популяції *P(k),* які відповідають схемі S. Якщо *P(k)S* = {ch$_1$.....ch$_{c(S,k)}$}, то

$$c(S,k)=\frac{\sum_{i=1}^{c(S,k)} F(ch_i)}{c(S,k)} \qquad (2.1)$$

Величина *F(S, k)* також називається пристосованістю схеми S на k ітерації. Нехай $\Im$ (k) позначає суму значень функцій пристосованості хромосом із популяції *P(k)* потужністю N, тобто

$$\Im(k) = \sum_{i=1}^{N} F(ch_{i(k)}) \qquad (2.2)$$

Позначимо через $\bar{F}$ (k) середнє значення функції адаптації хромосом цієї популяції, тобто

$$\bar{F}(k) = \frac{1}{N}\Im(k). \qquad (2.3)$$

Нехай $ch_i^{(k)}$ позначає елемент батьківського пулу *M(k).* Для кожного $ch_r^{(k)}$ ¿ *M(k)* і для кожного *i* = 1,..., *c(S, к)* ймовірність того, що $ch_r^{(k)}=ch_i$ визначається ставленням *F(ch$_i$)I F(k).* Тому очікувана кількість хромосом у популяції M*(k),* які рівні ch$_i$, складе

$$N\frac{F(ch_i)}{\Im(k)} = \frac{F(ch_i)}{\bar{F}(k)}$$

Таким чином, очікувана кількість хромосом з множинитва P*(k)* ¿ S, відібраних для включення до батьківського пулу *M(k),* буде дорівнювати

$$\sum_{i=1}^{c(S,k)} \frac{F(ch_i)}{\bar{F}(k)} = c(S,k)\frac{F(S,k)}{\bar{F}(k)},$$

що випливає з виразу (2.1).

Оскільки кожна хромосома з батьківського пулу M*(k)* одночасно належить популяції P*(k)*, то хромосоми, що становлять множину *M(k)* ¿ *S* - це ті самі особини, які були відібрані з множини *P*(k) ¿ *S* для включення в популяцію *M(k)*. Якщо кількість хромосом батьківського пулу M*(k),* відповідних схемам S (тобто кількість елементів множини M*(k)* ¿ *S),* позначити *b(S, k),* то з наведених міркувань можна зробити наступні висновки.

Властивість 2.1

Очікуване значення *b (S , k),* тобто очікуване значення кількості хромосом батьківського пулу *M(к),* відповідних схемі S визначається виразом

$$E[b(S,k)] = c(S,k)\frac{F(S,k)}{\bar{F}(k)} \quad .(3.9)$$

З цього випливає, що якщо схема S містить хромосоми зі значенням функції пристосованості, що перевищує середнє значення (тобто пристосованість схеми S на k-й ітерації виявляється більшою, ніж середнє значення функції пристосованості хромосом з популяції *P(k),* і тому *F (S , k) IF (k)*> 1), то очікувана кількість хромосом з батьківського пулу *M (k),* відповідних схемі S , буде більше кількості хромосом з популяції *P (k),* відповідних схемі S . Тому можна стверджувати, що селекція викликає поширення схем із пристосованістю «краще середньої» та зникнення схем із «гіршою» пристосованістю.

Визначимо необхідні для подальших міркувань поняття щодо порядка та охоплення схеми. Нехай *L* позначає довжину хромосом, що відповідають - схемі S .

Визначення 2.1

Порядок (order) схеми - кількість постійних позицій у схемі, тобто нулів та одиниць у алфавіті {0, 1, *}. Наприклад,

o(10*1) = 3, o(*01*10) = 4 ,o(**0*Г) = 2, o(*101**) = 3.

Порядок схеми *o* (S) дорівнює довжині *L* , якщо відняти кількості символів *, що легко перевірити на наведених прикладах (для *L* = 4 з одним

символом * і для L = 6 з двома, чотирма та трьома символами *). Легко помітити, що порядок схеми, що складається виключно із символів *, дорівнює нулю, тобто

o (****) = 0, а порядок схеми без жодного символу * дорівнює L ; наприклад, o (10011010) = 8. Порядок схеми o (S) - це завжди ціле число з інтервалу [0, L] .

Визначення 2.2

Охоплення (defining length) схеми S , званий також довжиною позначається *d(S)* - це відстань між першим і останнім постійним символом (тобто різниця між правою і лівою крайніми позиціями, що містять постійні символи). Наприклад,

d (10 * 1) = 4-1 = 3

d (**0 * 1 *) = 5-3 = 2

d (*01*10) = 6-2 = 4

d (*101**) = 4-2 = 2

Охоплення схеми *d (S)* - це ціле число з інтервалу [0, L - 1]. Зазначимо, що охоплення схеми з постійними символами на першій і останній позиції дорівнює L - 1 (як у першому з наведених прикладів). Охоплення схеми з єдиною постійною позицією дорівнює нулю, зокрема, *d (**1*)*= 0. Охоплення характеризує змістовність інформації, укладеної у схемі.

2.4.4 Вплив схрещування на обробку схем

Перейдемо до міркувань щодо впливу операції схрещування на обробку схем у генетичному алгоритмі. Насамперед зазначимо, що одні схеми виявляються більш схильними до знищення в процесі схрещування, ніж інші. Наприклад , розглянемо схеми *S1* = 1****0* і S_2 = **01***, а також хромосому з h =[1001101], що відповідає обом схемам. Видно, що схема S_2 має більше шансів «пережити» операцію схрещування, ніж схема S_1 , яка більше схильна до «розщеплення» в точках схрещування 1, 2, 3, 4 і 5. Схему - S_2 можна розділити тільки при виборі точки схрещування 3.

У ході аналізу впливу операції схрещування на батьківський пул $M(k)$ розглянемо деяку хромосому з множини $M(k) \in S$, тобто хромосому з батьківського пулу, що відповідає схемі S. Імовірність того, що ця хромосома буде відібрана для схрещування дорівнює p_c. Якщо жоден із нащадків цієї хромосоми не належатиме до схеми S, то це означає, що точка схрещування - повинна перебувати між першим і останнім постійним символом даної схеми. Імовірність цього дорівнює $d(S)/(l-1)$. З цього можна зробити такі висновки:

Властивість 2.2 (вплив схрещування)

Для деякої хромосоми з $M(k) \in S$ ймовірність того, що вона буде відібрана для схрещування і жоден з її нащадків не буде належати до схеми S, обмежена зверху величиною

$$p_c \frac{d(S)}{L-1}$$

Ця величина називається ймовірністю знищення схеми *S*.

Властивість 2.3

Для деякої хромосоми з $M(k) \in S$ ймовірність того, що вона не буде відібрана для схрещування або, що хоча б один з її нащадків після схрещування буде належати до схеми S, обмежена знизу величиною

$$1-p_c \frac{d(S)}{L-1}$$

Ця величина називається ймовірністю виживання схеми *S*.

Легко показати, що якщо дана хромосома належить до схеми S і відбирається для схрещування, а друга батьківська хромосома також належить до схеми S, то обидва їх нащадки теж належать до схеми S. Висновки 3.2 і 3.3 підтверджують значущість показника охоплення схеми *d(S)* для оцінки ймовірності знищення або виживання схеми.

2.4.5 Вплив мутації на батьківський пул

Розглянемо тепер вплив мутації на батьківський пул $M(k)$. Оператор мутації з ймовірністю $p_т$ випадковим чином змінює значення конкретної

позиції з 0 на 1 і назад. Очевидно, що схема переживе мутацію тільки в тому випадку, коли всі її постійні позиції залишаться після виконання цієї операції незмінними.

Хромосома з батьківського пулу, що належить до схеми S (тобто. хромосома з множини $M(k) \in S$) залишиться в цій схемі тоді і тільки тоді, коли жоден символ цієї хромосоми, відповідний постійним символам схеми S , не зміниться в процесі мутації. Імовірність такої події дорівнює $(1- p_m)^{o(S)}$. Цей результат можна подати у формі наступного висновку:

Властивість 2.4 (вплив мутації)

Імовірність того, що деяка хромосома з $M(k) \in S$ буде належати до схеми S після операції мутації, визначається виразом $(1- p_m)^{o(S)}$.

Ця величина називається ймовірністю виживання схеми S після мутації.

Властивість 2.5

Якщо ймовірність мутації $p_т$ мала ($p_m \ll 1$), можна вважати, що ймовірність виживання схеми S після мутації, визначена у висновку 3.4, приблизно дорівнює $1 - p_т\, o(S)$.

Ефект спільного впливу селекції, схрещування та мутації (висновки 3.1 - 3.4) з урахуванням факту, що якщо хромосома з множини $M(k) \in S$ дає нащадка, відповідного схемою S , він буде належати до $P(k+1) \in S$, веде до побудови наступної схеми репродукції :

$$E[c(S,k+1)] \geq c(S,k) \frac{F(S,k)}{\bar{F}(k)} \left(1-p_c \frac{d(S)}{L-1}\right)(1-p_m)^{o(s)}. \quad (3\ 10)$$

Залежність (3.10) показує, як змінюється від популяції до популяції кількість хромосом, що відповідають цій схемі. Ця зміна викликається трьома факторами, представленими у правій частині виразу (3.10), зокрема: $F(S,k)/\bar{F}(k)$ відображає роль середнього значення функції пристосованості, $1 - p_c\, d(S)/(L - 1)$ показує вплив схрещування та $(1- p_m)^{o(S)}$ - вплив мутації. Чим більше значення кожного з цих факторів, тим більшим виявляється очікувана

кількість відповідностей схемі S у наступній популяції. Висновок 3.5 дозволяє уявити залежність (3.10) у вигляді

$$E[c(S,k+1)] \geq c(S,k) \frac{F(S,k)}{\bar{F}(k)} (1-p_c \frac{d(S)}{L-1} - p_m o(s)). \quad (2.4)$$

Для великих популяцій залежність (2.4) можна апроксимувати виразом

$$c(S,k+1) \geq c(S,k) \frac{F(S,k)}{\bar{F}(k)} (1-p_c \frac{d(S)}{L-1} - p_m o(s)). \quad (2.5)$$

З формул (2.4) і (2.5) випливає, що очікувану кількість хромосом, відповідних схемі S в наступному поколінні, можна вважати функцією від фактичної кількості хромосом, що належать цій схемі, відносної пристосованості схеми, а також порядку та охоплення схеми. Помітно, що схеми з пристосованістю вище за середню і з малим порядком та охопленням характеризуються зростанням кількості своїх представників у наступних популяціях. Подібне зростання має показовий характер, що випливає з виразу (2.5). Для великих популяцій цю формулу можна замінити рекурентною залежністю виду

$$c(S, k+1) = c(S, k) \frac{F(S,k)}{\bar{F}(k)} \quad (2.6)$$

Якщо припустити, що схема S має пристосованість на $\%$ вище середньої, тобто.

$$F(S, k) = \bar{F}(k) + \varepsilon \bar{F}(k), \quad (2.7)$$

то при підстановці виразу (2.6) у нерівність (2.7) у припущенні , що ε не змінюється в часі, при старті від $k = 0$ отримуємо

$c(S, k) = c(S, 0)(1 + \varepsilon)^{\kappa}$,

$$\varepsilon = (F(S,k) - \bar{F}(k))/\bar{F}(k), \quad (2.8)$$

тобто $\varepsilon > 0$ для схеми з пристосованістю вище за середню і $\varepsilon < 0$ - в іншому випадку.

Рівність (2.8) визначає геометричну прогресію. З цього випливає, що в процесі репродукції схеми, що виявилися кращими (гіршими) за середні, вибираються на чергових ітераціях генетичного алгоритму в показово зростаючих (зменшуваних) кількостях. Зауважимо, що залежності (2.6) - (2.8)

засновані на припущенні, що функція пристосованості F набуває лише додатніх значень. При використанні генетичних алгоритмів для вирішення оптимізаційних завдань, в яких цільова функція може приймати і відємні значення, необхідні деякі додаткові співвідношення між функцією, що оптимізується, і функцією пристосованості.

2.4.6 Теорема про схеми

Кінцевий результат, який отримується з виразів (3.10) - (3.12), можна сформулювати у формі теореми. Це основна теорема генетичних алгоритмів, інакше звана теорема про схеми .

Теорема 2.1

Схеми малого порядку, з малим охопленням і з пристосованістю вище середньої формують зростаючу послідовність кількості своїх представників у наступних поколіннях генетичного алгоритму.

Відповідно до наведеної теореми важливим питанням стає кодування, яке має забезпечувати побудову схем малого порядку, з малим охопленням і з пристосованістю вище за середню. Непрямим результатом теореми 3.1 (про схеми) можна вважати наступну гіпотезу, звану гіпотезою про цеглини (або про будівельні блоки) .

Гіпотеза 2.1

Генетичний алгоритм прагне досягти близького до оптимального результату за рахунок комбінування хороших схем (з здатністю вище середньої) малого порядку та малого охоплення. Такі схеми називаються цеглинками (або будівельними блоками).

Гіпотеза про будівельні блоки висунута на підставі теореми про схеми з урахуванням того, що генетичні алгоритми досліджують простір пошуку за допомогою схем малого порядку та малого охоплення, які згодом беруть участь в обміні інформацією при схрещуванні.

Незважаючи на те, що для доказу цієї гіпотези робилися певні дослідження, проте в більшості не тривіальних додатків доводиться спиратися на емпіричні результати. Протягом останніх двадцяти років

опубліковано численні роботи, присвячені застосуванню генетичних алгоритмів, що підтверджують цю гіпотезу. Якщо вона вважається істинною, то проблема кодування набуває критичного значення для генетичного алгоритму; кодування повинне реалізувати концепцію малих будівельних блоків. Якість, що забезпечує генетичним алгоритмам явну перевагу над іншими традиційними методами, безперечно полягає в обробці великої кількості різних схем.

Виконання генетичних алгоритмів ґрунтується на обробці схем. Схеми малого порядку, з малим охопленням і пристосованістю вище середньої вибираються, розмножуються і комбінуються, в результаті чого формуються всі кращі кодові послідовності. Тому оптимальне рішення будується (відповідно до гіпотези цеглинок) шляхом об'єднання найкращих з отриманих до поточного моменту часткових рішень. Просте схрещування не надто часто знищує схеми з малим охопленням, проте ліквідує схеми із досить великим охопленням. Однак незважаючи на згубність операцій схрещування та мутації для схем високого порядку та охоплення, кількість оброблюваних схем настільки велика, що навіть при відносно низькій кількості хромосом у популяції досягаються дуже непогані результати виконання генетичного алгоритму.

Кількість схем, що ефективно обробляються, розрахована Холландом , становить $O(N^3)$. Це означає, що для популяції потужністю N кількість оброблюваних у кожному поколінні схем має порядок N^3 .

2.5 Еволюційні стратегії

Серед стратегій, що використовуються в сучасних генетичних алгоритмах, можна виділити стратегії елітізма, різноманіття, "свіжої крові", зміни розміру популяції, паралельні еволюції (міграції, турнірна).

Самою поширеною є стратегія елітизму, коли на всіх етапах еволюційного процесу зберігається елітна група з кращих особин популяції. Вважається, що краща особина містить корисний генетичний матеріал, який може бути використаний для пошуку оптимуму. Таким чином, використання

стратегії елітізму запобігає можливості видалення з популяції кращих особин при формуванні нового покоління.

Розмір елітної групи визначається особливостями алгоритму і залежить, як правило, від загальної чисельності популяції. В процесі роботи алгоритму потрібно стежити, щоб особини елітної групи не виродились в однакові або подібні генотопи.

Використання стратегії різноманіття вирішує проблему утворення в популяції множини однакових або подібних осіб. Ці особини звужують область пошуку рішення та приводять до більш сильного виродження наступних поколінь.

Технічно дана стратегія реалізується на кожному етапі еволюційного процесу попарним оцінюванням ступеня близькості всіх особин. Якщо між кількома особинами є подібність, яка оцінюється нижче за деяке задане порогове значення, з них залишається тільки одна особина (краща). Вакантні місця заповнюються нащадками, що отримані на поточній стадії еволюції або, за необхідності, новими випадковими рішеннями.

Стратегія "свіжої крові" полягає у введенні в нове покоління популяції певної кількості нових, випадково отриманих особин замість групи батьківських. Процедура може виконуватися систематично з заданим інтервалом між поколіннями або випадково на будь-якій стадії еволюції. Стратегія «свіжого крові» не повинна вступати в протиріччя з стратегією елітізма, тому, серед вилучених батьківських особин не повинно бути елітних.

Стратегія зміни розміру популяції передбачає нестабільність кількості особин, що беруть участь у процесі на кожній стадії еволюції. Кількість особин змінюється залежно від якісних характеристик популяції в цілому. Цими характеристиками можуть бути середня пристосованість або її зміна протягом певної кількості стадій еволюції.

Наприклад, при підвищенні середньої пристосованості популяції або динаміки її зміни об'єм популяції зростає. Відповідно, при погіршенні -

зменшується. Незалежно від правила зміни розміру популяції, повинні бути встановлені нижня і верхня межі, а також початкова кількість особин.

Стратегії паралельних еволюцій (міграційна, турнірна) скеровані на формування незалежних або обмежено залежних груп, всередині яких протікає еволюційний процес. В кожній групі можуть застосовуватися власні налаштування генетичного алгоритму, що може сприяти знаходженню кращого рішення. В подальшому відбувається перенесення генетичної інформації з однієї групи в іншу або їх об'єднання.

При реалізації стратегії міграції вся популяція поділяється на кілька груп, зазвичай, з різною кількістю особин. Оператори кросинговеру застосовуються до особин строго зі своїх спільнот. До окремих особин застосовуються різні оператори мутації і інверсії. Процеси еволюції відбуваються незалежно, кожна в своїй групі. І лише в деяких випадках (з невеликою ймовірністю) відбувається обмін генетичним матеріалом окремих особин між групами. Протягом еволюції збирається банк кращих рішень, які потім об'єднуються в єдину популяцію. Для популяції кращих рішень реалізується свій еволюційний процес.

Турнірна стратегія передбачає проведення повністю незалежних закінчених еволюційних процесів в кількох групах і об'єднання отриманих в кожній групі кращих рішень в одну або кілька нових груп з виконанням наступного етапу еволюції. Кількість таких етапів в загальному випадку може бути довільною і обмежується лише вимогами часу і обчислювальними можливостями. Природно, чим більше етапів, тим більша ймовірність знаходження глобального оптимуму. Розміри груп також можуть бути різними: від кількох десятків до кількох одиниць особин.

Еволюційне програмування було запропоновано Лоуренсом Дж. Фогелем в 1960 році. У той час штучний інтелект було обмежено двома основними напрямками досліджень: моделюванням людського мозку (нейронні мережі) і моделюванням поведінки людини (евристичне програмування). Альтернативний варіант Фогеля був спрямований на

моделювання процесу еволюції, як засобу отримання розумної поведінки. Фогель розглядає інтелект як складову частину здатності робити передбачення відповідно до заданої мети. На його думку прогнозування є необхідною умовою для розумної поведінки.

Гіпотези про вид залежності цільової змінної від інших змінних формулюються системою у вигляді програм. Процес побудови таких програм розглядається як еволюція в популяції програм. Якщо система знаходить програму, яка точно відтворює шукану залежність, вона починає вносити до неї невеликі модифікації і відбирає серед дочірніх програм лише ті, які підвищують точність.

Система "вирощує" кілька генетичних ліній програм, що конкурують між собою в точності знаходження шуканої залежності. Спеціальний транслюючий модуль перекладає знайдені залежності з внутрішньої мови системи на зрозумілу користувачеві мову (математичні формули, таблиці тощо), роблячи їх наочними. Для того, щоб зробити отримані результати більш зрозумілими, існує великий арсенал різноманітних засобів візуалізації виявлених залежностей.

Пошук залежності цільових змінних від інших факторів проводиться у формі функцій певного виду. Наприклад, в одному з найбільш вдалих алгоритмів цього типу - методі групового урахування аргументів (МГУА) залежність шукають у формі поліномів. Причому складні поліноми заміняються кількома простими, враховують лише деякі ознаки (групи аргументів). Отримані формули залежностей надаються до аналізу та інтерпретації.

Еволюційне програмування застосовне до різних інженерних завдань:Системи управління, системи ідентифікації. Маршрутизація трафіку.Військове планування.Обробка сигналів.Ігрові та навчальні програми.

Основною перевагою еволюційних методів оптимізації є можливості вирішення завдань з великою розмірністю за рахунок поєднання елементів

випадковості і строго визначених правил, основним з яких є закон еволюції: «перемагає найсильніший».

Іншим важливим чинником ефективності є відтворення процесів розвитку, де знайдені рішення за певним правилами породжують нові рішення, які будуть наслідувати кращі риси попередників. Як випадковий елемент використовується моделювання процесу мутації. З її допомогою характеристики певного рішення можуть бути випадково змінені, що надає новий напрямок в процесі еволюції і може пришвидшити процес вироблення кращого рішення.

Переваги еволюційних алгоритмів наступні. Наочність схеми і базових принципів еволюційних обчислень.Придатність для пошуку в складному просторі рішень великої розмірності.Можливість підбору початкових умов, комбінування еволюційних обчислень, продовження процесу еволюції доки є необхідні ресурси.Відсутність обмежень на вид цільової функції.Широка область застосування.

Недоліки еволюційних алгоритмів. Еволюційні обчислення не гарантують оптимальності отриманого рішення, але за заданий час можна отримати кілька хороших альтернативних рішень.Вимагається висока обчислювальна потужність.Відносно невисока ефективність на завершальних фазах еволюційного процесу.

Розділ 3 Криптоарифметика та генетичні алгоритми

3.1 Класичний розв'язок криптоарифметичної задачі

3.1.1 Завдання з обмеженнями

Багато задач, для рішення яких використовуються комп'ютерні обчислення, можна загалом віднести до категорії завдань з обмеженнями (constraint-satisfaction problems, CSP). CSP-завдання складаються з змінних, допустимі значення яких потрапляють в певні діапазони, відомі як області визначення. Для того щоб вирішити завдання з обмеженнями, необхідно задовольнити існуючі обмеження для змінних. Три основні поняття — змінні, області визначення та обмеження — прості та зрозумілі, а завдяки їхній універсальності завдання з обмеженнями набули широкого застосування.

Припустимо, що у нас є задача - призначити на п'ятницю зустріч для Джо, Мері і Сью. Сью повинна зустрітися хоча б із однією людиною. Областю визначення кожної змінної можуть бути час, коли вільний кожен з них. Наприклад, у змінної «Мері» область визначення складає 2, 3 і 4 години дня. У цього завдання є також два додаткових обмеження. По перше, Сью повинна бути присутня на зустрічі. По-друге, на зустрічі повинні бути присутні по крайньої мірі два людини. Рішення цього завдання з обмеженнями визначається трьома змінними, трьома областями визначення та двома обмеженнями, тоді завдання буде вирішено і при цьому не доведеться пояснювати користувачеві, як саме (рис. 3.1).

У деяких мовах програмування, таких як Prolog та Picat, є вбудовані засоби для рішення завдань з обмеженнями. У інших мовах звичайним підходом є створення структури, яка включає пошук з поверненнями і кілька евристик для підвищення продуктивності пошуку. Ми працюємо з мовою програмування Python, тому для роботи ми спочатку створимо структуру для CSP-завдань, яка буде вирішувати їх простим рекурсивним пошуком із поверненнями. Потім скористаємося цією структурою на вирішення криптоарифметичних задач.

Рис. 3.1. Завдання планування - це класичне застосування структур для задоволення обмежень

3.1.2 Побудова структури для завдання з обмеженнями

Визначимо обмеження у вигляді класу Constraint. Кожне обмеження Constraint складається з змінних variables, які обмежує, і методу satisfied(), який перевіряє, чи воно виконується. Визначення того, чи виконується обмеження, є основною логікою, базовим в визначенні конкретного завдання з обмеженнями. Реалізацію за замовчуванням потрібно перевизначити, тому що ми визначаємо Constraint як абстрактний базовий клас. Анотаціх як базові класи не призначені для реалізації. Тільки їхні підкласи, які перевизначають та

реалізують свої абстрактні методи @abstractmethod, призначені для такого використання (лістинг 3.1).

Лістинг 3.1. csp.py

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V')  # Тип variable для змінної
D = TypeVar('D')  # Тип domain для області визначення

# Базовий клас для всіх обмежень
class Constraint(Generic[V, D], ABC):
# Змінні , для яких існує обмеження
   def init(self, variables: List[V]) -> None:
      self.variables = variables
# Необхідно перевизначити у підкласах
@abstractmethod
def satisfied(self, assignment: Dict[V, D]) -> bool:
   pass
```

Анотації грають роль шаблонів у ієрархії класів В інших мовах , таких як C++, вони отримали більше ширше розповсюдження чим в Python. По суті , вони з'явилися в Python приблизно на середині життєвого шляху мови . При цьому багато класів колекцій з стандартної бібліотеки Python реалізовані за допомогою абстрактних базових класів [Ramalho L. Fluent Python (O'Reilly, 2015)].

Центральним елементом нашої структури відповідності обмеженням буде клас з назвою CSP (Лістинг 3.2). В CSP фактично зібрані всі змінні, області визначення та обмеження. Клас CSP використовує універсальні засоби, щоб бути досить гнучким, працювати з будь-якими значеннями змінних та

областей визначення (де V - Це значення змінних, а D - областей визначення). У CSP колекції variables, domains та constraints мають очікувані типи. Колекція variables - це list для змінних, domains - dict з відповідністю змінних спискам можливих значень (областям визначення цих змінних), а constraints — dict, де кожній змінній відповідає list накладених на неї обмежень.

Лістинг 3.2. csp.py (продовження)

```
# Завдання з обмеженнями складається з змінних типу V,
# які мають діапазони значень , відомі як області визначення , # типу D та
обмежень , які визначають , _ чи допустимим
# вибір даної області визначення для даної змінної
class CSP(Generic[V, D]):

  def init(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
self.variables: List[V] = variables
    # змінні , які будуть обмежені self.domains : Dict [V, List[D]] = domains #
домен кожною змінної self.constraints : Dict [V, List[Constraint[V, D]]] = {}
    for variable in self.variables: self.constraints[variable] = []
    if variable not in self.domains:
      raise LookupError("Every variable should have a domain assigned to it.")

def add_ constraint(self, constraint: Constraint[V, D]) -> None:
  for variable in constraint.variables:
    if variable not in self.variables:
      raise LookupError(" Variable in constraint not in CSP")
    else:
      self.constraints[variable].append(constraint)
```

Ініціалізатор init () створює constraints dict. Метод add_constraint() переглядає усі змінні, до яких відноситься це обмеження, і додає себе у відповідність constraints для кожної такої змінної. Обидва методи мають

найпростішу перевірку помилок і викликають raise LookupError, якщо variable відсутня в області визначення або існує constraint для неіснуючої змінної.

Для того, щоб дізнатися, чи відповідає чи дана конфігурація змінних і вибраних значень області визначення заданим обмеженням необхідно розглянути наступне. Ми будемо називати таку задану конфігурацію привласненням. Нам потрібна функція, яка перевіряла б кожне обмеження для заданою змінної по відношенню до присвоюванню, щоб побачити, чи задовольняє значення змінної присвоювання цим обмеженням . У лістингу 3.3 реалізована функція constant() як метод класу CSP.

Лістинг 3.3. csp.py (продовження)

```
# Перевіряємо , відповідає чи привласнення значення , перевіряючи Усе обмеження # для даної змінної
def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
    for constraint in self.constraints[variable]:
        if not constraint.satisfied(assignment):
            return False
    return True
```

Метод constant() перебирає всі обмеження для цієї змінної (це завжди буде змінна, щойно додана до присвоєння) і перевіряє, чи виконується обмеження з огляду на нове присвоєння. Якщо присвоєння задовольняє всі обмеження, повертається True. Якщо будь-яке обмеження, накладене на змінну, не виконується, повертається значення False.

3.1.3 Пошук з поверненням

Для пошуку розв'язання задачі в такій структурі виконання обмежень використовуватиметься простий пошук із поверненнями. Повернення — це підхід, при якому, якщо пошук зайшов у глухий кут, ми повертаємося до останньої відомої точки, де було прийнято рішення, перед тим як зайти в глухий кут, і обираємо інший шлях.

Пошук із поверненням, реалізований у наступній функції backtracking_search(), є свого роду рекурсивним пошуком у глибину. Ця функція додається в якості методу в клас CSP (лістинг 3.4).

Лістинг 3.4. csp.py (продовження)

```
def backtracking_search(self, assignment: Dict[V, D] = {})-> Optional[Dict[V, D]]:
    # привласнення завершено , якщо існує присвоєння # для кожною змінної ( базовий випадок )
    if len(assignment) == len(self.variables):
        return assignment
        # отримати змінні з CSP, але не з присвоєння
    unassigned: List[V] = [v for v in self.variables if v not in assignment]
        # отримати Усе можливі значення області визначення # для першою змінної без присвоєння
    first: V = unassigned[0]
    for value in self.domains[first]:
        local_assignment = assignment.copy()
    local_assignment[first] = value
    # якщо ні - продовжуємо рекурсію
    if self.consistent(first, local_assignment):
        result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
    # якщо результат не знайдено , закінчуємо повернення
    if result is not None:
        return result
    return None
#Досліджуємо
backtracking_search()
    if len(assignment) == len(self.variables):
    return assignment
```

Базовий випадок для рекурсивного пошуку означає, що потрібно знайти правильне надання для кожною змінної. Зробивши це, ми повертаємо перший валідний екземпляр рішення (і не продовжуємо пошук).

unassigned: List[V] = [v for v in self.variables if v not in assignment]

first: V = unassigned[0]

Щоб вибрати нову змінну, область визначення якої будемо досліджувати, ми просто переглядаємо усі змінні і знаходимо першу, яка не має присвоєння. Для цього створюємо список list змінних у self.variables, але не в assignment через генератор списків, і називаємо його unassigned. Потім вилучаємо з unassigned перше значення.

for value in self.domains [перший]: local_assignment = assignment.copy () local_assignment [first] = value

Ми намагаємось присвоїти цій змінній усі можливі значення області визначення по черзі. Нове присвоєння для кожної змінної зберігається у локальному словнику local_assignment.

```
if self.consistent (first, local_assignment ):
        result: Optional[ Dict [V, D]] =
        self.backtracking _search ( local_assignment )
if result is not None :
        return result
```

Якщо нове привласнення в local_assignment узгоджується з усіма обмеженнями , що перевіряється з допомогою consistent(), ми продовжуємо рекурсивний пошук для нового присвоєння. Якщо нове присвоєння виявляється завершеним (базовий випадок), передаємо його вгору ланцюжком рекурсії.

return None

Зрештою, якщо ми розглянули усі можливі значення області визначення для конкретної змінної і не виявили рішення, в якому використовувався б існуючий набір призначень, то повертаємо None, що вказує на відсутність

рішення. У результаті по ланцюжку рекурсії буде виконаний повернення до точці, в якої могло бути прийнято інше попереднє привласнення.

3.1.4 Розв'язок SEND + MORE = MONEY

Розглянемо найпростішу криптоарифметичну задачу.

SEND + MORE = MONEY - це криптоарифметична головоломка, де потрібно знайти такі цифри, які, будучи підставленими замість літер, зроблять математичне твердження вірним. Кожна літера в задачі представляє одну цифру від 0 до 9. Жодні дві різні літери не можуть представляти одну й ту саму цифру. Якщо літера повторюється, це означає, що цифра у рішенні також повторюється. Щоб простіше було вирішити це завдання вручну, варто побудувати слова у стовпчик:

SEND
+MORE
=MONEY

Завдання легко вирішується вручну, потрібно лише трохи алгебри і інтуїції. Але досить проста комп'ютерна програма допоможе зробити це швидше, використовуючи безліч можливих рішень.

Представимо SEND + MORE = MONEY як завдання з обмеженнями (лістинг 3.5).

Лістинг 3.5. send_more_money.py

```
from csp import Constraint, CSP
from typing import Dict, List, Optional
class SendMoreMoneyConstraint(Constraint[str, int]):
  def init(self, letters: List[str]) -> None:
    super().init(letters)
    self.letters: List[str] = letters
  def satisfied(self, assignment: Dict[str, int]) -> bool:
    # якщо є повторювані значення , то рішення не підходить
    if len(set(assignment.values())) < len(assignment):
      return False
```

```
        # якщо усі змінні призначені , перевіряємо чи правильна сума
        if len(assignment) == len(self.letters):
            s: int = assignment["S"]
            e: int = assignment["E"]
            n: int = assignment["N"]
            d: int = assignment["D"]
            m: int = assignment["M"]
            o: int = assignment["O"]
            r: int = assignment["R"]
            y: int = assignment["Y"]
            send: int = s * 1000 + e * 100 + n * 10 + d
            more: int = m * 1000 + o * 100 + r * 10 + e
            money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
            return send + more == money
        return True
```

Метод SendMoreMoneyConstraint satisfied() виконує кілька дій. По-перше, він перевіряє, чи різні літери відповідають різним цифрам. Якщо ні, то рішення неправильне і метод повертає False. Потім метод перевіряє чи всім буквам призначені цифри. Якщо так, то він перевіряє чи є формула SEND + MORE = MONEY коректною з даними числами. Якщо так, то рішення знайдено і метод повертає True. У протилежному випадку повертається False. Зрештою, якщо ще не всім буквам присвоєно цифри, то повертається True.

Це зроблено для того, щоб після отримання часткового рішення робота продовжувалася .

Спробуємо це запустити (лістинг 3.6).

Лістинг 3.6. send_more_money.py (продовження)

```
letters: List[str] = [ "S", "E", "N", "D", "M", "O", "R", "Y" ]
possible_digits : Dict [ str, List[int]] = {}
for letter in letters:
```

```
    possible_digits [letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
possible_digits [ "M" ] = [1]
    # ми не приймаємо відповіді ,які починаються з 0
csp : CSP[ str, int] = CSP(letters, possible_digits )
csp.add_constraint ( SendMoreMoneyConstraint (letters))
solution: Optional[ Dict [str, int]] = csp.backtracking_search ()
if solution is None :
    print( "No solution found!" )
else :
    print(solution)
```

Ми попередньо задали значення для літери M. Це зроблено для того, щоб виключити відповідь, в якому M відповідає 0, тому що, обмеження не враховує, що число не може починатися із нуля.

Рішення виглядає так:

{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2 }

3.2 Вдосконалений алгоритм розв'язку на основі UGA

3.2.1 Уточнення задачі

У 3.1 для вирішення задачі використовуються структура з обмеженнями. Це завдання може бути вирішене за розумний час з допомогою генетичного алгоритму набагато ефективніше.

Сформулюємо постановку задачі так, щоб її можна було би розвязати з допомогою генетичного алгоритму.

Зручне представлення для криптоарифметичних завдань - використання індексів списку у вигляді цифр. Таким чином, для подання десяти можливих цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) потрібний перелік з десяти елементів. Символи, які потрібно знайти в задачі, можна потім переміщати з місця на місце. Наприклад, якщо є підозра, що рішення завдання включає в себе символ E, представлений цифрою 4, то list[4] = "E".

У записі SEND + MORE = MONEY використовуються вісім різних літер (S, E, N, D, M, O, R, Y), так що два слота в масиві залишаться пустими. Їх можна заповнити пробілами без вказівки.

Удосконалений код для рішення SEND + MORE = MONEY - лістинг SendMoreMoney2 (Літинг 3.7, Додатки). Метод fitness( ) дещо схожий на satisfied() з SendMoreMoneyConstraint.

3.2.2 Особливості реалізації

Існує серйозна відмінність між методом satisfied( ) та fitness. Тут ми повертаємо 1 /(difference + 1). difference - це абсолютне значення різниці між MONEY та SEND + MORE. Воно показує , наскільки далека дана хромосома від рішення завдання. Якщо б ми намагалися мінімізувати fitness( ), то повернення одного лише значення difference було б достатньо. Але оскільки GeneticAlgorithm намагається максимізувати fitness( ), необхідно обчислити зворотне значення , тому ми ділимо 1 на difference.

Таблиця 3.1. Як рівняння 1/(difference + 1) дозволяє обчислити життєздатність з метою її максимізації

difference	difference + 1	fitness (1/ (difference + 1))
0	1	1,000
1	2	0,500
2	3	0,250
3	4	0,125

Чим менше різниця, тим краще і чим більша життєздатність, тим краще. Розглянута формула об'єднує ці два фактора. Розподіл 1 на значення життєздатності — простий спосіб перетворення задачі мінімізації на завдання максимізації.

У методі random_instance() використовується функція shuffle() з модуля random. Метод crossover() вибирає два випадкові індекси у списках letters обох хромосом і переставляє літери так, щоб одна з літер першої хромосоми виявилася на том ж місці в другій хромосомі і навпаки. Він виконує ці

перестановки у дочірніх хромосомах, отже розміщення букв у двох дочірніх хромосомах виявляється комбінацією батьківських хромосом. mutate() змінює дві випадкові літери у списку letters.

Альтернативний спосіб із завдання мінімізації отримати завдання максимізації - просто змінити знак . Однак це буде працювати тільки в том випадку , якщо ми маємо справу тільки із додатніми числами.

Навіть при правильному налаштуванні існує певний елемент випадковості. Завдання може бути вирішене протягом кількох секунд або кількох хвилин. На жаль, такою є природа генетичних алгоритмів.

Лістинг 3.7. send_more_money2.py (продовження)

```
if name == " main : _
   initial_population: List[SendMoreMoney2] =
[SendMoreMoney2.random_instance() for _ in range(1000)]
   ga: GeneticAlgorithm[SendMoreMoney2] =
GeneticAlgorithm(initial_population = initial_population, threshold = 1.0,
max_generations = 1000, mutation_chance = 0.2,crossover_chance = 0.7,
selection_type = GeneticAlgorithm.SelectionType.ROULETTE )
   result: SendMoreMoney2 = ga.run()
   print(result)
```

Наступні результати були отримані при запуску алгоритму, при якому завдання було вирішено за три покоління з використанням 1000 особин у кожному поколінні, створеному раніше.

Generation 0 Best 0.0040650406504065045 Avg 8.854014252391551e 05

Generation 1 Best 0.16666666666666666 Avg 0.001277329479413134

Generation 2 Best 0.5 Avg 0.014920889170684687

8324 + 913 = 9237 Difference: 0

У цьому варіанті, якщо не вказати обмеження для M (не рівне 0), то отримаємо SEND = 8324, MORE = 913 і MONEY = 9237. Фактично якщо M = 0, то існує кілька рішень завдання. Тут MORE - це 0913, а MONEY - 09237. Нуль просто ігнорується.

3.3 Деякі аспекти оптимізації

Припустимо, що у нас є якась інформація, яку ми хочемо стиснути. Припустимо, що це список предметів і їхня послідовність неважлива. Розглянемо, при який послідовності елементів ступінь стиску буде максимальної. Для більшості алгоритмів стиснення порядок елементів впливає на ступінь стиснення.

Для нашої роботи застосуємо функцію compress() з модуля zlib зі стандартними налаштуваннями. Далі показано повне рішення для списку з 12 імен (див. лістинг 3.9., Додатки). Якщо не брати генетичний алгоритм, а просто запустити compress() для 12 імен у тому порядку, в якому вони були представлені, отримаємо стислі дані обсягом 165 байт.

Ця реалізація дещо схожа на класичну реалізацію з SEND + MORE = MONEY. Функції crossover() та mutate() практично однакові. У вирішенні обох завдань ми беремо список елементів, постійно перебудовуємо його та перевіряємо перестановки. Можна написати загальний суперклас для вирішення обох завдань, який працював би з широким спектром завдань. Будь-яка завдання, якщо її можна, можливо уявити в вигляді списку елементів, для якого потрібно знайти оптимальний порядок, може бути вирішена аналогічним чином. Єдина реальна точка налаштування цих підкласів — відповідна функція життєздатності.

Робота list_compression.py може тривати дуже багато часу. Так відбувається тому, що, на відміну від двох попередніх завдань, ми не знаємо заздалегідь, що є правильною відповіддю, тому у нас немає реального порогу, до якого слід прагнути. Натомість ми встановлюємо кількість поколінь і число особин в кожному поколінні як довільне велике число і сподіваємося на краще. Стосовно мінімальної кількості байтів, яку вдасться отримати при стисканні в результаті перестановки 12 імен то у нашому найкращому випадку, використовуючи конфігурацію з попереднього рішення, після 546 поколінь

генетичний алгоритм знайшов послідовність із 12 імен, яка дала стиск розміром 159 байт.

Усього лише 6 байт у порівнянні з початковим порядком - економія становить приблизно 4 %. В даному випадку для 12 імен можна сказати, що 4 % не має значення, але якщо б це був набагато більший список, який багаторазово передавався б по мережі, то економія зросла б багаторазово. Приміром якби це був список розміром 1 Мбайт, який передавався б через Інтернет 10 000 000 разів. Якби генетичний алгоритм міг оптимізувати порядок списку для стиснення, щоб заощадити 4 %, то він зекономив б приблизно 40 Кбайт на кожну передачу і в результаті 400 Гбайт пропускної здібності для всіх передач. Це не дуже велика економія, але вона може виявитися значною настільки, що матиме сенс один раз запустити алгоритм і отримати майже оптимальний порядок стиснення.

Але проблема полягає у наступному. Насправді невідомо, чи знайшли ми оптимальну послідовність для 12 імен, не кажучи вже про гіпотетичний список розміром 1 Мбайт. Для перевірки слід перевірити ступінь стиснення для всіх можливих послідовностей в список. Але навіть для списку всього лише з 12 пунктів це важкоздійсненні операції - 479001600 можливих варіантів (12). Використання генетичного алгоритму, який намагається знайти оптимальний варіант буде більш доцільним, навіть якщо ми не знаємо, є чи його остаточне рішення справді оптимальним.

Розділ 4 Особливості програмної реалізації

4.1 Розробка веб-застосунку

4.1.1 Технологій для реалізації

Для реалізації проекту було вибрано мову програмування Python, оскільки вона ідеально підходить для роботи з блоком генетичних алгоритмів та дозволяє створювати необхідні візуалізації, а також з допомогою додаткового набору бібліотек легко можна створити веб-додаток, який буде містити всі основні функції, які нам потрібні.

Отже, набір використовуваних технологій, які дозволять нам реалізувати сервіс та його функції наступний.

По-перше, Python 3.11 – мова програмування, що добре підходить для прототипування та прискореної розробки додатків різної спрямованості, з широким набором бібліотек.

По-друге, DASH – набір з бібліотек Plot.ly та Flask; Plot.ly дозволяє швидко вибудовувати графіки для аналітики та вибудовувати карти за рахунок вбудованої підтримки JavaScript; Flask є одним з популярних і легких фреймворків для створення веб-додатків за рахунок використання Werkzeug для побудови WSGI, з вбудованим шаблонизатором Jinja2.

По-третє, Pandas – бібліотека для обробки та аналізу даних.

Також нам потрібен Visual Studio – середовище розробки та Git – розподілена система управління версіями.

4.1.2 Розгортання веб-сервера на основі Dash

Dash — це платформа з відкритим вихідним кодом для створення інтерфейсів візуалізації даних, яка допомагає фахівцям з обробки даних створювати аналітичні веб-програми, не потребуючи передових знань у галузі веб-розробки.

Створюючи додаток з Dash ми постійно процюємо з локальним сервером, який запускається на localhost.

В основі вибраного пакета Dash лежать бібліотека Flask, яка виступає в ролі фреймворку веб-сервера для обробки запитів, і бібліотека Plot.ly, яка займатиметься відображенням графіки на сторінці, спільно з відображенням html і css компонентів через Flask.

В основі роботи покладено систему callback - всі дані проходять між зворотними функціями та звичайними функціями програми. Потрібно змінити файл на інший, прочитати і відобразити його - все виконує відповідна функція.

Для встановлення необхідних компонентів виконаємо відповідні інсталяції.

pip install dash==0.31.1 # The core dash backend

pip install dash-html-components==0.13.2 # HTML components

pip install dash-core-components==0.38.1 # Supercharged components

pip install dash-table==3.1.7 # Interactive DataTable component

Програми Dash складаються із двох частин. Перша частина - "layout" описує те, як виглядає наша програма. Друга частина описує інтерактивність програми.

Dash надає Python класи для всіх візуальних компонентів програми. Розробники надають набір компонентів так званих dash_core_components і dash_html_components. Можна побудувати свій компонент використовуючи JavaScript і React.js.

У dash_core_components містяться різні динамічні форми такі як, наприклад, списки, графіки і чек-бокси, що випадають.

У dash_html_components містяться html конструкції, якими можна загорнути наші форми. Наприклад, Div блоки або теги заголовків H1, H2, і так далі. Надавати абстракцію від html можна за допомогою словників Python.

Dash містить hot-reloading. Вона активується тоді, коли запускається функція app.run_server(debug=True). Ця можливість оновлює ваш браузер щоразу, коли ви робите правки в коді і зберігаєте результат. Таким чином немає потреби щоразу перезапускати сервер.

Dash містить компонент для кожного HTML тега. Але він також може приймати всі аргументи ключових слів, як і елементи HTML.

4.1.3 Багаторазові компоненти

В роботі нам потрібні деякі елементи, які будуть змінюватися, наприклад, залежно від вхідних даних користувача нашої програми. Для цього Dash передбачені так звані reusable components. Розглянемо їх на прикладі таблиці, дані для якої завантажуватимуться з Pandas dataframe.

Лістинг 4.1.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd
df = pd.read_csv(
    'https://***.csv')
def generate_table(dataframe, max_rows=10):
    return html.Table(
        # Header
        [html.Tr([html.Th(col) for col in dataframe.columns])] +
        # Body
        [html.Tr([
            html.Td(dataframe.iloc[i][col]) for col in dataframe.columns
        ]) for i in range(min(len(dataframe), max_rows))]
    )
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
app.layout = html.Div(children=[
    html.H4(children='Data'),
    generate_table(df)
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

4.1.4 Компоненти ядра

По замовчуванню, веб-додаток буде доступний по localhost:port наприклад, http://127.0.0.1:8050/ у веб-переглядачі.

Модуль Dash Core Components (dash.dcc) надає доступ до багатьох інтерактивних компонентів. Імпортувати dash.dcc можна за допомогою from dash import dcc. Модуль dcc є частиною Dash, і знаходиться в репозиторії Dash GitHub.

У програмах Dash зазвичай використовують Dash Enterprise Design Kit для керування стилем і макетом основних компонентів Dash. Макет складається з дерева «компонентів», таких як html.Div і dcc.Graph. Dash HTML Components (dash.html) має компонент для кожного тегу HTML. Компонент html.H1(children='Hello Dash') створює HTML-елемент <h1>Hello Dash</h1> у вашій програмі.

Не всі компоненти є чистим HTML. Dash Core Components (dash.dcc) містить компоненти вищого рівня, які є інтерактивними та створюються за допомогою JavaScript, HTML і CSS через бібліотеку React.js.

Кожен компонент повністю описується через атрибути ключових слів. Dash є декларативним: в основному можна написати свою програму за допомогою цих атрибутів.

Існує поняття дочірньої властивості children. За умовами, це завжди перший атрибут, який теоретично можна опустити, тобто присвоювати значення без іменування: html.H1(children='Hello Dash') — те саме, що html.H1('Hello Dash'). Він може містити рядок, число, один компонент або список компонентів.

4.1.5 Basic Dash Callbacks

app.layout описує, як виглядає програма, і є ієрархічним деревом компонентів. Модуль Dash HTML Components (dash.html) надає класи для всіх тегів HTML, а ключові аргументи описують такі атрибути HTML, як style,

className та id. Модуль Dash Core Components (dash.dcc) створює компоненти вищого рівня, такі як елементи керування та графіки.

Дуже важливу роль в Dash відіграють Basic Dash Callbacks - функції зворотного виклику: функції, які Dash автоматично викликає щоразу, коли змінюється властивість вхідного компонента, щоб оновити деяку властивість в іншому компоненті (вихід).

Для оптимальної взаємодії з користувачем і продуктивності завантаження діаграм робочі програми Dash повинні враховувати можливості черги завдань, HPC, Datashader і можливості горизонтального масштабування Dash Enterprise.

Лістинг 4.2.

```
@app.callback(
   Output(component_id='my-output', component_property='children'),
   Input(component_id='my-input', component_property='value')
)
def update_output_div(input_value):
   return f'Output: {input_value}'
```

«Output» та «Input» нашої програми описуються як аргументи декоратора @app.callback.

У Dash вхідні та вихідні дані нашої програми є просто властивостями певного компонента. У цьому прикладі нашим входом є властивість «value» компонента, який має ідентифікатор «my-input». Наш результат — це властивість «children» компонента з ідентифікатором «my-output».

Кожного разу, коли вхідна властивість змінюється, функція, яку обертає декоратор зворотного виклику, викликається автоматично. Dash надає цій функції зворотного виклику нове значення вхідної властивості як аргумент, а Dash оновлює властивість вихідного компонента тим, що було повернуто функцією.

Ключові слова component_id і component_property є необов’язковими (для кожного з цих об’єктів є лише два аргументи).

Об’єкт dash.dependencies.Input відрізняється від об’єкта dcc.Input. Перше просто використовується в визначеннях зворотного виклику, а останнє є фактичним компонентом.

Ми не встановлюємо значення для властивості дочірніх компонента my-output у макеті. Коли програма Dash запускається, вона автоматично викликає всі зворотні виклики з початковими значеннями вхідних компонентів, щоб заповнити початковий стан вихідних компонентів. У лістингу вище, якщо ми вказали компонент div як html.Div(id='my-output', children='Hello world'), він буде перезаписаний під час запуску програми. Це називається «реактивним програмуванням», оскільки виходи автоматично реагують на зміни вхідних даних.

Кожен компонент описується повністю через набір ключових аргументів. Ті аргументи, які ми встановлюємо в Python, стають властивостями компонента, і ці властивості зараз важливі. Завдяки інтерактивності Dash ми можемо динамічно оновлювати будь-які з цих властивостей за допомогою зворотних викликів. Часто ми оновлюємо властивість дочірніх компонентів HTML для відображення нового тексту (дочірні елементи відповідають за вміст компонента) або властивість figure компонента dcc.Graph для відображення нових даних. Ми також можемо оновити стиль компонента або навіть доступні параметри компонента dcc.Dropdown.

4.2 Функціонал

Веб-додаток доступний по localhost:port. В даному випадку після запуску сервера можна отримати доступ до ресурсу по адресу http://127.0.0.1:8050/ у веб-переглядачі. Про те,що сервер запущено ми бачимо із повідомлення:

Dash is running on http://127.0.0.1:8050/

* Serving Flask app 'main'

* Debug mode: on

Головна сторінка виглядає наступним чином.

Рис. 4.1 Головна сторінка

В додатку реалізовано кілька блоків. По-перше, блок розвязування криптоарифметичних задач. Для зручності тестування додатка є можливість обрати одну з відомих задач.

Рис. 4.2.Типові завдання

Після натискання Знайти розвязок отримуємо один або декілька результатів (розвязків може бути декілька).

Рис. 4.3. Розвязок SEND + MORE == MONEY

Наступний блок – створення криптоарифметичних завдань. Логіка створення – необхідно створити такий вираз, де слова, якими зашифровано приклад мають зміст як окремо, так і разом в сукупності, утворюючи смислову загадку. І при цьому вони шифрують привильний математичний приклад.

Наприклад, NUWM + RIVNE == SUMMER

Можна складати завдання як із сумою, так і з різницею.

Рис.4.4. Створення криптоарифметичних задач

Третій блок – розв’язування довільних криптоарифметичних задач. Ми можемо розвязати щойно створену NUWM + RIVNE == SUMMER

Рис. 4.5. NUWM + RIVNE == SUMMER

Або наприклад, 1/(2*X-Y) == 1. Дозволено використовувати операції: +-*/, піднесення до степеня **.

Рис. 4.6. Ілюстрація обчислень

4.3 Практична реалізація

4.3.1 Макет програми Dash

Розглянемо Лістинг 4.3 (див. Додатки). У цьому лістингу властивості «value» dcc.Slider є вхідними даними програми, а вихідні дані програми є властивістю «figure» dcc.Graph. Щоразу, коли значення dcc.Slider змінюється, Dash викликає функцію зворотного виклику update_figure з новим значенням. Функція фільтрує фрейм даних за цим новим значенням, створює об'єкт фігури та повертає його в програму Dash.

У цьому прикладі є кілька шаблонів. Ми використовуємо бібліотеку Pandas для завантаження нашого фрейму даних на початку програми: df = pd.read_csv('...'). Цей фрейм даних df знаходиться в глобальному доступі і може бути прочитаний у функціях зворотного виклику.

Завантаження даних у пам'ять може бути ресурсомістким. Завантажуючи дані запиту на початку програми, а не всередині функцій зворотного виклику, ми гарантуємо, що ця операція виконується лише один раз — під час запуску сервера. Коли користувач відвідує програму або взаємодіє з програмою, ці дані (df) уже знаходяться в пам'яті.

Зворотний виклик не змінює вихідні дані, він лише створює копії фрейму даних шляхом фільтрації за допомогою pandas. Зворотні виклики ніколи не повинні змінювати змінні поза їхньою областю. Якщо зворотні виклики змінюють глобальний стан, то сеанс одного користувача може вплинути на сеанс наступного користувача, і коли програму розгорнуто в кількох процесах або потоках, ці зміни не будуть спільні для сеансів.

4.3.2 Деякі аспекти розвязування криптоарифметичних задач

В лістингу 4.4 (див. Додатки) вище ми враховуємо, що число, яке ми шукаємо, не може починатися з нуля, оскільки тоді вираз втрачає сенс. Ми працюємо з десятковою системою, тому цифр у нас 10.

Зручне представлення для криптоарифметичних завдань - використання індексів списку у вигляді цифр. Таким чином, для подання десяти можливих

цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) потрібний перелік з десяти елементів. Символи, які потрібно знайти в задачі, можна потім переміщати з місця на місце. Наприклад, якщо є підозра, що рішення завдання включає в себе символ E, представлений цифрою 4, то list[4] = "E".

Удосконалений код для рішення SEND + MORE = MONEY - лістинг SendMoreMoney2 (Літинг 4.5). Метод fitness( ) дещо схожий на satisfied() з SendMoreMoneyConstraint.

4.3.3 Спрощений алгоритм

Розглянемо простіший варіант – нуль може бути першим.

Навіть при правильному налаштуванні існує певний елемент випадковості. Завдання може бути вирішене протягом кількох секунд або кількох хвилин. На жаль, такою є природа генетичних алгоритмів.

Лістинг 4.6. send_more_money2.py

```
if name == " main : _
   initial_population: List[SendMoreMoney2] =
[SendMoreMoney2.random_instance() for _ in range(1000)]
   ga: GeneticAlgorithm[SendMoreMoney2] =
GeneticAlgorithm(initial_population = initial_population, threshold = 1.0,
max_generations = 1000, mutation_chance = 0.2,crossover_chance = 0.7,
selection_type = GeneticAlgorithm.SelectionType.ROULETTE )
   result: SendMoreMoney2 = ga.run()
   print(result)
```

Наступні результати були отримані при запуску алгоритму, при якому завдання було вирішено за три покоління з використанням 1000 особин у кожному поколінні, створеному раніше.

Generation 0 Best 0.0040650406504065045 Avg 8.854014252391551e 05

Generation 1 Best 0.16666666666666666 Avg 0.001277329479413134

Generation 2 Best 0.5 Avg 0.014920889170684687

8324 + 913 = 9237 Difference: 0

У цьому варіанті, якщо не вказати обмеження для M (не рівне 0), то отримаємо SEND = 8324, MORE = 913 і MONEY = 9237. Фактично якщо M = 0, то існує кілька рішень завдання. Тут MORE - це 0913, а MONEY - 09237. Нуль фактично просто ігнорується.

Висновки

Значну роль в розвитку природних систем відіграє еволюція, тому дослідники прагнуть розробляти моделі, що реально пояснюють принципи роботи цього механізму: відтворення структури популяції, народження і виродження особин, колективна поведінка та події між ними.

Генетичні алгоритми є алгоритмами пошуку, що побудовані на принципах, подібних до принципів природного відбору і генетики. Насамперед, це принцип виживання найбільш перспективних рішень, обмін інформацією, в якій наявний елемент випадковості. Разом ці принципи спроможні моделювати природні процеси успадкування і мутації. Додатковою властивістю цих алгоритмів є невтручання людини в ітераційний процес пошуку, впливати можна лише опосередковано, задаючи певні параметри. Зберігається біологічна термінологія в спрощеному вигляді і основні поняття лінійної алгебри

Досягнення оптимуму в багатокритеріальних задачах є вкрай складним та вимагає значної кількості ресурсів. Генетичні алгоритми скеровані на знаходження кращого задовільного значення, а не оптимального рішення задачі. І така властивість сприяє розвитку і зростанню популярності генетичних алгоритмів.

Список використаних джерел

1. Айерленд К., Роузен М. Класичне введення в сучасну теорію чисел. М.: Мир. 1987. 416 с.
2. Болотов А.А., Гашков С.Б., Фролов А.Б., Часовскіх А.А. Алгоритмічні основи елліптичної криптографії. М.: в-во, 2004. 499 с.
3. Брассар Ж. Сучасна криптологія. М., ПОЛІМЕД, 1999. – 176 с..
4. Лідл Р., Нідеррайтер Г. Кінцеві поля (том 1, 2).1988.
5. Бабаш А.В., Баранова Е.К. Криптографічні методи захисту інформації (для бакалаврів та магистрів) . М., 2015. 224 с.
6. Бабаш А.В. Криптографічні методи захисту інформ.: Навчальний посібник: Т.1 . М.: Ріор, 2018. 480 с.
7. Баранова, Е.К. Криптографічні методи захисту информації. Лабораторний практикум (для бакалаврів) . М., 2018. 288 с.
8. Запечніков С.В., Казарін О.В., Тарасов А.А. Криптографічні методи захисту информації: Навчальний посібник. Л.: Юрайт, 2016. -309 с.
9. Зубов А.Ю. Криптографічні методи захисту информації. Досконалі шифри . М.: Геліос АРВ, 2005. 192 с.
10. Рябко Б.Я., Фіонов А.Н. Криптографічні методи захисту информації. М.: ГЛТ, 2013. 229 с.
11. Жельников В. Поява шифрів . Криптографія від папіруса до комп'ютера. М.: ABF, 1996. 335 с. ISBN 5-87484-054-0
12. Шнайер Б. Прикладна криптографія. Протоколи, алгоритми, вихідні тексти на мові Ci = Applied Cryptography. Protocols, Algorithms and Source Code in C. M.: Тріумф, 2002. 816 с. ISBN 5-89392-055-4
13. Піліді В. С. Криптографія. Вступні розділи. ЮФУ, 2009. 110 с.
14. David Kahn, Remarks on the 50th Anniversary of the National Security Agency, November 1, 2002.
15. Набебін А.А. Модулярна арифметика и криптографія. М.: MEI, 2007. 201с.

16. Петров А.А. Комп'ютерна безпека. Криптографічні методи захисту. М.: ДМК, 2000. –448 с.
17. Романець Ю.В., Тимофєєв П.А., Шаньгін В.Ф. Захист інформації в комп'ютерных системах и мережах– 2-е вид., перероб. і доп. М.: Радіо і зв'язок, 2001. 376 с.
18. Смарт Н. Криптографія. М.: Техносфера, 2005. 528 с.
19. Фергюсон Нільсон, Шнайер Брюс. Практична криптографія.
20. Фомічев В.М. дискретна математика и криптографія. Курс лекцій / під заг. ред. Д-ра фіз.-мат. Н.Д. Подуфалова., 2003 . 400с.
21. Menezes A., van Oorshot P., Vanstone S. Handbook of applied cryptography, 1997.
22. Лапоніна О.Р. Основи мережевої безпеки: криптографічні алгоритми і протоколи їхньої взаємодії. М.: Інтернет-Ун-т Інформ. Технологій, 2007. 531с.
23. Черемушкін А.В. Криптографічні протоколи. Основні властивості і вразливості. М.: Вид. центр "Академія", 2009. 272 с.
24. Аграновський А.В., Хаді Р.А. Практична криптографія: алгоритмі і їх програмування. М.: СОЛОН-Прес, 2009. 256 с.
25. Оліфер В.Г., Оліфер Н.А. Мережеві операційні системи., 2009, 668 с.

Схожість

Джерела з Бібліотеки 228

1	Студентська робота	ID файлу: 12018888	Навчальний заклад: Lviv Polytechnic National University		1.65%
2	Студентська робота	ID файлу: 1005759464	Навчальний заклад: National Aviation University	24 Джерело	1.3%
3	Студентська робота	ID файлу: 1005789071	Навчальний заклад: Poltava National Technical Yuri Kond	3 Джерело	1.06%
4	Студентська робота	ID файлу: 1005732552	Навчальний заклад: Lviv Polytechnic National University	4 Джерело	0.91%
5	Студентська робота	ID файлу: 1000086621	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.75%
6	Студентська робота	ID файлу: 1000676314	Навчальний заклад: National University of Life and Environmenta...		0.74%
7	Студентська робота	ID файлу: 2055021	Навчальний заклад: Lviv Polytechnic National University		0.67%
8	Студентська робота	ID файлу: 5757147	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.65%
9	Студентська робота	ID файлу: 8571118	Навчальний заклад: Yuriy Fedkovych Chernivtsi National University		0.5%
10	Студентська робота	ID файлу: 2053879	Навчальний заклад: Lviv Polytechnic National University	2 Джерело	0.44%
11	Студентська робота	ID файлу: 8283845	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.34%
12	Студентська робота	ID файлу: 1000062917	Навчальний заклад: National Technical University of Ukr	17 Джерело	0.31%
13	Студентська робота	ID файлу: 1003054104	Навчальний заклад: National Technical University of Ukra	6 Джерело	0.3%
14	Студентська робота	ID файлу: 1011568603	Навчальний заклад: Vasyl Stus Donetsk National University		0.3%
15	Студентська робота	ID файлу: 1003856443	Навчальний заклад: National Technical University of Ukra	2 Джерело	0.29%
16	Студентська робота	ID файлу: 1013096738	Навчальний заклад: National University of Water Manag	30 Джерело	0.28%
17	Студентська робота	ID файлу: 113670	Навчальний заклад: Lviv Polytechnic National University		0.26%
18	Студентська робота	ID файлу: 1009654478	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.24%
19	Студентська робота	ID файлу: 1004025086	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.23%
20	Студентська робота	ID файлу: 6054524	Навчальний заклад: National University of Water Management and N...		0.22%

21	Студентська робота	ID файлу: 1000077739	Навчальний заклад: Lviv Polytechnic National University		0.22%
22	Студентська робота	ID файлу: 1056604	Навчальний заклад: Lviv Polytechnic National University		0.21%
23	Студентська робота	ID файлу: 4831675	Навчальний заклад: Yuriy Fedkovych Chernivtsi National Uni	4 Джерело	0.2%
24	Студентська робота	ID файлу: 1004149963	Навчальний заклад: Donetsk National Technical University		0.18%
25	Студентська робота	ID файлу: 1000098785	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.18%
26	Студентська робота	ID файлу: 1013060173	Навчальний заклад: Zaporizhzhya National University		0.18%
27	Студентська робота	ID файлу: 1000018318	Навчальний заклад: Vinnytsia State Pedagogical Univers	51 Джерело	0.17%
28	Студентська робота	ID файлу: 1008363182	Навчальний заклад: National Aviation University	2 Джерело	0.17%
29	Студентська робота	ID файлу: 1011573146	Навчальний заклад: Vasyl Stus Donetsk National University		0.17%
30	Студентська робота	ID файлу: 5968053	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.16%
31	Студентська робота	ID файлу: 1006780625	Навчальний заклад: Donetsk National Technical Universit	7 Джерело	0.13%
32	Студентська робота	ID файлу: 1011573525	Навчальний заклад: Vasyl Stus Donetsk National University		0.13%
33	Студентська робота	ID файлу: 5545068	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.12%
34	Студентська робота	ID файлу: 1011424717	Навчальний заклад: National Aviation University	3 Джерело	0.11%
35	Студентська робота	ID файлу: 8319142	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.11%
36	Студентська робота	ID файлу: 5972336	Навчальний заклад: National Technical University of Ukraine	9 Джерело	0.11%
37	Студентська робота	ID файлу: 1004168745	Навчальний заклад: National University of Life and Envir	25 Джерело	0.1%
38	Студентська робота	ID файлу: 1009689720	Навчальний заклад: National Aviation University		0.1%
39	Студентська робота	ID файлу: 1015115962	Навчальний заклад: Vasyl Stus Donetsk National University		0.09%
40	Студентська робота	ID файлу: 1011573175	Навчальний заклад: Vasyl Stus Donetsk National University		0.07%
41	Студентська робота	ID файлу: 5968965	Навчальний заклад: National Technical University of Ukraine "Kyiv Po...		0.07%
42	Студентська робота	ID файлу: 1000973486	Навчальний заклад: National Aviation University		0.07%

43	Студентська робота	ID файлу: 1000766237	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.06%
44	Студентська робота	ID файлу: 1001041635	Навчальний заклад: National Aviation University	3 Джерело	0.06%
45	Студентська робота	ID файлу: 1011398770	Навчальний заклад: National Aviation University		0.05%
46	Студентська робота	ID файлу: 1011522344	Навчальний заклад: National University of Water Manage	4 Джерело	0.05%
47	Студентська робота	ID файлу: 1003867931	Навчальний заклад: Donetsk National Technical University		0.05%
48	Студентська робота	ID файлу: 1014693898	Навчальний заклад: National Technical University of Ukra	2 Джерело	0.05%

Цитати

Цитати **3**

1 2.2.2 **Особливості задання параметрів Усі параметри для алгоритму, а саме скільки хромосом повинно бути в популяції, який поріг зупиняє алгоритм, як вибирати хромосоми для розмножения, як вони мають схрещуватися і з якою ймовірністю, з якою ймовірністю мають відбуватися мутації, скільки поколінь потрібно створити будуть налаштовуватись у класі** GeneticAlgorithm (**Листинг** 5.2).

2 Fluent Python (O'Reilly, 2015)].

3 "11*0****1*00". **Тобто за допомогою схем можна виділяти загальні ділянки двійкових рядків і маскувати відмінності. Маючи у складі схем** m **символів** "