КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра системного аналізу та теорії прийняття рішень

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 124 Системний аналіз

на тему:

МОДЕЛІ ТА МЕТОДИ РОЗВ'ЯЗАННЯ ЗАДАЧІ ПОШУКУ ОПТИМАЛЬНОГО МАРШРУТУ ТА ЗАДАЧІ КОМІВОЯЖЕРА ЗА УМОВ ЗМІНИ СТАНУ СИТУАЦІЇ

Виконала студентка 4-го курсу
Нагорна Владлена Вадимівна

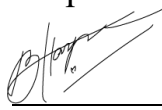
(підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Івохін Євген Вікторович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент



(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри системного аналізу та теорії прийняття рішень
«01» червня 2023 р.,
протокол № 13
Завідувач кафедри
О. Г. Наконечний

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 72 сторінки, 13 ілюстрацій, 1 таблиця, 11 джерел посилань.

ДИНАМІЧНА ЗАДАЧА КОМІВОЯЖЕРА, ДИНАМІЧНА АДАПТАЦІЯ АЛГОРИТМІВ ПОШУКУ ОПТИМАЛЬНОГО ШЛЯХУ, МОДЕЛЮВАННЯ ПОШИРЕННЯ НЕБЕЗПЕКИ. АЛГОРИТМИ ПОШУКУ В ДИНАМІЧНОМУ ГРАФІ.

Об’єктом роботи є динамічна адаптація алгоритмів вирішення динамічної задачі комівояжера.

Метою роботи є розробка методу симуляції відпалу для вирішення задачі комівояжера та моделювання надзвичайної ситуації.

Результати роботи: Виконано детальний огляд адаптацій деяких алгоритмів для вирішення проблеми динамічної задачі комівояжера, розроблено програму, що візуалізує пошук оптимального маршруту методом симуляції відпалу в режимі реального часу. Також розроблена модель простого клітинного автомату, що імітує розповсюдження пожежі, тобто надзвичайної ситуації.

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

TSP – задача комівояжера;

DTSP – динамічна адаптація задачі комівояжера;

SA – метод імітації відпалу;

GA – генетичний алгоритм;

ACO – алгоритм побудови мурашиної колонії;

КА – клітинний автомат;

ACO - SA – гібридний метод поєднання двох алгоритмів для вирішення динамічної адаптації задачі комівояжера;

ЗМІСТ

ВСТУП

Динамічна задача комівояжера - одна з найважливіших і найскладніших задач в області оптимізації маршрутів. Вона виникає в багатьох реальних ситуаціях, коли потрібно знайти оптимальний маршрут для відвідування низки міст з найменшими витратами часу та коштів. Однак багато реальних проблем туристичних агентств є динамічними, оскільки умови можуть змінюватися з часом.

Мета полягає в вивчені різних алгоритмів та розробці методу імітаційного моделювання відпалу для забезпечення того, щоб алгоритм, який розв'язує динамічну задачу комівояжера, міг адаптуватися до змін умов і знаходити оптимальний маршрут. Крім того, дослідження моделює надзвичайні ситуації, зокрема поширення пожежі, щоб проаналізувати вплив небезпек на вибір оптимального маршруту.

Результати включають детальне дослідження адаптації декількох алгоритмів для вирішення динамічної задачі комівояжера та розробку програми для візуалізації пошуку оптимального маршруту шляхом імітації відпалу в реальному часі. Також була розроблена модель простого клітинного автомата для імітації розповсюдження пожежі - типового прикладу надзвичайної ситуації.

Робота є важливим внеском у дослідження динамічних задач комівояжера та адаптації алгоритмів пошуку оптимального шляху до змінних умов. Вона є основою для подальших досліджень у цій галузі і може бути використана для розробки ефективних рішень реальних задач маршрутизації та аварійних ситуацій.

РОЗДІЛ 1. МАТЕМАТИЧНІ ЗАДАЧІ ТА АЛГОРИТМИ ФОРМУВАННЯ ШЛЯХІВ ПЕРЕСУВАННЯ

Аналіз існуючих підходів для розв’язання задач побудови шляхів виходу

Алгоритми пошуку шляху є дуже важливим і широко застосовуваним рішенням в інформатиці та інших галузях, таких як логістика, транспорт, геологія, біологія та інші. Останніми роками розвиток алгоритмів пошуку шляху значно прискорився завдяки новим методам та технологіям. Наприклад, алгоритми на основі штучного інтелекту, нейронних мереж, генетичних алгоритмів та інших підходів стають все більш популярними та ефективними.

Зараз існує безліч алгоритмів пошуку шляху, включаючи алгоритми на основі графів, дерев, штучного інтелекту, еволюційних та генетичних алгоритмів, алгоритми на основі нейронних мереж та багато інших. Кожен з цих алгоритмів має свої переваги та недоліки та може бути застосований для розв'язання різних задач.

Для вирішення задачі побудови шляхів виходу можна використовувати різноманітні архітектури нейронних мереж, такі як зворотній поширення, віджимання, довготривала короткочасна пам'ять та інші. Дані архітектури можуть бути використані для класифікації об'єктів, прогнозування результатів та побудови шляхів виходу.

Застосування штучних нейронних мереж до задачі побудови шляхів виходу зазвичай потребує великої кількості вхідних даних, таких як картографічні дані та дані про перешкоди. Ці дані можуть бути використані для тренування нейронної мережі, яка вирішує задачу побудови шляхів виходу.

Один з прикладів застосування штучних нейронних мереж до задачі побудови шляхів виходу - це метод Deep-Q Learning. У цьому методі нейронна мережа навчається вирішувати задачу побудови шляхів виходу,

користуючись великою кількістю симуляцій та підсиленням. Метод Deep-Q Learning може бути застосований до широкого спектру задач побудови шляхів виходу, включаючи задачі в великих масштабах та з високою складністю.

Однак, незважаючи на швидкий розвиток штучного інтелекту та нейронних мереж, алгоритми на графах все ще залишаються однією з найбільш популярних та важливих галузей теорії алгоритмів. Це пов'язано з тим, що графи можуть моделювати складні системи зі значною кількістю залежностей, наприклад, мережі зв'язків, соціальні мережі, транспортні системи, системи логістики, а також біологічні та хімічні реакції.

Окрім того, алгоритми на графах мають широкий спектр застосувань, таких як знаходження шляхів, потоків, максимального паросполучення, мінімального кістякового дерева, знайдення клік, максимального незалежного множини та багатьох інших. Важливими також є алгоритми пошуку кількох шляхів, транзитивного замикання, та алгоритми знаходження вершин або ребер, які є критичними для графа. Більше того, алгоритми на графах допомагають розв'язувати складні задачі планування та прийняття рішень, що робить їх важливим інструментом для бізнесу та інших галузей.

Одним з підходів для розв'язання задачі побудови шляхів виходу є алгоритми пошуку в глибину (DFS) та пошуку в ширину (BFS). Ці алгоритми використовуються для знаходження шляху від стартової точки до кінцевої точки в графі. Якщо такий шлях існує, то ці алгоритми знайдуть його. Але, якщо граф має велику кількість вершин та ребер, ці алгоритми можуть стати недостатньо ефективними.

Ще одним підходом є алгоритм A* (A-star). Він використовує евристичну функцію, щоб визначити, який шлях може бути оптимальним для з'єднання стартової та кінцевої точок [7]. Алгоритм A* знаходить найкоротший шлях в графі, розглядаючи не тільки відстані між вершинами, але і припустимі оцінки відстаней до кінцевої точки. Цей підхід може бути ефективним для великих графів, але потребує використання евристичної

функції, яка може бути неправильною та привести до неправильного розв'язання задачі.

Інший підхід - генетичний алгоритм, який може використовуватися для побудови шляхів в графі. У цьому підході створюється популяція можливих шляхів, і застосовується процес мутації та схрещування для того, щоб отримати нові шляхи. Відбір кращих шляхів здійснюється на основі функції оцінювання, яка визначає які шляхи є найбільш оптимальними

Алгоритми пошуку шляху можуть бути використані для розв'язання різноманітних задач, таких як побудова маршрутів транспорту, пошук найближчих маршрутів для доставки товарів, навігація в мобільних додатках, пошук оптимального маршруту для роботів та багато інших. Важливість алгоритмів пошуку шляху полягає в тому, що вони дозволяють ефективно знаходити оптимальні рішення в складних задачах з великою кількістю можливих варіантів.

Пошук шляху в динамічному графі за допомогою алгоритму A*

Граф називається динамічним, коли його вершини, ребра або значення змінюються з плином часу. До динамічної форми графів відносять соціальні мережі, мережі зв’язку, структури фондових ринків тощо.

Динамічний граф, як модель динамічного зв’язку, являє собою послідовність «класичних» графів, не маючих паралельних ребер, перехід між якими описується різними операціями на графах (виключення або включення ребра, виключення або включення вершини і т.д.). Звичайні зміни, що залежать від часу, полягають у крайових вагах, які також можуть моделювати крайові з'єднання / роз'єднання.

Задача пошуку найкоротшого шляху є однією з найбільш вивчених комбінаторних задач оптимізації, але динамічних графам приділялося набагато менше уваги в літературі, а ніж класичним [7].

Динамічні графи можна класифікувати за видами операцій, які з ними можна проводити. Динамічні графіки бувають повністю або частково динамічними. У повністю динамічних графах можна використовувати всі типи операцій.

До таких операцій входять:

• Включення вершини $x, x \notin V$.

• Виключення вершини $x, x \in V$.

• Включення ребра (x, y, w), де $x, y \in V$, $(x, y) \notin E$, $x \neq y$.

• Виключення ребра (x, y), де $x, y \in V$, $(x, y) \in E$.

• Збільшення ваги (x, y, w), де $x, y \in V$, $(x, y) \in E$, $w > W(x, y)$.

• Зменшення ваги (x, y, w), де $x, y \in V$, $(x, y) \in E$, $w < W(x, y)$.

У частково динамічних графах можна тільки включення ребер та зменшення значення їх ваг, або виключення ребер та збільшення ваг.

Для задачі пошуку найкоротшого шляху включення вершини x у граф не матиме ніякого ефекту, оскільки x не межує з жодною вершиною. Для операції виключення ми припускаємо, що видалення вершини x інтерпретується як видалення цієї вершини з графом разом з усіма ребрами до неї та від неї. Виключення ребра можна описати як збільшення ваги до нескінченності. Включення ребра матиме ефект лише коли між вершинами, що поєднуються, не було іншого ребра, або ж вага цього ребра менша за вагу існуючого. Перераховувати це кожен раз не дуже швидко та неефективно, тому було створено багато модифікацій вже існуючих алгоритмів, які зможуть швидко маніпулювати та обробляти змінні.

Метод вирішення повністю динамічної задачі заснований на використанні двох операцій: виключення та включення ребра, за допомогою яких враховуються усі можливі модифікації графа. Практична цінність методу полягає в тому, що, в порівнянні з відомими швидкими алгоритмами, цей метод дозволяє вирішувати задачу актуалізації відстаней після видалення ребра на розріджених графах швидше і при цьому не використовуючи жодної додаткової інформації.

- Включення ребра

Розглядається додавання ребра $e(a_0, b_0)$ ваги $w(a_0, b_0)$ між вершинами a_0 та b_0. Необхідно перерахувати відстані $l(a_i, b_j)$, які в результаті додавання даного ребра зменшується. Передбачається, що вага ребра, що додається менше поточної відстані між вершинами

$w(a_0, b_0) < l(a_0, b_0)$, так як інакше перерахунок відстаней робити не потрібно.

Ребро, що додається $e(a_0, b_0)$ можна представляти як «міст», що з'єднує безліч вершин $A = \{a_i\}$ і $B = \{b_j\}$, які розбивають безліч вершин графа V на вершини, які ближче до $a0$, і вершини, які ближче до b_0:

$$A = \{v \in V \mid l(v, a_0) < l(v, b_0)\}, \quad (1)$$

$$B = \{v \in V \mid l(v, b_0) < l(v, a_0)\}. \quad (2)$$

Припустимо, що вершина a_i у напрямку від кореня дерева пов'язана ребрами з вершин a_i+1, a_i+2, . . . , a_i+k. Також припустимо, що в даний момент ми перераховуємо відстань між a_i і деякою вершиною b_j. Відстань між a_i та b_j необхідно перерахувати, якщо довжина шляху через ребро, що включається $e(a_0, b_0)$ буде менше поточної відстані між a_i та b_j , тобто якщо виконується умова

$$l(a_i, a_0) + w(a_0, b_0) + l(b_0, b_j) < l(a_i, b_j). \quad (3)$$

Якщо нерівність не виконується, то величина $l(a_i, b_j)$ не перераховується, тому що існує шлях між a_i і b_j , що не проходить через $e(a_0, b_0)$ і має не більшу довжину. Але в цьому випадку не треба також перераховувати відстані між b_j та вершинами a_i+1, a_i+2, . . . , a_i+k.

На підставі цього можна зменшити розміри A і B, виключаючи вершини в A, яким не слід перераховувати відстані до b_0 та вершини в

B, яким не слід перераховувати відстані до a_0. Формально в A та B включаються

відповідно, вершини a_iта b_j , що задовольняють умовам

$$l(a_i\ , a_0) + w(a_0, b_0) < l(a_i\ , b_0), \quad (4)$$

$$l(b_j\ , b_0) + w(a_0, b_0) < l(b_j, a_0). \quad (5)$$

- Виключення ребра

Розглядається виключення ребра $e(a_0, b_0)$ ваги $w(a_0, b_0)$між вершинами a_0 та b_0. Необхідно перерахувати відстані $l(a_i, b_j)$, значення кожної, в результаті виключення ребра, збільшиться. Передбачається, що вага ребра, що виключається, дорівнює поточній відстані між вершинами $w(a_0, b_0) = l(a_0, b_0)$, тому що в іншому випадку перерахунок відстаней робити не потрібно.

Аналогічно до попередньої операції можна зменшити A та B.

$$l(a_i\ , a_0) + w(a_0, b_0) = l(a_i\ , b_0), \quad (6)$$

$$l(b_j\ , b_0) + w(a_0, b_0) = l(b_j\ , a_0). \quad (7)$$

Відмінністю процедури виключення ребра від випадку включення ребра є відсутність інформації про довжину найкоротшого альтернативного шляху.

На кожному з альтернативних шляхів між вершинами a_0 і b_0 існує точка, від якої відстань до 0 і до 0 однакова. Використовуючи такі точки, можна позначити всі можливі варіанти альтернативних шляхів між a_0 і b_0, ставлячи у відповідність кожному варіанті відмінну рівновіддалену точку. Набір цих точок також визначить і всі можливі варіанти альтернативних шляхів, що не проходять через ребро $e(a_0, b_0)$, між усіма парами вершин в A та B.

Так як рівновіддалені точки мають однакову відстань до вершин, інцидентних віддаленому ребру, то при видаленні ребра $e(a_0, b_0)$ відстані від рівновіддалених точок до всіх інших вершин графа не зміниться. Таким чином, нова відстань між вершинами a_i та b_j в A та B

може бути визначена як мінімальна сума відстаней через рівновіддалені точки:

$$l(a_i, b_j) = \min k \left(l(a_i, c_k) + l(c_k, b_j) \right). \quad (8)$$

Завдяки динамічним графам можна розвити програму до рівня, коли вона буде шукати шлях в режимі «реального часу». Також, додаючи ефективні моделі, завдяки клітинним автоматам, поширення небезпеки різного роду, використання алгоритму можливе для розроблення системи допомоги для евакуації з використанням сучасних технологій та динамічних світлових вказівників.

Клітинні автомати – це дискретні динамічні системи, поведінка яких повністю визначається в термінах локальних взаємозв'язків. Область застосування моделей такої системи безмежна: від найпростіших «хрестиків нуликів» до штучного інтелекту. Інтерес до предмета розширився за межі академічної науки, що зумовлено підвищенням обчислювальної потужності комп'ютера і доступності. Найбільш дослідженими є двовимірні клітинні автомати, наприклад гра «Життя» та «Мураха Ленгтона».

Клітинний автомат - дискретна модель. Включає регулярну решітку осередків, кожен з яких може перебувати в одному з кінцевого безлічі станів, таких як 1 і 0. Решітка може бути будь-якої розмірності. Для кожного осередку визначено безліч осередків, званих околицею. Наприклад, околиця може бути визначена як всі осередки на відстані не більше 2 від поточної.

Задача комівояжера

Комівояжер – мандрівний торговець, що відвідує різні міста на своєму шляху. Його ціль полягає в тому, щоб пройти кожне місто зі свого плану по одному разу, та повернутись до населеного пункту, який він покинув

першим. Задача комівояжера (TSP) полягає в пошуку найвигіднішого маршруту для пересування між містами за поліноміальний час.

Задача в сучасному вигляді була сформульована ще в 1934 році, та в області оптимізації дискретних задач все ще відіграє важливу роль. Також цю проблему часто використовують для перевірки та покращення нових методів вирішення оптимізаційних задач.

Щоб привести задачу до її загального вигляду позначимо населені пункти плану комівояжера числами $T = (1, 2, 3, \dots, n)$. Маршрут торгівця може бути описаний як $t = (j_1, j_2, \dots, j_n, j_1),\ j_i \in T$. Як ми можемо бачити, маршрут є зацикленим, про що характеризує j_1 на початку та в кінці списку.

Задача комівояжера є алгоритмічно складною, тому що при кількості вершин після 10 кількість маршрутів збільшується експоненціально, наприклад кількість маршрутів для 15 міст дорівнює 15! = 1,307,674,368,000

В симетричній задачі комівояжера зустрічається $(n-1)!/2$ оптимальних маршрутів, в свою чергу для асиметричної – $(n-1)!$, це спричинено тим, що в симетричній задачі комівояжера цільова функція між сусідніми вершинами рівна, тому немає необхідно рахувати по два рази перехід від точки А до точки

Асиметрична та симетрична задачі комівояжера

Для симетричної задачі комівояжера (Symmetric Traveling Salesman Problem) матриця відстаней між точками задана симетрично, тобто відстань між точкою i та точкою j дорівнює відстані між точкою j та точкою i. Також, в цьому випадку кожний маршрут можна проїхати у будь-якому напрямку, оскільки відстань між двома точками буде однаковою незалежно від напрямку руху.

Для асиметричної задачі комівояжера (Asymmetric Traveling Salesman Problem) матриця відстаней між точками задана асиметрично, тобто відстань між точкою i та точкою j може відрізнятись від відстані між точкою j та

точкою i. У цьому випадку кожний маршрут має певний напрямок і може бути проїханий лише в одному напрямку.

Формально, задача полягає в тому, щоб знайти гамільтонів цикл найменшої ваги в орієнтованому графі, де вага кожної дуги відповідає вартості подорожі між двома вершинами. Для обох випадків задачі комівояжера можуть бути застосовані різні методи і алгоритми для пошуку оптимального маршруту. Однак, враховуючи відмінності між симетричною та асиметричною задачами, необхідно звертати увагу на відповідні властивості та особливості кожної з них при виборі методу та розробці алгоритму для вирішення задачі.

Задача комівояжера у разі використання реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості маршрутів, а також від напрямку руху.

Приклад неорієнтованого графа симетричної задачі комівояжера та орієнтованого графа асиметричної задачі комівояжера:

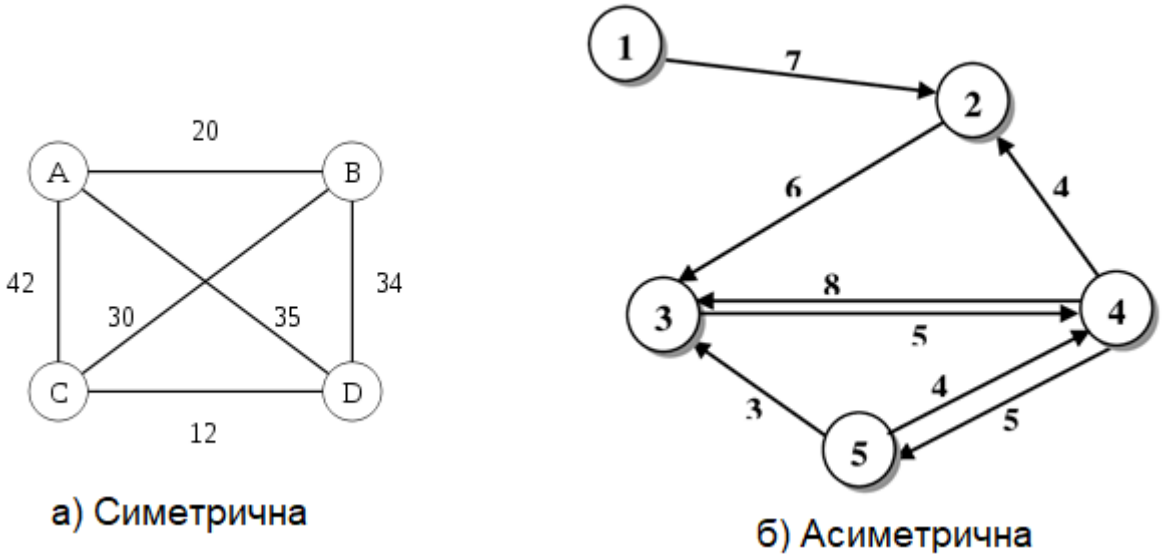


Рисунок 1 – Приклад симетричної та асиметричної задачі комівояжера

Класи складності

При вирішенні дуже великої кількості природних задач оптимізації математики часто стикалися з тим, що ці задачі були певною мірою важкорозв'язні. Важко вирішувані в тому сенсі, що ефективних, які мають поліноміальну складність алгоритмів для них досі не знайдено, і є вагомі докази, що дозволяють припустити, що для цих задач ефективних алгоритмів рішення немає. Принциповим результатом дослідження таких задач стали поняття класів P і NP.

Клас P (polynomial) – клас завдань, які можна вирішити детермінованими алгоритмами із поліноміальним часом роботи.

Клас NP (non-deterministic polynomial) – клас завдань, які можна вирішити за поліноміальний час недетермінованими алгоритмами

Пояснимо відмінність детермінованих та недетермінованих алгоритмів.

Припустимо, алгоритм доходить до місця, в якому має бути зроблений вибір з кількох альтернатив. Детермінований досліджував би одну альтернативу і потім повертався б для дослідження інших, а недетермінований може досліджувати всі можливості одночасно, "копіюючи", по суті, самого себе для кожної альтернативи. Усі копії недетермінованого алгоритму працюють незалежно і можуть, у свою чергу, створювати подальші копії. Якщо копія виявляє, що вона зробила неправильний вибір, вона припиняє виконуватись, а якщо вона знаходить рішення, то оголошує про свій успіх, і всі копії припиняють працювати.

Очевидно, що клас P є підмножиною класу NP, тобто PNP. Важливою проблемою є збіг цих класів. Введемо поняття NP-повної та NP-складної задач: Задача P1 перетворюється на задачу P2 , якщо будь-який окремий випадок задачі P1 можна перетворити за поліноміальний час у деякий окремий випадок задачі P2, так що розв'язання задачі P1 можна отримати за поліноміальний час вирішення цього окремого випадку задачі P2.

Задача є NP -складною, якщо кожна задача з NP перетворюється на неї, і задача є NP -повною, якщо вона одночасно є NP - складною та входить до NP.

Клас NP -повних задач має наступні цікаві властивості, що випливають із визначення:

- жодна NP-повна задача не може бути розв’язаною жодним відомим поліноміальним алгоритмом;
- якщо існує поліноміальний алгоритм для якоїсь NP - повної задачі, тобто поліноміальні алгоритми для всіх NP -повних завдань.

Алгоритмічний час задачі комівояжера тісно пов'язаний з задачею повного перебору або задачі рівності класів P та NP. Опис цієї проблеми можна подати наступним чином: якщо відповідь на задачу можна знайти за прийнятний час (поліноміальний), то чи можливо, використовуючи поліноміальну пам'ять також знайти відповідь на це питання, тобто чи є можливість перевірити сертифікат (певне значення, яке використовується для перевірки) за такий же час, що і знайти його.

Окрім вже названих класів існують також і інші класи складності, наприклад:

- Клас BPP (*bounded-error, probabilistic, polynomial*) — називається клас предикатів *P(x)*, обчислюваних на імовірнісних машинах Тюрінга за поліноміальний час з помилкою не більше $1/3$.
- Клас мов L — множина мов, розв'язних на детермінованій машині Тюрінга з використанням $O(\log(n))$ (Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину) додаткової пам'яті для входу довжиною n.
- Клас мов NL — множина мов, розв'язних на недетермінованій машині Тюрінга з використанням $O(\log(n))$ додаткової пам'яті для входу довжиною n.

- Клас NC. Задача належить до цього класу, якщо існують константи c і k такі, що її можна розв'язати за час $O(log^c n)$ при використанні $O(n^k)$ паралельних процесорів. Стівен Кук назвав його «класом Ніка» на честь Ніка Піппенжера, який широко дослідив схеми з полілогарифмічною глибиною і поліноміальним розміром.

Наведені класи складності вважаються легкими, так само як клас P, . До класів, що припускаються складними належить клас NP та ще декілька класів, наприклад:

- PSPACE – формально, це множина всіх мов, які можуть бути вирішені за поліноміальний час на детермінованій машині Тюрінга з поліноміально обмеженою пам'яттю $O(n^k)$. Найважчі задачі класу PSPACE — PSPACE-повні задачі.

  Також існують класи, які вважаються складними, наприклад:

- EXPTIME (*Exponential Time*) — клас задач, які розв'язні на машині Тюрінга за час $O(2^{p(n)})$, де p(n) — поліноміальна функція від n.

$$EXPTIME = \bigcup_{k=1}^{\infty} O\left(2^{n^k}\right), \tag{9}$$

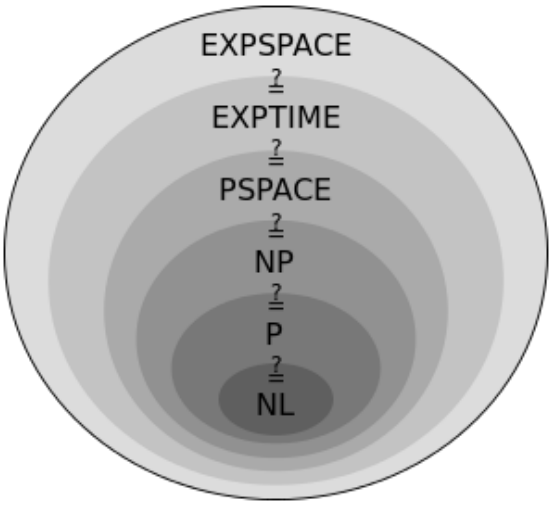


Рисунок 2 - Відношення вкладення між класами складності

Жадібні алгоритми

Жадібні алгоритми (Greedy algorithms) - це ціле сімейство алгоритмів (інколи ще говорять про жадібний підхід або жадібне програмування), тому немає якогось конкретного жадібного алгоритму, який можна закодувати. Проте, всі жадібні алгоритми побудовані згідно одного принципу - обирати оптимальне вирішення на кожному кроці, не зважаючи на кроки, які були зроблені до або будуть зроблені після. Іншими словами, жадібний алгоритм робить локально оптимальний вибір у сподіванні, що він приведе до глобального оптимального розв'язку. Характерною особливістю даної групи методів є час пошуку розв’язку, а недоліком - те, що розв’язок часто не є оптимальним.

Потрібно відмітити, що не існує можливості оцінити застосовність жадібних алгоритмів у вирішенні конкретної прикладної задачі, однак для задач, що вирішуються такими методами, виділяють дві особливості:

- До них можна застосувати так званий «Принцип жадібного вибору»;
- Для них характерна властивість «Оптимальності для підзадач».

До задачі можна застосувати принцип жадібного вибору у тому випадку, коли послідовність локальних оптимумів прямує до глобального оптимального розв’язку.

Приклад функції, що реалізує жадібний алгоритм з застосуванням принципу жадібного вибору для вирішення задачі комівояжера:

```
def greedy_nearest_neighbor(start_city, distance_matrix):
    current_city = start_city
    unvisited_cities = set(range(len(distance_matrix))) - {start_city}
    tour = [start_city]
    while unvisited_cities:
        nearest_city = min(unvisited_cities, key=lambda city:
distance_matrix[current_city][city])
        tour.append(nearest_city)
        unvisited_cities.remove(nearest_city)
```

```
    current_city = nearest_city
return tour
```

Мурашиний алгоритм

Мурашиний алгоритм (Ant colony optimization, ACO) - відносно молодий алгоритм. Перша версія алгоритму, запропонована Марко Доріго у 1992 році, була спрямована на пошук оптимального шляху у графі. Сутність алгоритму полягає в аналізі й використанні моделі поведінки мурах, які шукають шлях від колонії до джерела харчування, та представляє собою мета евристичну оптимізацію.

Якщо змоделювати переміщення мурах на деякому графі, ребра якого являють собою можливі шляхи переміщення мурах, то крім позитивного зворотного зв'язку (відкладання феромону), існує також негативний зворотній зв'язок, який проявляється в вигляді такого явища як випаровуваності феромону. Присутність негативного зворотного зв'язку гарантує, що мурахи будуть шукати інші шляхи від колонії до джерела їжі, тобто процес знаходження локального оптимального рішення не є єдиним. Рішенням задачі буде найбільший вміст феромону на ребрах графа, який є оптимальним.

- Розміщуємо мурах в вершинах заданого графу (містах);
- Починаємо рух мурах.

В якості напряму руху мурах використовуємо ймовірність, яку розраховуємо за формулою:

де:

$$P_i = \frac{l_i^q * f_i^p}{\sum_{k=0}^{N} l_k^q * f_k^p}, \quad (10)$$

P_i – ймовірність переходу за шляхом i,

l_i - величина, обернена до ваги i -го переходу,

f_i - кількість феромону на i –мут переході,

q і p - евристичні параметри, що визначають «жадібність» та «стадність» алгоритму ($q + p = 1$);

Розв'язок, отриманий таким алгоритмом, як і в інших випадках, не буде точним. Але при вдалому підборі параметрів q і p можна добитися найбільш наближеного до оптимального результату.

Приклад функції, що реалізує мурашиний алгоритм для вирішення задачі комівояжера:

```
def ant_colony_optimization(distance_matrix, num_ants, num_iterations,
evaporation_rate=0.5, alpha=1, beta=2):
    pheromone_matrix = np.ones(distance_matrix.shape) / len(distance_matrix)
    best_tour = None
    best_tour_length = float('inf')
    for iteration in range(num_iterations):
        ant_tours = []
        for ant in range(num_ants):
            current_city = np.random.randint(len(distance_matrix))
            unvisited_cities = set(range(len(distance_matrix))) - {current_city}
            tour = [current_city]
            while unvisited_cities:
                next_city = select_next_city(pheromone_matrix, current_city,
unvisited_cities, alpha, beta)
                tour.append(next_city)
                unvisited_cities.remove(next_city)
                current_city = next_city
            tour_length = tour_distance(tour, distance_matrix)
            ant_tours.append((tour, tour_length))
            if tour_length < best_tour_length:
                best_tour = tour
                best_tour_length = tour_length
        pheromone_matrix *= evaporation_rate
        for tour, tour_length in ant_tours:
            for i in range(len(tour) - 1):
                pheromone_matrix[tour[i], tour[i+1]] += 1 / tour_length
        pheromone_matrix += pheromone_matrix.T
    return best_tour
```

Алгоритм імітації відпалу

Алгоритм імітації відпалу (Simulated annealing, SA) – стохастичний алгоритм що у своїй суті ґрунтується на імітації фізичного процесу, який відбувається при кристалізації речовини з рідкого стану в твердий. Процес кристалізації застосовується для виготовлення найрізноманітніших продуктів, починаючи від їжі та ліків і завершуючи паливом. Більшість продуктів агрохімічної та фармацевтичної промисловості в процесі розробки та виготовлення піддаються кільком етапам кристалізації. Сам відпал це вид термічної обробки, що полягає в нагріванні матеріалу до температури вище критичної, тривалій витримці матеріалу на цій температурі, та повільному охолодженні з метою наближення до вихідного стану. Алгоритм імітації відпалу є адаптацією алгоритму Метрополіс-Гастінгс. Це метод Монте-Карло Марковських ланцюгів та у статистиці використовується для отримання послідовності випадкових вибірок з розподілів, де прямий вибір є ускладненим, та таким методом обрану послідовність можна використовувати для “вгадування” розподілу даних.

Опис алгоритму імітації відпалу для вирішення задачі комівояжера:

1) Ініціалізувати змінні: температура, кількість ітерацій, рівень охолодження, поточний шлях та його вартість.
2) Згенерувати початковий шлях випадковим чином.
3) Повторювати наступну кроки до тих пір, поки не буде досягнуто задану кількість ітерацій або поки температура не досягне заданого мінімального рівня:
    - Створити новий шлях, змінивши поточний шлях випадковим чином.
    - Обчислити різницю між вартістю нового шляху та поточного шляху.
    - Якщо різниця вартості менша за 0, прийняти новий шлях як поточний.

- Інакше прийняти новий шлях з ймовірністю, яка залежить від рівня температури та різниці між вартістю нового та поточного шляху.
- Охолодити систему до певного рівня згідно з заданою функцією охолодження.

4) Повернути знайдений шлях та його вартість.

Для математичного опису даного алгоритму введемо позначення:

S – множина усіх станів системи;

f(s) - функція зміни стану;

s_i - стан системи на *i-му* кроці;

s_k - новий стан (кандидат);

t_{min}, t_i, t_{max}- мінімальна, поточна та вихідна температури відповідно;

T(t) - функція зміни температури;

E(s) - значення цільової функції.

Алгоритм починає працювати з вихідного стану s_1, початковою температурою $t_1 = t_{max}$ і з заданою мінімальною температурою t_{min}.

Для усіх номерів *i=1,2,...* поки $t_i > t_{min}$ повторюємо:

1) $s_k = f(s_{i-1})$;

2) $\Delta E = E(s_k) - E(s_{i-1})$;

3) якщо $\Delta E < 0$, тоді $s_i = s_k$;

4) інакше прийняття нового стану відбувається з деякою ймовірністю $\exp(-\Delta E / t_i)$;

5) вибрати випадкове число M на інтервалі (0,1);

6) якщо $\exp(-\Delta E / t_i) > M$, виконати перехід $s_i = s_k$, інакше перейти до наступного кроку;

7) зменшити температуру t : $t_{i+1} = T(t_i)$;

8) повернути останній стан s_i, $i = i+1$.

Генетичний алгоритм

Генетичний алгоритм (Genetic algorithm, GA) – це евристичний метод оптимізації, який базується на таких принципах еволюції, як: схрещування, селекція, формування нової популяції.

Вперше подібний алгоритм був опублікований у 1975 році Джоном Генрі Холландом в його книзі «Adaptation in Natural and Artificial». Алгоритм має широкий спектр застосування при вирішенні задач на графи, задач компонування або при створенні «штучного інтелекту». Нові популяції формуються до тих пір, поки користувача не задовільнить результат, або кількість поколінь дійде до заздалегідь обраного максимуму.

Задачу комівояжера також можна вирішити за допомогою генетичного алгоритму.

Спершу генерується випадкова сукупність потенційних розв'язків задачі. Кожен індивід популяції - це послідовність відвідування міст. Наприклад, якщо є п'ять міст, що треба обійти, то один індивід буде виглядати так:

[1, 3, 2, 5, 4].

Після створення нової популяції необхідно оцінити її ефективність, тобто визначити фітнес (оцінку пристосованості) кожної особини. Для задачі комівояжера, фітнес можна визначити як відстань, яку проходить кожна особина.

На основі отриманих результатів оцінки особин проводиться відсіювання для подальшої мутації найкращих кандидатів. Залишаються особини з найліпшою оцінкою пристосованості. Відбір кращих кандидатів можна здійснити за допомогою різних методів, таких як елітарний відбір (відбір кращих особин), турнірний відбір (вибір найкращих особин з декількох випадково обраних) та рулетковий відбір (відбір з ймовірністю, що пропорційна відповідності особини). Чим більші значення фітнесу, тим більша вірогідність, що особина буде вибрана для наступного покоління.

Після відбору кращих особин, на їх основі, утворюються нові популяції (селекція). Це можна зробити різними методами, наприклад:

Кросовер - це процес створення нових шляхів шляхом комбінації частин шляхів батьків. Одна з популярних стратегій кросовера для задачі комівояжера - це OX-кросовер (Order Crossover), в якому беруться випадкові ділянки двох батьків, а решта місць заповнюються з урахуванням порядку міст у вихідних особинах (в кожному шляху).

Мутація - це процес зміни одного або кількох міст шляху особини з метою створення нового шляху. Наприклад, випадково вибирається декілька міст в шляху і змінюється їх порядок.

Вставка - це процес заміни деяких міст в шляху на інші міста, що не були у вихідному шляху особини. Це може допомогти розв'язати проблему з локальними оптимумами.

Таким чином, генетичний алгоритм продовжує працювати, генеруючи нові покоління зразків-кандидатів та покращуючи якість рішення до тих пір, поки не буде досягнуто задовільного рівня точності.

Початкова популяція генерується зазвичай випадково. Єдиний критерій – достатня різноманітність особин, щоб уникнути потрапляння популяції в найближчий локальний екстремум, що не є глобальним.

Оцінювання популяції необхідно для того, щоб виявити більш пристосованих і менш пристосованих особин. Для підрахунку пристосованості кожної особини використовується функція пристосованості (цільова функція)

$f_i = f(G_i)$, де $G_i = \{g_{ik} | k = 1,2, \dots, N\}$ – хромосома i-ї особини, g_{ik}– значення k-го гена i-ї особини, N – кількість генів в хромосомі.

Селекція (відбір) необхідна, щоб вибрати більш пристосованих особин для схрещування. Наприклад розглянемо рулеточну селекцію. В даному варіанті селекції ймовірність p_i i-ї особини взяти участь в схрещуванні пропорційна значенню її пристосованості f_i і дорівнює $p_i = \frac{f_i}{\sum_j f_j}$.

Відібрані в результаті селекції особини (батьківські) схрещуються і дають потомство. Хромосоми нащадків формуються в процесі обміну генетичною інформацією (із застосуванням оператора кросовера) між батьківськими особинами. Створені таким чином нащадки складають популяцію наступного покоління.

Приклад коду, що реалізує генетичний алгоритм для задачі комівояжера:

```
def generate_population(size, num_cities):
    population = []
    for i in range(size):
        population.append(np.random.permutation(num_cities))
    return population


def fitness(population, distances):
    fitness_scores = []
    for individual in population:
        score = 0
        for i in range(len(individual) - 1):
            score += distances[individual[i], individual[i+1]]
        score += distances[individual[-1], individual[0]]
        fitness_scores.append(1/score)
    return fitness_scores


def select_parents(population, fitness_scores):
    idx1, idx2 = random.sample(range(len(population)), 2)
    parent1 = population[idx1]
    parent2 = population[idx2]
    if fitness_scores[idx1] > fitness_scores[idx2]:
        return parent1
    else:
        return parent2


def recombine(parent1, parent2):
    child = [-1] * len(parent1)
```

```
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child[start:end] = parent1[start:end]
    for i in range(len(parent2)):
        if parent2[i] not in child:
            for j in range(len(child)):
                if child[j] == -1:
                    child[j] = parent2[i]
                    break
    return child


def mutate(individual, probability):
    for i in range(len(individual)):
        if random.uniform(0, 1) < probability:
            j = int(random.uniform(0, len(individual)))
            individual[i], individual[j] = individual[j], individual[i]
    return individual


def next_generation(current_gen, elite_size, mutation_rate, distances):
    population_size = len(current_gen)
    fitness_scores = fitness(current_gen, distances)
    elite_count = int(population_size * elite_size)
    elite_individuals = np.array(current_gen)[np.array(fitness_scores).argsort()[-
elite_count:]].tolist()
    parents = []
    for i in range(population_size - elite_count):
        parents.append(select_parents(current_gen, fitness_scores))
    children = []
    for i in range(len(parents)):
        child = recombine(parents[i], select_parents(current_gen, fitness_scores))
        child = mutate(child, mutation_rate)
        children.append(child)
    return elite_individuals + children


def genetic_algorithm(num_cities, pop_size, elite_size, mutation_rate, generations,
distances):
    population = generate_population(pop_size, num_cities)
```

```
    for i in range(generations):
        population = next_generation(population, elite_size, mutation_rate, distances)
    best_individual = max(population, key=lambda x: 1/sum([distances[x[i], 
x[(i+1)%num_cities]] for i in range(num_cities)]))
    return best_individual, 1/sum([distances[best_individual[i], 
best_individual[(i+1)%num_cities]] for i in range(num_cities)])
```

Острівна модель генетичного алгоритму

Острівна модель (Island Model) генетичного алгоритму - це підхід до розв'язання задачі оптимізації, де популяцію розділяють на декілька груп або "острівків", які розв'язують одну й ту саму задачу паралельно. Кожен острівок має свій власний набір параметрів генетичного алгоритму і свою власну популяцію, яка еволюціонує відповідно до своїх параметрів.

Після певної кількості поколінь, найкращі рішення з кожного острівка переміщуються на інші острівки, де вони можуть стати новими батьківськими особинами, що починають новий процес еволюції. Цей обмін між острівками може бути здійснений з різними інтервалами часу та в залежності від різних параметрів.

Острівна модель може покращити здатність генетичного алгоритму до глобального пошуку, дозволяючи популяціям відкривати нові області пошуку. Також, паралельне вирішення задачі на декількох острівках може прискорити час розв'язання задачі, особливо якщо кількість острівків добре підібрана для конкретної задачі. Острівна модель генетичного алгоритму може бути реалізована, наприклад, шляхом поділу популяції на декілька підпопуляцій, які еволюціонують незалежно одна від одної, інколи з обміном частинами генетичної інформації між ними. Основна ідея полягає в тому, що кожна підпопуляція розв'язує задачу на своєму "острові", де має своє середовище і умови життєдіяльності. Це дозволяє збільшити мінімум функції пристосованості в кожній з підпопуляцій і зменшити ймовірність потрапляння в локальні мінімуми.

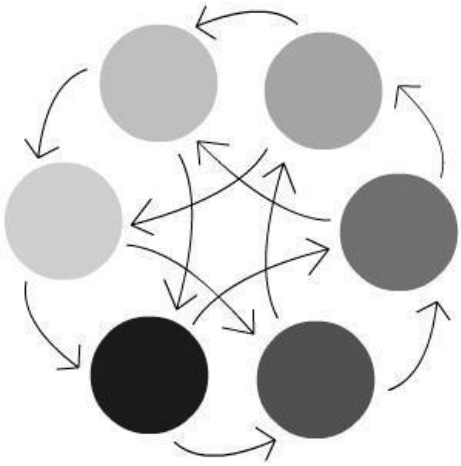


Рисунок 3 - Острівна модель генетичних алгоритмів

Приклад коду острівної моделі генетичного алгоритму може виглядати наступним чином:

```
# ініціалізувати популяцію для кожного острова
for i in range(num_islands):
    islands[i] = generate_population(pop_size)

# виконувати еволюцію для кожного острова
for gen in range(num_generations):
    # обміняти найкращих особин між островами
    exchange_best_individuals(islands)

    # здійснити еволюцію на кожному острові
    for i in range(num_islands):
        new_population = []
        # застосувати генетичні оператори для еволюції на острові
        for j in range(pop_size):
            parent1, parent2 = selection(islands[i])
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)
        islands[i] = new_population

# об'єднати популяції з кожного острова
population = merge_islands(islands)
```

РОЗДІЛ 2. ПОСТАНОВКА ДИНАМІЧНОЇ ЗАДАЧІ КОМІВОЯЖЕРА ТА МЕТОДИКА ЇЇ РОЗВ'ЯЗАННЯ

Пошук шляху в динамічному графі

Динамічний граф характеризується зміною в часі його сутностей графа, таких як вершини, ребра та ваги. Однією з поширених варіацій, які відбуваються з часом, є модифікація ваг ребер графа. Ці зміни також можуть служити як представлення встановлення або припинення крайових з'єднань за умови, що значення ваги дуги набуває нескінченного значення. Різні задачі комбінаторної оптимізації, які стосуються графів, можуть бути сформульовані в контексті динамічних графів. У багатьох випадках ця характеристика фундаментально змінює визначення проблеми, що вимагає різних підходів у статичному та динамічному сценаріях. Незважаючи на те, що це добре досліджена проблема комбінаторної оптимізації в існуючій літературі, питання визначення найкоротшого шляху отримало відносно менше уваги щодо динамічних варіацій графіка.

Класифікація динамічних графів може визначатися різними типами операцій, які вони здатні підтримувати. Динамічні графи можна класифікувати як повністю або частково динамічні. До таких операцій входять:

- Вставка вершини x, $x \notin V$.
- Видалення вершини x, $x \in V$.
- Вставка ребра x,y,w , де $x,y \in V$, $x,y \notin E$, $x \neq y$.
- Видалення ребра x,y , де $x,y \in V$, $x,y \in E$.
- Збільшення ваги x,y,w , де $x,y \in V$, $x,y \in E$, $w > W(x,y)$.
- Зменшення ваги x,y,w , де $x,y \in V$, $x,y \in E$, $w < W(x,y)$.

Частково динамічні графи підлягають певним обмеженням, згідно з якими дозволено лише вставлення та зменшення ваги ребер (інкрементні операції) або видалення та збільшення ваги ребер (декрементні операції).

Щодо проблеми найкоротшого шляху, вставка вершини x у граф вважається неефективною, оскільки x наразі не має спільних меж із жодною вершиною. У контексті другої операції передбачається, що видалення вершини x тягне за собою одночасне видалення всіх ребер, які з'єднані з нею в межах графа. Акт усунення ребра можна витлумачити як збільшення його ваги до нескінченності. Вплив вставки ребра залежить від відсутності будьякого іншого ребра, що з'єднує відповідні вершини, або від ваги нового ребра, що є нижчим за вагу попереднього ребра.

Повторне обчислення шляху з кожною зміною в графіку спричиняє великі витрати та неефективність. Було розроблено нові алгоритми для ефективного визначення найкоротшого шляху для динамічних графіків шляхом використання попередньо отриманої інформації для прискорення поточного процесу пошуку. Вищезазначені алгоритми включають, серед інших, D*, D* Lite, LPA*, HPA*, Adaptive A*.

Алгоритм Tree-Adaptive A*

Tree-AA* — алгоритм інкрементного евристичного пошуку найкоротшого шляху. Алгоритм Tree-AA* є розширенням алгоритму Adaptive A* (AA*), який застосовується для планування шляху. Він узагальнює адаптивний алгоритм Path-AA*, використовуючи суфікс шляху мінімальної вартості поточного пошуку A*, щоб дозволити наступному пошуку A* завершити процес пошуку раніше. Алгоритм Tree-AA* розширює алгоритм Adaptive A* повторно використовуючи суфікси шляхів мінімальної вартості поточного та всіх попередніх пошуків A*.

Алгоритм Adaptive A* працює за «принципом оновлення» в рамках ієрархічного шляху A* (HPA*). Якщо значення евристики h кожної вершини,

розглянутої пошуком A* з послідовним h-значенням, встановлюється fзначення цільової вершини мінус g-значення вершини, тоді отримані значення h знову збігаються і слабо домінують над вихідними значеннями h. Тобто нова евристика $h'(s) = f(s) - h(s)$ є негіршою за $h(s)$ при пошуку шляху до тієї самої цільової вершини. Таким чином, пошук A* з отриманими оновленими значеннями h проглядає не більше вершин, ніж пошук A* з вихідними значеннями h. Цільова вершина повинна залишатися незмінною від пошуку A* до пошуку A*, але стартовий стан може змінюватися, а деякі витрати руху можуть збільшуватися, але не зменшуватися. Таким чином, Adaptive A* може бути використаний для планування шляху з припущенням про вільний простір, що зазвичай робить пошук A* більш цілеспрямованим і, таким чином, пришвидшує їх.

Алгоритм Path-AA* базується на «принципі завершення» та розширенні «стратегії кешування шляху». Зокрема, якщо алгоритм має відомості про найкоротший шлях від даної вершини до цільової вершини (тобто використовує багаторазове дерево шляхів), і всі вершини шляху мають евристичні значення h, які еквівалентні їхнім цільовим витратам, тоді стандартний A* пошук може завершитися, коли він знаходиться на межі перевірки вершини в багаторазовому дереві шляхів повторного використання, включаючи цільову вершину. Отже, алгоритм Path-AA* має потенціал завершити пошук раніше, ніж стандартний пошук A*, який зупиняється лише тоді, коли цільова вершина знаходиться на межі обстеження. Мінімальна відстань між поточною вершиною агента та вершиною, яка є членом дерева повторного використання, у поєднанні зі шляхом від вершини дерева повторного використання до цільової вершини вздовж дерева повторного використання, становить найбільш прямий маршрут від теперішнього часу агента розташування до цільової вершини.

Алгоритм Tree-AA* є розширенням алгоритму Path-AA*, який використовує суфікси шляхів з мінімальною вартістю, отриманих із поточного та всіх попередніх пошуків A*. Такий підхід полегшує створення

багаторазової деревовидної структури. Структура зберігає шляхи з найнижчою вартістю від різних вершин до певної цілі, організовані у структурі дерева з цільовою вершиною, яка служить коренем. Якщо h значення всіх станів у дереві повторного використання дорівнюють їхнім кінцевим значенням і відомі шляхи з мінімальною вартістю від кількох вершин до цільової вершини, тоді прямий пошук A* може бути припинено, коли алгоритм знаходиться на грані розширення стану в дереві повторного використання.

Tree-AA* повинен підтримувати дві операції, а саме додавання шляху до дерева багаторазового використання та видалення шляхів із дерева багаторазового використання:

Додавання шляху до дерева багаторазового використання. Якщо пошук A* завершується, коли він збирається розглянути вершину у дереві багаторазового використання, включаючи цільову вершину, тоді Tree-AA* додає шлях-гілку від поточної вершини агента до вершини в дереві багаторазового використання до дерева багаторазового використання. Це робиться тому, що найкоротший шлях від поточного стану агента до вершини у дереві багаторазового використання та найкоротший шлях від вершини у дереві багаторазового використання до цільової вершини вздовж гілки дерева багаторазового використання формують найкоротший шлях від поточної вершини агента до цільової, а h −значення усіх вершин на шляху дорівнюють своїм кінцевим значенням (оскільки Adaptive A* оновлює значення h таким чином).

Видалення шляху із дерева багаторазового використання. Коли вага ребра у дереві багаторазового використання зростає, тоді Tree-AA* використовує найбільший префікс дерева багаторазового використання, який не містить ребр зі збільшеними витратами. Під префіксом дерева ми маємо на увазі верхню частину дерева, що включає його корінь. Це працює тому, що всі гілки результуючого дерева є шляхами з мінімальними витратами від деякої вершини до цілі та h-значенням усіх вершин в отриманому дереві все

ще рівні цільовим витратам. Коли вартість ребра від вершини s до вершини s′ зростає, тоді Tree-AA* знаходить найбільший префікс дерева багаторазового використання, видаляючи з нього як ребро s,s' , так і піддерево з коренем у вершині s.

Алгоритм Tree-AA* ефективно виконує вищезгадані операції шляхом підтримки двох змінних для кожної вершини та трьох змінних для кожного шляху $x = [v_0,...,v_n]$ у багаторазовому дереві. Тут v_0 позначає початкову вершину шляху, тоді як v_n представляє кінцеву вершину, яку пошук A* розширив би після завершення. Кожен окремий маршрут у багаторазовому дереві виділяється окремим цілим значенням, яке відповідає номеру пошуку A*, що вказує точку, в якій він був включений у багаторазове дерево, з нумерацією, починаючи з одиниці. Кожен маршрут у дереві повторного використання представляє префікс шляху з найменшою вартістю від даного стану до цільового стану. Вершини в дереві мінімальних шляхів виявляють властивість, згідно з якою їхні значення h еквівалентні їхнім кінцевим значенням, що призводить до строго монотонного зменшення вздовж шляху[3].

Моделювання поведінки поширення небезпеки. Моделювання перетворень і рухів реальних або абстрактних частинок у дискретному часі та просторі тепер можливо в сучасних обчислювальних системах, що дозволяє моделювати складні фізичні та хімічні явища. Процеси реакції-дифузії є одним із прикладів цієї широкої категорії подій. Вони являють собою наочне уявлення про дифузію речовин, нелінійну або порогову природу хімічних, фазових і біологічних змін і навіть прямі заміни типу «було – стало». Використання звичайних математичних моделей, заснованих на диференціальних рівняннях, може бути складним, якщо не неможливим, коли маємо справу з такими явищами через нелінійність і непостійність компонента реакції. Дискретне моделювання, побудоване навколо концепції клітинного автомата (КА) фон Неймана як заміни, виникло в результаті цього пошуку альтернативного методу для моделювання процесів.

Клітинний автомат

Математично модель на основі клітинного автомату можна задати четвіркою

$$\aleph = \langle A, X, \Theta, \rho \rangle, \tag{11}$$

де X – набір клітин, представлений їх іменами або індексами; A - алфавіт станів комірки в X; Θ – набір операторів локальної комірки, також званий глобальним оператором; ρ – режим роботи моделі. Стани клітин можуть бути представлені числовими значеннями, булевими векторами або символами. Крім того, імена комірок можуть складатися з натуральних чисел або векторів, що позначають координати дискретних просторових точок. Позначення $a,\ x$ використовується для представлення комірки $x \in X$, яка знаходиться в стані $a \in A$. Отже, добуток множини A та множини X є множиною, що охоплює всі можливі комірки в усіх можливих станах. Будь-яка його підмножина $\Omega = \{(a_1, x_1), \dots, (a_s, x_s)\}$, де $\{x_1, \dots, x_s\} = \mathrm{X}$ називається клітковим масивом моделі клітинного автомату. Масив станів комірок $(a_1, \dots, a_s)$ складають глобальний стан моделі клітинного автомату [8].

Локальний оператор визначається за допомогою функцій $\varphi_i : X \to X$, $i = 1, 2, \dots, q$, такі що $x \neq \varphi_1(x) \neq \varphi_2(x) \neq \cdots \neq \varphi_q(x)$. Їх підмножини

$$N(x) = \{x, \varphi_{1(x)}, \dots, \varphi_{n(x)}\},\ \ E(x) = \{\varphi_{n+1}(x), \dots, \varphi_q(x)\}, \qquad 0 \leq n \leq q, \tag{12}$$

$$S(x) = \{(u_0, x), \dots, (u_n, \varphi_n(x))\},$$

$$C(x) = \{(u_{n+1}, \varphi_{n+1}(x)), \dots, (u_q, \varphi_q(x))\}, \tag{13}$$

називаються сусідством, округом, локальною конфігурацією та контекстом клітки x.

Локальні оператори з множини Θ, призначаються коміркам клітинного автомату, таким чином щоб кожна комірка була пов’язана з одним або кількома операторами. Режим роботи такої моделі можна визначити шляхом визначення розподілу ймовірностей на множині $\Theta(x)$ операторів,

відображених у комірку x, яка керує порядком їх вибору під час процесу функціонування. Клітинний оператор x, який є довільним локальним оператором у $\Theta(x)$ представляється записом $\theta(x)$. Відповідно, застосувавши цей оператор до всіх комірок клітинного автомату, що перебувають в певній конфігурації $S(x)$ з контекстом $C(x)$, можемо отримати наступну конфігурацію

$$S'(x) = \theta(x)[S(x), C(x)] = \{(u'_0, x), (u'_{1,\varphi_1(x)}), \dots, (u'_{n,\varphi_n(x)})\}, \quad (14)$$

стани якої обчислюються

$$u_k = f_k(u_0, u_1, \dots, u_q), k = 0,1, \dots, n, \quad (15)$$

використовуючи функції переходів $f_k: A^{q+1} \to A$.

КА-модель працює на дискретній шкалі часу, позначеній $t = 0,1, ...$, де кожен інтервал часу називається ітерацією. Робота моделі базується на початковому масиві комірок $\Omega(0)$. Масив клітинок Ω на ітерації t + 1 дорівнює функції Θ, застосованій до Ω на ітерації t. Процес передбачає застосування локальних операторів $\theta(x) \in \Theta$ до всіх комірок $(u, x) = \in \Omega(t)$ у заздалегідь визначеному порядку, що призводить до глобального переходу $\Omega(t) \to \Omega(t+1)$ від поточного масиву комірок до наступного. Акт застосування функції $\theta(x)$ до комірки (u, x) , яка належить набору $\Omega(t)$, передбачає заміну станів комірок, що належать набору $S(x)$ у $\Omega(t)$ на стани комірок, які мають однакові назви або індекси з множини $S'(x) = \theta(x)[S(x), C(x)]$. Зауважимо, що в результаті виконання оператора значення контексту $C(x)$ не змінюється, але в місце конфігурації $S(x)$ записується нова — $S'(x)$.

Відповідно до моделі, глобальні переходи вважаються коректними, коли немає колізій під час обчислення $\Omega(t+1)$ за $\Omega(t)$. Зокрема, це означає, що не повинно бути спроб змінити стан однієї клітини більше одного разу в один і той же момент часу t.

Еволюція змодельованої моделі представлена у вигляді послідовності масивів комірок $\Omega(0)$, $\Omega(1)$,..., $\Omega(t)$,... через ітераційне застосування

глобального оператора Θ до масивів $\Omega(t)$. У випадку, коли ітераційний процес досягає збіжності, тобто існує таке t, що $\Omega(t) = \Omega + 1 = \cdots$, система досягає стану рівноваги. В іншому випадку клітинний автомат емулює процес, який демонструє коливальну або хаотичну поведінку.

В залежності від задання оператора переходів, КА класифікують на три рівня складності:

- *прості* КА-моделі, які використовують один локальний оператор, $|\Theta| = 1$, тобто здатні імітувати тільки одну елементарну дію;
- *складні* КА-моделі, у яких глобальний оператор працює з безліччю локальних операторів, $|\Theta| > 1$, і тому здатні імітувати складні процеси;
- *композиції глобальних операторів* Θ_i, такі, як послідовна та паралельна.

Вони всі Θ_i працюють послідовно чи паралельно, оновлюючи запропоновані їм підмасиви $\Omega_i \subset \Omega$ і використовуючи як контекстів набори клітин із усього масиву Ω.

Ітерація глобального оператора $\Theta(\Omega(t))$ у всіх випадках розглядається як процедура, яка передбачає застосування всіх $\theta_i(x) \in \Theta(x)$ до всіх $(u, x) \in \Omega(t)$ і складається з $|X| \cdot |\Theta|$ кроків. Послідовність дій визначається режимом роботи ρ. Для складної моделі цей режим складається з двох складових ρ_x та ρ_Θ , визначеними як просторова компонента, яка керує послідовністю вибору комірки, і операторна компонента, яка керує послідовністю локального вибору оператора. У випадку композицій глобальних операторів, кожен оператор функціонує у своєму окремому режимі, і їхня взаємодія залежить від типу композиції [8].

Приклад функції мовою python для реалізації простої моделі клітинного автомату:

```
def update_cells():
    new_grid = [[False] * GRID_HEIGHT for _ in range(GRID_WIDTH)]
    for x in range(GRID_WIDTH):
        for y in range(GRID_HEIGHT):
            # Перевірка чи пожежа вже є в поточній клітині
```

```
            if grid[x][y]:
                new_grid[x][y] = True
                # Розповсюдження пожежі на сусідні клітини
                for i in range(-1, 2):
                    for j in range(-1, 2):
                        if i == 0 and j == 0:
                            continue
                        if x + i >= 0 and x + i < GRID_WIDTH and y + j >= 0 and y + j < GRID_HEIGHT:
                            # Ймовірність поширення пожежі в сусідню клітину
                            if random.random() < FIRE_CHANCE:
                                new_grid[x + i][y + j] = True
    return new_grid
```

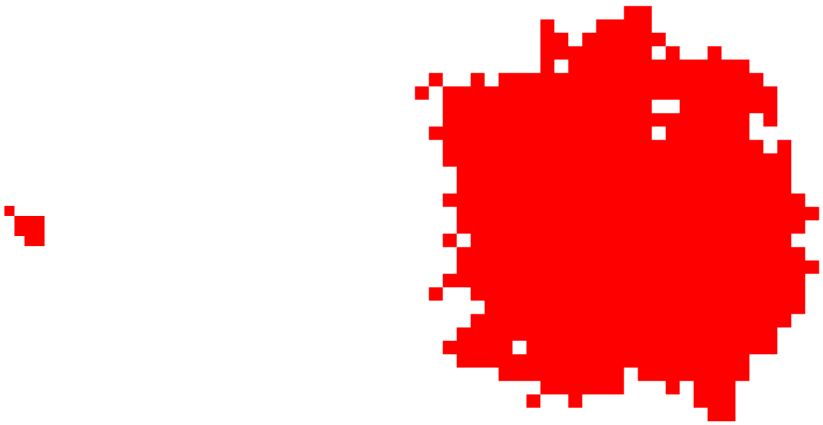


Рисунок 4 - Приклад розповсюдження пожежі, як простої КА-моделі

Режими функціонування моделей клітинного автомату

В залежності від вибраного режиму роботи КА-моделі спостерігатимуться доволі різні еволюції клітинного автомату при таких самих початкових конфігураціях. Просторова компонента може функціонувати в режимах:

- *синхронний* – оператор $\theta(x)$ застосовується до всіх клітин $(u, x) \in \Omega(t)$ в будь якому порядку, проте нові значення (u', x) спочатку зберігаються в додаткову пам'ять Ω', після чого заповнюється конфігурація $\Omega(t+1)$;

- *асинхронний* – оператор $\theta(x)$ застосовується до клітин $(u, x) \in \Omega(t)$ послідовно у випадковому або заданому порядку, причому нові значення станів клітин заміняють теперішні відразу.

В складній моделі для задання режиму роботи необхідно додатково визначити режим функціонування другої операторної компоненти. Є два різних варіанти задання оператора $\theta(x) \in \Theta(x)$ для застосування до обраної клітини $x \in X$: *детермінований* і *стохастичний*. Зауважимо, що при складній КА-моделі ітерація складається з $|X| \cdot |\Theta|$ виборів клітини і оператора на її застосування.

Композиція глобальних операторів передбачає об'єднання кількох глобальних операторів в уніфіковану структуру, де кожен складовий оператор може мати просту або складну структуру та працювати відповідно до свого попередньо встановленого режиму. Ця композиція може бути *послідовною*, коли ітерація складається з обчислень L проміжних глобальних переходів $\Omega(t+1) = \Theta L(\Theta L - 1(\dots(\Theta 1()\Omega(t))))$, або *паралельною*, коли множини клітин X і операторів Ω поділені рівномірно на L підмножин

$X = X^{(1)} \cup \dots \cup X^{(L)}$ та $\Omega = \{\Omega^{(1)} \cup \dots \cup \Omega^{(L)}\}$, при чому усі підмножини $\Omega(j) \in \Omega$ оновлюються одночасно відповідними глобальними операторами $\Theta(j)$.

Інваріанти моделей клітинного автомату

Поняття інваріанту моделі клітинного автомату ввела в своїй роботі О. Л. Бандман [11] в пошуках методу переходу від фізикохімічного опису моделюючого процесу до відображення клітинного автомату і його інтерпретації. Інваріант, який є безрозмірним і не залежить від підходу математичного представлення, дозволяє встановити коефіцієнти масштабування між фактичними та модельними значеннями. Інваріантна модель пов'язує просторовий (розмір комірки в метрах) і часовий (тривалість

ітерації в секундах) масштаби, оскільки клітинні автомати моделюють просторову динаміку. Таким чином, ці шкали не можуть бути самостійно обрані взагалі. КА-моделі реакційно-дифузійних процесів, включають дифузію та реакцію, кожна зі своїм інваріантом. Інваріанти складних процесів не завжди є функцією інваріантів їх складових, а функції масштабування не завжди можна виразити аналітично через інваріант. У цих складних обставинах лише обчислювальний експеримент і таблиці та криві, що їх з'єднують, можуть визначити інваріанти та масштаби.

Інваріанти дифузійних процесів. Математичні моделі дифузійних процесів зазвичай описують в неперервному вигляді за допомогою рівнянь Лапласа. Вони характеризуються безрозмірним коефіцієнтом $D = \frac{d\tau}{h^2}$, де d — коефіцієнт дифузії, виражений в м^2 за секунду; τ - крок за часом у секундах; h — крок по простору в метрах. Для цих моделей можна використовувати інваріант, що рівний відповідному параметру D, оскільки він не залежить від початкових умов.

З першого закону Фіка для дифузії в ізотропному середовищі, який говорить, що коефіцієнт дифузії - це маса речовини, яка рухається через одиницю площі за одну одиницю часу з градієнтом концентрації одиниці, легко зрозуміти, яке значення D є. У булевому масиві комірок градієнт концентрації дорівнює 1 на межі між областю, де всі клітини мають стан u = 1, і областю, де всі клітини мають стан u = 0. Отже, кількість речовини (кількість частинок), яка розповсюджується на Δt = 1 дорівнює ймовірності того, що дві граничні комірки поряд одна з одною змінять стани.

Два найбільш відомими булевими КА, які здатні моделювати процес дифузії є:

- *Синхронний клітинний автомат*

$$\aleph = \langle A, X, \Theta \rangle, \qquad A = \{0,1\}, \qquad X = \{x : x = (i,j)\}. \tag{16}$$

Для його функціонування визначаються парна і непарна підмножини X і шаблон $T_{2\times 2} = \{(i,j), (i,j+1), (i+1,j+1), (i+1,j)\}$, який визначає

локальну конфігурацію і локальний оператор $\Theta(i,j)$, який є стохастичним

$$u'_k = \begin{cases} u_{(k+1)mod4}, & if\ rand < p \\ u_{(k-1)mod4}, & jf\ rand \geq (1-p) \end{cases}, \quad (17)$$

де p – ймовірність застосування $\Theta(i,j)$. Ітерація спершу відбувається для множини X_0 а потім для X_1 . В [x] показано, що

$$D_{2\sigma} = \begin{cases} 1 & if\ p = 0.5 \\ p, & if\ p < 0.5 \end{cases}, \quad (18)$$

- *Асинхронний клітинний автомат*

  В цьому клітинному автоматі, виду (16), кожна випадково обрана клітина обмінюється станом з однією із сусідніх клітин з рівною ймовірністю. Для цього задається шаблон з локальною конфігурацією

$$S(x) = \{(u_0, x), (u_1, x + a_1), (u_2, x + a_2), (u_3, x + a_3), (u_4, x + a_4)\}, \quad (19)$$

  Очікується, що на одній ітерації частинка перейде з комірки $(1, x)$ в одну з суміжних комірок $(0, x + a_k)$ з імовірністю 0,5, причому

$$D_{2\sigma} = \begin{cases} 0.5 & if\ p = 1 \\ 0.5 \cdot p, & if\ p < 1 \end{cases}, \quad (20)$$

Визначення масштабів параметрів моделі КА, а саме довжини h сторони комірки в метрах і часу ітерації τ в секундах, полегшується знанням інваріанта, оскільки можна вибрати одну зі шкал на основі заданих умов, тоді як іншу можна визначити за допомогою математичної залежності

$$D = \frac{d \cdot \tau}{h^2}, \quad (21)$$

Інваріант реактивних процесів. У контексті моделей клітинних автоматів хімічних реакцій інваріантом рекомендується вважати безрозмірний коефіцієнт $R = k \cdot \tau$. Тут k являє собою константу швидкості реакції за секунду часу, яка є добре встановленим параметром для більшості досліджених реакцій. Крім того, τ позначає тривалість ітерації клітинного автомата, виміряну в секундах. Слід зазначити, що інваріантність хімічної реакції моделі клітинного автомата не залежить від величини клітини, таким чином, коефіцієнти масштабування h і τ виключають один одного. Отже, h

можна визначити на основі умов задачі. Інваріант R фізично означає зміну концентрації $\Delta C/C$ реагенту протягом однієї ітерації τ. Параметр τ служить зв'язком між теоретичними значеннями моделі та відповідними фактичними значеннями. Якщо модель включає одиночну реакцію, доцільно вибрати τ як еквівалент тривалості зазначеної реакції. У випадках, коли змодельований процес охоплює численні реакції та потенційно додаткові дії, співвідношення швидкостей залежать від ймовірностей реалізації відповідних локальних операторів, які залежать від усіх інваріантів.

Інваріант процесу «поширення фронту». Явища, які належать до сімейства реакційно-дифузійних явищ, у яких реакція описується нелінійною функцією

$$F(u) = \alpha u(1-u), \quad F(0) = F(1) = 0, \quad 0 < \alpha \leq 1, \ 0 \leq u \leq 1, \tag{22}$$

І є процесами «дифузії зі зростанням кількості речовини». Для характеристики цих процесів використовуються диференціальні рівняння типу

$$u_t = D \cdot u_{xx} + F(u), \tag{23}$$

де D – коєфіцієнт дифузії.

Для моделювання процесу поширення фронту можна використати асинхронний клітинний автомат $\aleph_\alpha = \langle A, X, \Theta(X) \rangle$ з булевим алфавітом $A = \{0,1\}$, множиною клітин $X = \{(i,j): i,j = 0, \dots, N\}$ і глобальним оператором $\Theta(X) = \Phi_z\big(\theta_d(x), \theta_r(x)\big)$ (Φ_z – один з способів композиції). Локальний оператор дифузії θ_d визначається на шаблоні

$$T(i,j) = \{(i,j), (i,j+1), (i-1,j), (i+1,j), (i,j-1)\}, \tag{24}$$

і при його застосуванні на обрану клітку (u_0, x_k)її стан u_0 заміняється на стан клітини-сусіда u_l з однаковою ймовірністю:

$$(u_0 = u_l)\&(u_l = u_0), \ \ if \ \frac{l-1}{4} \, rand_1 < \frac{l}{4} \ \& \, rand_2 < p_d, l = \overline{1,4}. \tag{25}$$

Інваріантом процесу є *швидкість поширення фронту*. В [x] аналітичним шляхом доведено, що швидкість поширення фронту при t → ∞ залежить від коефіцієнтів дифузії (D) та швидкості реакції (α) наступним чином

$$V = 2\sqrt{D \cdot \alpha}. \tag{26}$$

Інваріант процесу обмеженої дифузією агрегації. Процес агрегації обмеженої дифузією (diffusion-limited agregation), не може бути представлений у вигляді диференціального рівняння. Його перша модель була заснована на процесі руху в дискретному просторі та взаємодії частинок.

Процесу відповідає еволюція асинхронного клітинного автомату, $\aleph = \langle A, X, \Theta(X)\rangle, A = \{0,1,b\}, X = \{(i,j)\}$. Локальний оператор $\Theta(i,j)$ визначається як композиція локального оператора наївної дифузії θ_dта оператора прилипання $\theta_r = \Theta(i,j) = \Phi_z\big(\theta_d(i,j), \theta_r(i,j)\big)$. Таким чином, функція переходу тут визначається

$$u_0' = \begin{cases} b, & if\ \ (u_0 = 1)\ and\ (u_l = b)\ \ and\ \ rand < p_r \\ u_0, & else \end{cases}, \tag{27}$$

Зауважимо, що коефіцієнтом прилипання однозначно визначається фрактальна розмірність, яка є основною характеристикою реального явища агрегації обмеженої дифузією. Тому її можна брати за інваріант для моделі клітинного автомату процесу агрегації обмеженої дифузією.

Багато-агентний пошук шляхів

Багато-агентий підхід до процесу евакуації в динамічному середовищі (Multi Agent Evacuation with Dynamic Obtancles) — це анонімна форма задачі багатоагентного пошуку шляхів у динамічно змінному графі. Процес евакуації відбувається в неорієнтованому графі, позначеному як $G = (V, E)$, де вершини класифікуються як безпечні або небезпечні. Зокрема, множина

вершин V може бути виражена як об'єднання двох непересічних підмножин, а саме S і D, які відповідають безпечним і небезпечним групам вершинам відповідно. Анонімна форма задачі багатоагентного пошуку шляхів у динамічно змінному графі передбачає що цільова вершина агента не є чітко визначеною вершиною графа, а може бути будь-якою вершиною з визначеної групи цільових вершин. Граф G виступає середовищем на якому відбувається поширення небезпеки.

Мета полягає в тому, щоб визначити набір шляхів для групи агентів $\mathrm{A} = \{a_1, a_2, \dots, a_k\}$, що дозволяє їм пройти до безпечних вершин S уникаючи небезпечні вершини D. Множина вершин D може змінюватись відповідно до оператора, який відображає процес поширення небезпеки. Оскільки граф являє собою регулярну сітку, для моделювання небезпеки зручно обрати моделювання клітинного автомату. Кожен агент з A, починає свій шлях з окремої вершини, тим самим гарантуючи, що в будь-який даний момент часу вершину займає не більше ніж один агент. Розв'язок задачі включає план $\pi = \{c_1, c_2, \dots, c_m\}$,, де кожен елемент плану $c_m(a) \in S \forall a \ni A$ і $c_t: A \to V$.

Математично задачу багато-агентної евакуації в динамічному середовищі можна задати п'ятіркою $\mathrm{E} = [\mathrm{G} = (\mathrm{V}, \mathrm{E}), \aleph, \mathrm{A}, c_0, \mathrm{D}, \mathrm{S}]$, де G — середовище, $\aleph$ – модель клітинного автомату поширення небезпеки в середовищі G, A — набір агентів, $c_0 : A \to V$— початкова конфігурація агентів, D і S такі, що $D \subseteq V$, $S \subseteq V$, $V = D \cup S$ вважаючи, що $D \cap S \neq \emptyset$ і $|\mathrm{S}| \geq \mathrm{k}$ представляють набір під загрозою та набір безпечних вершин відповідно.

Як правило, намагаються скоротити тривалість багатоагентної евакуації в динамічному середовищі, яка визначається як загальна кількість часу, необхідної для досягнення останнім агентом цільової вершини від початку евакуації. Тому, за цільову функцію для задачі багатоагентного пошуку шляхів у динамічно змінному графі вибрано функцію «makespan».

Для вирішення поставленої задачі пропонується використання ієрархічної моделі поведінки агентів, у якій зазвичай більш обізнані агенти

дотримуються плану евакуації, розробленого централізованим алгоритмом. Для цього агенти діляться на два види: лідери та послідовники. Мотивація такого рішення – евакуаційна ситуація в середовищі коли не усі агенти здатні самостійно знаходити шляхи до виходів або є обмеженими у можливостях. Наприклад, агенти можуть представляти викладачів і учнів школи. Лідери шукають вихід із будівлі, а послідовники утворюють зграї навколо лідерів, слідують за ними та евакуюються з їх допомогою.

Динамічна задача комівояжера

Динамічна задача комівояжера, також відома як Dynamic Traveling Salesmen Problem (DTSP), є модифікованою версією традиційної задачі комівояжера. Необхідність цієї модифікації виникає, коли вхідні дані зазнають поступових змін з часом. В такому випадку метою буде визначення найбільш ефективного маршруту, який відвідує всі міста, враховуючи появу нових міст або зміну відстаней між ними під час виконання алгоритму. Якщо іншими словами, то проблема комівояжера передбачає пошук найкоротшого замкнутого маршруту серед набору з N міст. TSP — це комбінаторна задача оптимізації, яка охоплює різні аспекти, окрім простого визначення. Розв’язання цієї задачі за допомогою алгебраїчних алгоритмів потребує часу, який експоненціально зростає разом із розміром проблеми, що робить її класифікованою як NP-повну задачу. Проблема комівояжера має численні практичні застосування в науці та техніці, включаючи маршрутизацію транспортних засобів, розробку інтегральних схем, планування автоматизованих керованих транспортних засобів, керування роботами тощо.

Оскільки динамічна модифікація задачі комівояжера передбачає постійні зміни, її вирішення є складним завданням, можливо й складнішою за TSP. Для її ефективного вирішення можна використовувати різні підходи, такі як динамічне програмування, адаптивні алгоритми, еволюційні методи, а також . їх гібридні поєднання.

Метаевристичні алгоритми

Метаевристичний алгоритм - це тип алгоритму оптимізації, який використовує евристичні (емпіричні) методи для вирішення складних оптимізаційних задач. Вони базуються на природних явищах, поведінці та ідеях, які існують у реальному світі. Ці алгоритми спрямовані на пошук найкращого рішення з великого простору можливих альтернатив [10].

Ключовою особливістю метаевристичних алгоритмів є те, що вони здатні знаходити рішення в складних просторах, де інші методи є неефективними або незастосовними. Метаевристичні алгоритми також можуть працювати з проблемами які мають великою кількістю обмежень невизначеності і нелінійності.

Основна ідея метаевристичних алгоритмів полягає у використанні імовірнісних або ітераційних процедур, які імітують процес пошуку рішення. Принципи, що відтворюють природні чи соціальні взаємодії або інші стимули, можуть бути використані для пошуку та оцінки, модифікації та покращення рішень.

Деякі з найвідоміших метаевристичних алгоритмів включають генетичні алгоритми, мурашині колонії, вільний пошук, пошук прибутку, ройові алгоритми та симуляції відпалу. Кожен з цих алгоритмів має свої особливості, але всі вони спрямовані на пошук оптимального рішення для великого простору можливих альтернатив.

Хоча метаевристичні алгоритми не гарантують знаходження оптимального рішення, вони можуть знаходити дуже хороші наближені рішення і є дуже ефективними для багатьох практичних завдань. Вони використовуються в багатьох сферах, таких як оптимізація виробничих процесів, розподіл ресурсів, транспортні маршрути, проблеми планування та штучний інтелект.

Таким чином, метаевристичні алгоритми є потужним інструментом для вирішення складних оптимізаційних задач, які вимагають евристичного

підходу і пошуку наближених рішень. Вони можуть знайти ефективні рішення там, де інші методи є неефективними або неможливими.

Адаптація мурашиного алгоритму для динамічної задачі комівояжера

Алгоритм мурашиної колонії є одним із методів, які застосовуються для вирішення динамічної задачі комівояжера. Цей алгоритм поєднує ідеї імітації поведінки мурах у природі та колективного інтелекту.

У контексті динамічної задачі комівояжера, алгоритм мурашиної колонії використовується для пошуку найкоротшого шляху, що проходить через всі міста, при зміні вхідних даних протягом часу. Застосування цього алгоритму дає можливість враховувати зміни у розташуванні міст або у відстанях між ними, що можуть виникати протягом виконання алгоритму.

Такий підхід до розв’язання DTSP застосовується в статі, що була розглянута на конференції у Брюсселі [1]. Переглянувши результати роботи приведеної в цій статті, стане зрозуміло, з якими проблемами та на якому етапі з ними можна зіштовхнутись, або, можливо, як можна їх вирішити у випадку евакуації.

В наведеній DTSP мурахи часто стикаються з перевантаженнями. Це забезпечується тим, що затори створюються виключно на дорогах, які є частиною оптимального маршруту, визначеного AS у цей конкретний момент. Крім того, ці затори демонструють певну органічну схему поступового збільшення, а потім зменшення, доки їх не буде вирішено. Далі приводяться модифікації, що б підходи до налаштування алгоритму AS для ефективної обробки таких динамічних сценаріїв.

Було запроваджено мінімальний поріг τ_0 для рівня феромонів на кожному шляху в AS-DTSP. Також іншою впровадженою модифікацією є метод струшування або струсу, тобто збурення навколишнього середовища, щоб вирівняти рівні феромонів певним чином. Коли рівень феромонів на

шляху стає значно вищим, ніж на всіх інших шляхах, що ведуть з міста, майже завжди буде обрано його. Незважаючи на те, що це забезпечує стабільний вибір хорошого кандидатського шляху в статичних сценаріях, це заважає мурахам досліджувати альтернативні маршрути, коли на певній дорозі виникає затор.

Процес струшування змінює співвідношення між кількістю феромонів на всіх дорогах, водночас гарантуючи збереження відносного порядку: якщо істинність наступного твердження $\tau_{i\,j}(t) > \tau_{i,\,j,}(t)$ зберігається до процесу струшування, воно також зберігає свою істинність і після струшування. Для струшування використовується логарифмічна формула:

$$\tau_{ij} = \tau_0 \cdot \left(1 + \log\left(\tau_{ij}(t)/\tau_0\right)\right), \tag{28}$$

На *рисунку 5* показано детальний графік змін значення феромонів. Формула приводить до того, що значення феромонів, що наближаються до τ_0, незначно зміщуватимуться в напрямку τ_0, тоді як вищі значення будуть зміщуватися відносно більше в напрямку τ_0. Важливим зауваженням є те, що τ_0 є мінімальним значенням для τ, забезпечуючи виконання умови $\tau_{i\,j} \geq \tau_0$.

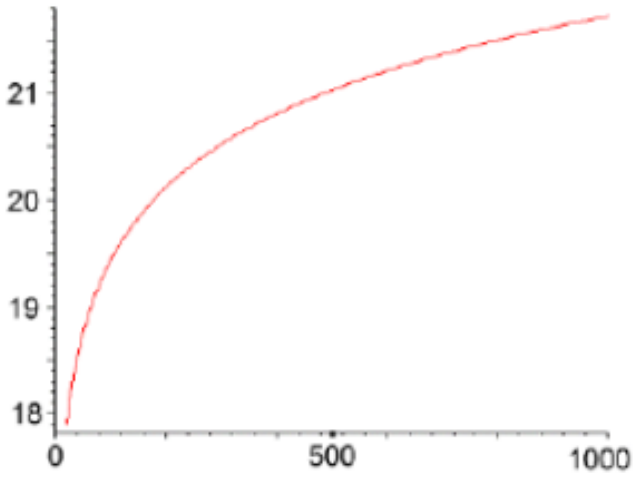


Рисунок 5 – Функція струшування, рівень значення до та після процесу струшування

Можливою проблемою методом струсу є його глобальний характер, тобто вплив на значення феромонів на всіх дорогах. Коли виникає серйозна проблема, зокрема, коли затор виникає біля краю карти. З цієї причини використовується локальне струшування, тобто наведена вище формула застосовується лише до тих шляхів, які знаходяться на відстані меншій за $p \cdot MaxDistans$ ($p -$ це розподіл ймовірності за набором невідвіданих міст), від одного з двох міст, між якими виникає затор. $MaxDistans$ є максимальною відстанню між будь-якими двома містами у вихідній задачі (без заторів), а p є значенням між 0 і 1.

Псевдокод методу локального збурення:

if the distance between cities (a, b) changed then
  for every road (i, j) **do**
    if $(d_{ai} < p \cdot MaxDist) \vee (d_{bi} < p \cdot MaxDist) \vee$
      $(d_{aj} < p \cdot MaxDist) \vee (d_{bj} < p \cdot MaxDist)$ **then**
      $\tau_{ij} = \tau_0 \cdot (1 + log(\tau_{ij}/t_0))$
    endif
  endfor
endif

При p = 0 значення феромонів не змінюються. Глобальне струшування еквівалентне при p = 1, оскільки всі значення феромонів тепер підлягають струсу.

Одним з тестів, що наведено у роботі [1], є застосування мурашиного алгоритму для двадцять п’яти міст у маршруті. Особливістю обчислення результатів є те, що міста розташовані на перетині цілих ліній сітки. Періодично виникають затори між дуже близькими містами, які, можливо, знаходяться на оптимальному маршруті. Ці місця були вибрані емпірично шляхом виконання початкової задачі через стандартну реалізацію алгоритму мурашиної колонії для задачі комівояжера.

Отже, було виявлено деякі пари міст, які майже завжди знаходяться поруч одне з одним у найкращому знайденому рішенні. За допомогою запропонованою автором адаптації можна впевнено стверджувати, що всі затори впливають на оптимальне рішення. Було встановили три затори, які і є

об'єктом дослідження. У цьому випадку, на протязі ітерацій з 100 по 150, утворюється один затор на певній дорозі. Від 200 до 250 ітерацій цей затор зникає, але одночасно з’являються два інших на інших дорогах. Всі затори з’являються та зникають з приростом та зменшенням в 10 одиниць, з одним приростом кожні 5 ітерацій. Це дозволяє алгоритмам вчасно реагувати на незначні зміни. Якщо затор виникає враз, алгоритм не може демонструвати свою здатність змінювати свій маршрут, навіть якщо пробка продовжує зростати. Для алгоритму мурашиної колонії були встановлені наступні параметри: $\alpha = 1, \beta = 6, m = n = 25, Q = 100, \tau_0 = 10^{-6}$.

Наступним кроком задача ускладнюється, кількість міст підвищується до ста. На цьому прикладі буде видно, як алгоритм вирішує подібну, але більш складну та динамічну задачу. Шляхом випадкового розташування було створено 100 міст на сітці заданого розміру. Після деякого періоду з початку роботи алгоритму починаються створюватися нові затори кожні 50 ітерацій, поступово збільшуючи довжину шляху за рівними кроками. Протягом однакових 25 ітерацій остання створена пробка поступово зникає. Затор виникає на дорозі, що знаходиться на поточному найкращому маршруті. Цей систематичний підхід до введення заторів не є реалістичним, але він має перевагу в тому, що дозволяє чітко порівнювати різні підходи.

Отже, припустивши, що розв’язок без заторів, який досягається після певного часу оптимізації, має довжину x, починається додавання нових заторів кожні 50 ітерацій з одночасним видаленням попереднього затору. В результаті довжина кінцевого рішення становить x + довжина одного затору.

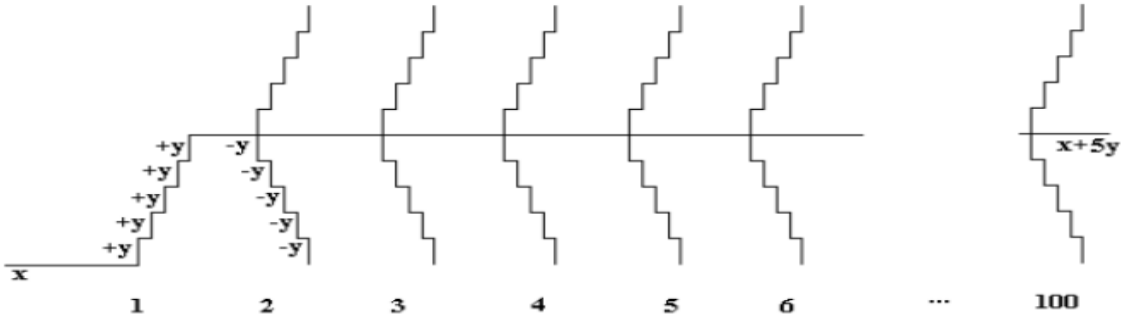


Рисунок 6 – Проблема інтенсивності заторів у задачі розміром в 100 міст.

Оскільки всі затори створюються через 5 рівних кроків у наведеному тестовому середовищі завжди є 5∗y заторів (крім перших 20 ітерацій). Інтенсивність появи та затухання заторів наведена на *рисунку 6*. З використанням заздалегідь встановленого довільного порядку міст, моделювання першого затору починається на виїзді з міста номер 1, другий - на виїзді з міста номер 2 і так далі. Затори зникають у тому ж порядку. Таким чином, автор стикається з проблеми, які трохи схожі кожного запуску експерименту. Кожен експеримент складається з 5000 ітерацій після періоду запуску, оскільки створюється точно 100 заторів. Кожен експеримент повторювався 10 разів.

Середню довжину маршруту для струсу, довжиною в 25 міст, при p = 1 можна побачити *рисунку 7*. До точки 1 цей екземпляр є статичним. Також можна побачити дві цікаві точки, розташовані безпосередньо після 100 і 200 ітерацій: це динамічні частини проблеми задачі. У обох випадках ми спостерігаємо пік на графіку. Низькі піки свідчать про швидке відновлення після зростаючої пробки. Два інші цікаві значення, які вказують на здатність відновлюватися після змін, це результат безпосередньо перед другим піком і результат після останньої ітерації.

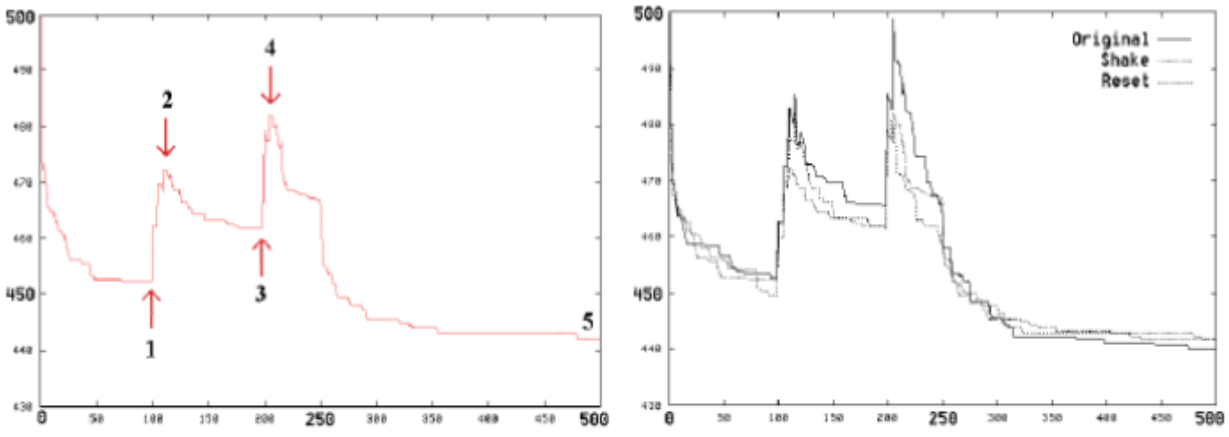


Рисунок 7,8 - Приклад тривалості шляху по 25 містам протягом 500 ітерацій, із позначенням розташування контрольних точок (p = 1); -Приклад результатів 10 запусків кожної з наведених на діаграмі стратегій, Shake(p=1).

Після вирішення задачі без заторів, середнє оптимальне рішення після 500 ітерацій складає 441,4. Оцінки всіх стратегій, що розглядає автор протягом всього моделювання, наведені на *рисунку 8*. Результати за чотирма контрольними точками наведені на *рисунку 9*.

Оригінал	485.2	465.4	498.7	**440.1**
p = 0.1	477.7	463.0	489.0	443.4
p = 0.25	479.5	465.0	485.0	440.6
p = 1	**472.0**	461.8	481.8	441.9
reset	483.6	**461.2**	**480.0**	441.6

Рисунок 9 – результати за чотирма контрольними в DTSP при кількості міст 25.

Гібридне поєднання алгоритмів мурашиної колонії та симуляції відпалу

Гібридні алгоритми, засновані на оптимізації мурашиних колоній та симуляції відпалу, поєднують принципи та ідеї цих двох метаевристичних методів для підвищення ефективності та якості розв'язання оптимізаційних задач.

ACO - це метаевристичний алгоритм, заснований на поведінці мурах, який знаходить оптимальні шляхи в графі або мережі. Мурахи спілкуються між собою, залишаючи феромонні сліди на своїх шляхах. Цей феромон приваблює інших мурах і спонукає їх обирати шлях з більшими феромонними слідами. Таким чином, алгоритм використовує механізми позитивного підкріплення та збагачення феромонів для пошуку найкращого маршруту.

SA - ще один алгоритм метаевристики, натхненний процесом охолодження металів. Він імітує процес термічного охолодження матеріалів, коли система перебуває в різних температурних станах. Алгоритм допускає постійну ймовірність переходу до гіршого рішення на початку процесу, але ця ймовірність поступово зменшується в міру охолодження системи. Це

дозволяє уникнути стиснення до локальних мінімумів і дозволяє шукати глобально оптимальне рішення.

Гібридні алгоритми, що поєднують ці два методи, покращують пошук оптимального рішення, використовуючи переваги обох. Він використовує феромонну матрицю, подібну до мурашника, для зберігання та передачі інформації про якість шляху. Крім того, алгоритм може використовувати механізм відбору та грубої сили симуляції відпалу, щоб знаходити нові рішення та уникати попадань в локальні оптимуми.

Гібридні алгоритми, засновані на оптимізації мурашиних колоній та симуляції відпалу, мають кращу здатність знаходити глобально оптимальні рішення і можуть забезпечити кращий баланс між інтенсивним пошуком та дослідженням простору рішень. Цей підхід широко використовується для розв'язання різноманітних оптимізаційних задач і може бути адаптований до конкретних вимог та характеристик задачі.

Тема гібридного поєднання алгоритмів мурашиної колонії та симуляції відпалу представлена в статті [2]. В ній розповідається, що DTSP - це комбінаторна динамічна оптимізаційна задача, що складається з TSP та серії ітерацій TSP. Кожна ітерація створюється шляхом модифікації попередньої ітерації. Автором запропонований новий гібридний метаевристичний алгоритм для розв'язання DTSP.

Алгоритм поєднує два метаевристичні принципи - оптимізацію ACO та SA: ACO використовує матрицю феромонів для зберігання та передачі інформації про якість шляху, тоді як SA імітує процес охолодження, що дозволяє перейти до гіршого рішення з певною ймовірністю.

Гібридний алгоритм також використовує знання про динамічні зміни, передаючи інформацію, зібрану на попередніх ітераціях, за допомогою феромонної матриці. Важливість гібридизації та використання знань про динамічне середовище перевіряється на еталонних DTSP різного розміру.

Отримані результати порівнюються з чотирма іншими сучасними метаевристичними підходами і це порівняння дає гарний результат на

користь запропонованого алгоритму. Крім того, поведінка алгоритму була проаналізована з різних точок зору, таких як швидкість збіжності до локального оптимуму, еволюція різноманітності популяції під час оптимізації, залежність від часу та обчислювальна складність алгоритму.

Нижче, в якості прикладу, буде наведено псевдокод, що був приведений у статті [2] для використання алгоритму ACO. На кожній ітерації $i = 0, 1,2, \dots, I$ алгоритм повторює постійний процес у циклі. Результатом кожної ітерації є вартість C_i та шлях R_i, що представляють і-й розв'язок TSP.

Першим важливим кроком у кожному циклі є ініціалізація матриці феромонів F. Деталі цього процесу описано . Зауважте, що на нульовій ітерації ($i = 0$) для ініціалізації використовується невизначений шлях $R_{i-1} = R_{-1}$. При $i = 0$ алгоритм не використовує цей параметр, тому це не впливає на його функцію.

Процес пошуку розв'язку задачі і-го комівояжера відбувається протягом декількох поколінь. У кожному поколінні для генерації множини розв'язків використовується евристична інформація TSP, отримана з попереднього покоління у вигляді феромонної матриці. Кожен розв'язок r_a знаходиться незалежно для кожної мурахи в колонії ($a = 1,2,.., N_a$) знаходять незалежно.

Для кожної мурашки розв'язок створюється шляхом поступового вибору вершин з множини V_i вставки їх R_i в циклі. Для цього використовується множина U, яка містить вершини, яких ще немає на маршруті r_a. Цикл продовжується до тих пір, поки множина U не стане порожньою, тобто поки маршрут r_a не буде містити всі вершини з множини V_i. Кожна вершина $U_j \in U$ є кандидатом на включення до маршруту r_a, і ймовірність p_j цього включення обчислюється згідно з наступним рівнянням

$$p_j(U_j) = \frac{D_{kj}^{-\alpha} \cdot F_{kj}^{\beta}}{\sum_{U_i \in U} D_{kj}^{-\alpha} \cdot F_{kj}^{\beta}}. \quad (29)$$

На основі цих ймовірностей вибирається наступна вершина маршруту за принципом простого колеса рулетки.

Відстань між останньою вершиною в маршруті r_a, та вершиною-кандидатом U_j позначається як D_{kj}. F_{kj} – феромонний слід між останньою вставленою вершиною маршруту та вершиною U_j. Коефіцієнт α контролює вплив відстані між вузлами на ймовірність, а коефіцієнт β контролює вплив феромону на ймовірність. Встановлено, що ймовірність зменшується зі збільшенням відстані.

Наведений алгоритм, для спрощення, використовує фіксовану кількість поколінь для знаходження рішення TSP, реальна реалізація включає дві додаткові умови завершення для окремої ітерації. Перша умова - максимальне обмеження часу; друга - максимальна задана кількість поколінь.

Важливою частиною в алгоритмі є визначення принципу ініціалізації феромонної матриці на початку кожної ітерації TSP. У першій фазі всі елементи N × N феромонної матриці F встановлюються на значення 1, що представляє початкову силу феромонного зв'язку між двома вершинами. У другій фазі використовується рішення з попередньої ітерації, щоб оновити матрицю. Це ґрунтується на припущенні, що лише невеликий відсоток вершин змінюється між ітераціями від проблеми DTSP. Тому інформація з рішення, знайденого на попередній ітерації, також залишається актуальною на поточній ітерації. Ця інформація інтегрується у феромонну матрицю шляхом зміцнення слідів між сусідніми вершинами маршруту R_{i-1}. Сила цього зміцнення контролюється коефіцієнтом ініціалізації феромону $\tau \geq 1$. У випадку нульової ітерації, де немає відомого маршруту з попередньої ітерації, друга фаза циклу функції не виконується. Також можна пропустити другу фазу, у такому випадку , коли задачу DTSP потрібно розв'язувати, як набір незалежних задач TSP.

Реалізація демонстрації принципу вибору наступної вершини з множини U на основі ймовірностей $p_1, p_2, \dots, p_{|U|}$. У цьому алгоритмі

використовується простий принцип колеса рулетки. $RandU(a,b)$ представляє генератор псевдовипадкових чисел, що має рівномірний розподіл у діапазоні від a до b.

Algorithm_DTSP_ACO $(V,N_g,N_a,\rho,\delta,\tau,\alpha,\beta)$

$C = 0$
$R = \varnothing$
for $i = 0$ **to** I **do** // Iterations
  $C^i = \infty$
  $R^i = \varnothing$
  $F =$Initialize_Pheromone_Matrix $\left(\left|V^i\right|, R^{i-1}, \tau\right)$
  for $g = 1$ **to** N_g **do** // Generations
    $c^{best} = \infty$
    for $a = 1$ **to** N_a **do** // Route for each ant
      $k = 1$
      $c^a = 0$
      $r^a = \left\{V_1^i\right\}$
      $U = V^i - \left\{V_1^i\right\}$
      while $U \neq \varnothing$ **do**
        for each $U_j \in U$ **do**
          compute $p_j\left(U_j\right) = f\left(r_k^a, V^i, F, \alpha, \beta\right)$
        $r_{k+1}^a =$Select_Next_Vertex$\left(U, p_1, p_2, \ldots, p_{|U|}\right)$
        $U = U - \left\{r_{k+1}^a\right\}$
        $c^a = c^a + \cos t\left(r_k^a, r_{k+1}^a\right)$
        $k = k + 1$
      $r_{k+1}^a = V_1^i$
      $c^a = c^a + \cos t\left(r_k^a, r_{k+1}^a\right)$
      if $c^a < c^{best}$ **then do**
        $c^{best} = c^a$
        $r^{best} = r^a$
      if $c^{best} < C^i$ **then do**
        $C^i = c^{best}$
        $R^i = r^{best}$
      for each $F_{m,n} \in F$ **do** // Evaporate pheromones
        $F_{m,n} = F_{m,n} \cdot (1 - \rho)$
      for $k = 1$ to N **do** // Update pheromones
        $F_{r_k^{best}, r_{k+1}^{best}} = F_{r_k^{best}, r_{k+1}^{best}} + \delta \cdot \left(C^i / c^{best}\right)$
  $C = C + C^i$
  $R = R + \left\{R^i\right\}$
return C,R

Приклад псевдокоду, що був наведений в статті [2] для даного алгоритму: псевдокод функції що формує матрицю феромонів

Initialize_Pheromone_Matrix (N,R^{i-1},τ)

$F = (F_{m,n}) \in \mathbb{R}^{N \times N}$
for $F_{m,n} \in F$ **do**
  $F_{m,n} = 1$
if $i > 0$ **then do**
  for $k = 1$ to N **do**
    $F_{R_k^{i-1}, R_{k+1}^{i-1}} = \tau$
return F

Псевдокод, що представляє реалізацію принципу вибору наступної вершини з множини U:

Select_Next_Vertex ($U, p_1, p_2, \ldots p_{|U|}$)

1. $p_{sum} = \sum_k p_k$
2. $p_{rnd} = \text{RandU}(0, p_{sum})$
3. **for** $k = 1$ **to** $|U|$ **do**
4. **if** $p_{rnd} \le \sum_{l=1}^{k} p_l$ **then do**
5. return U_k

N_g – кількість поколінь алгоритму;

N_a – кількість мурах в колонії;

ρ – коефіцієнт розповсюдження феромонів;

δ – коефіцієнт оновлення розповсюдження феромонів;

τ – коефіцієнт ініціалізації феромонів;

α – коефіцієнт правдоподібності відстані;

β – коефіцієнт ймовірності розповсюдження феромону.

Частина моделювання відпалу (SA) алгоритму ACO-SA була інспірована процесом відпалу, що застосовується в металургії, де матеріал нагрівається та контрольовано охолоджується з метою зменшення дефектів. Основна ідея SA полягає у прийнятті гірших рішень з певною ймовірністю, що розширює простір пошуку для глобального оптимуму. SA може бути застосований як до безперервних, так і до дискретних просторів станів [9].

Наведений алгоритм SA використовується лише для проблеми TSP та її серії ітерацій. Це пов'язано з необхідністю гібридизації з ACO. Щоб використати SA для DTSP, алгоритм можна виконувати повторно з використанням рішення з попередньої ітерації як вхідне.

T_{max} – максимальна температура стану системи;

T_{min} – мінімальна температура стану системи;

γ – коефіцієнт охолодження;

$n_{1_{max}}$ – максимальна кількість перетворень за покоління;

$n_{2_{max}}$ −максимальна кількість перетворень за покоління.

Алгоритм пропонує псевдокод алгоритму SA для TSP. Вхідним параметром в алгоритм є маршрут R_{SA}, який представляє початкове рішення. Це може бути будь-яке допустиме рішення, знайдене іншим алгоритмом, наприклад, алгоритмом найближчого сусіда, або згенеровано випадковим чином, тобто містить всі вершини випадковим чином.

Алгоритм працює у поколіннях, та протягом кожного покоління використовується одне значення для температури. Після завершення генерації покоління температура знижується, і починається наступне покоління. Коли температура стає нижчою за мінімальний поріг T_{min}, алгоритм завершує свою роботу і повертає найкраще знайдене рішення.

Особливу увагу в алгоритмі приділяється процесу перетворення поточного рішення у нове рішення.

Детально це питання досліджується в самій статті [2]. Нове рішення замінює вихідне рішення з певною ймовірністю. Якщо нове рішення є кращим за вихідне, його завжди замінюють. В іншому випадку ймовірність заміни залежить від різниці у якості обох рішень і поточної температури. Високі температури збільшують шанси прийняти гірші рішення.

Псевдокод моделювання SA:

Algorithm_TSP_SA $(V^i, R^{SA}, T_{max}, T_{min}, \gamma, n_{1max}, n_{2max})$

1. $r^{SA} = R^{SA}$
2. $c^{SA} = C^{SA} = \sum_{k=1}^{N} \left| R_k^{SA} - R_{k+1}^{SA} \right|$
3. $T = T_{max}$
4. **while** $T \geq T_{min}$ **do** // Generations
5. $n_1 = n_2 = 1$
6. **while** $n_1 \leq n_{1max}$ **and** $n_2 \leq n_{2max}$ **do** // Transformations
7. $r^{SA'} = \text{Transform\_Solution}\left(r^{SA}, |V^i|, T, T_{max}, T_{min}\right)$
8. $c^{SA'} = \sum_{k=1}^{N} \left| r_k^{SA'} - r_{k+1}^{SA'} \right|$
9. $p\left(r^{SA} \to r^{SA'}\right) = f\left(c^{SA}, c^{SA'}, T\right)$
10. **if** $\text{RandU}(0,1) \leq p\left(r^{SA} \to r^{SA'}\right)$ **then do** // Replacements
11. $r^{SA} = r^{SA'}$
12. $c^{SA} = c^{SA'}$
13. $n_2 = n_2 + 1$
14. **if** $c^{SA} < C^{SA}$ **then do**
15. $R^{SA} = r^{SA}$
16. $C^{SA} = c^{SA}$
17. $n_1 = n_1 + 1$
18. $T = \gamma \cdot T$
19. **return** C^{SA}, R^{SA}

На *рисунку 10* зображено розвиток ентропії під час оптимізації для заданих параметрів. Мінімальні та максимальні межі ентропії також відзначені. Синя крива відображає усереднене значення, яке було обчислене на основі 60 експериментів оптимізації за допомогою алгоритму ACO-SA. Світло-блакитна область показує діапазон ентропії, який був спостережений у цих 60 експериментах.

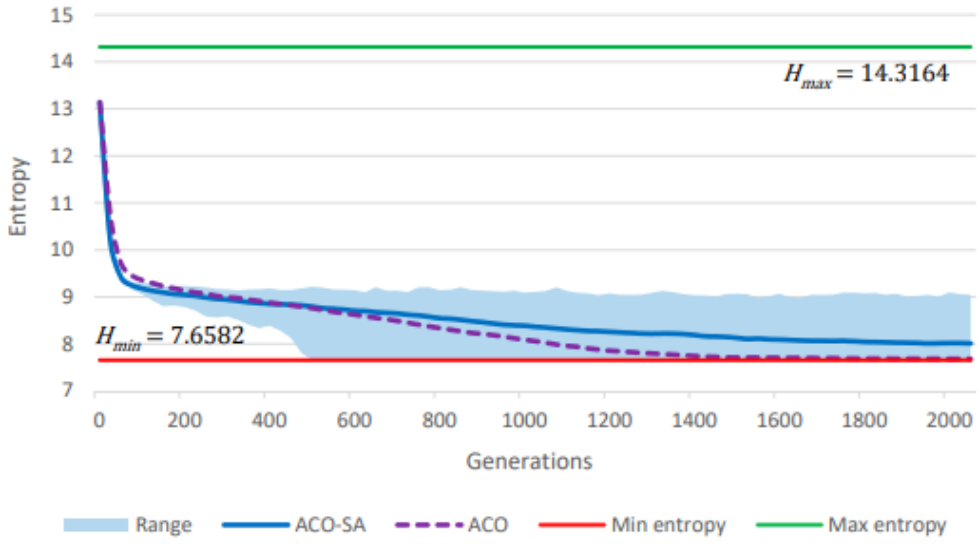


Рисунок 10 – результати 60 експериментів оптимізації за алгоритмом ACO-SA

Тут

$$H_{max} = log_2 \frac{1}{N}; \tag{30}$$

$$H_{min} = log_2 \frac{1}{N_a \cdot N}; \tag{31}$$

формули, за якими рахується значення ентропії популяції.

На *рисунку 11* показано лінійну залежність алгоритму від кількості поколінь. На наступному *рисунку 12* також представлена залежність алгоритму від кількості вершин. Для всіх шести еталонних екземплярів був розрахований середній час T_1, необхідний для однієї генерації, і його залежність від кількості вершин N. З *рисунку 12* видно, що існує квадратична залежність, що підтверджує оптимізаційний підхід.

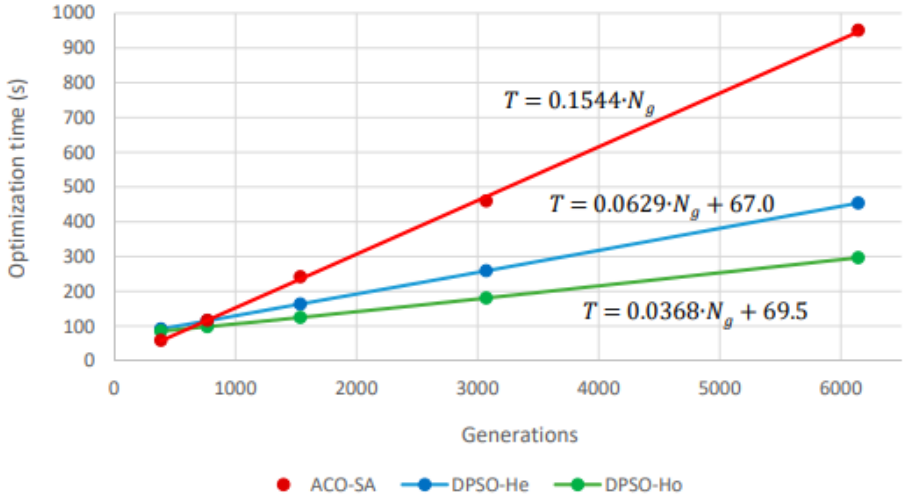


Рисунок 11 – Збіжність часу оптимізації від кількості поколінь.

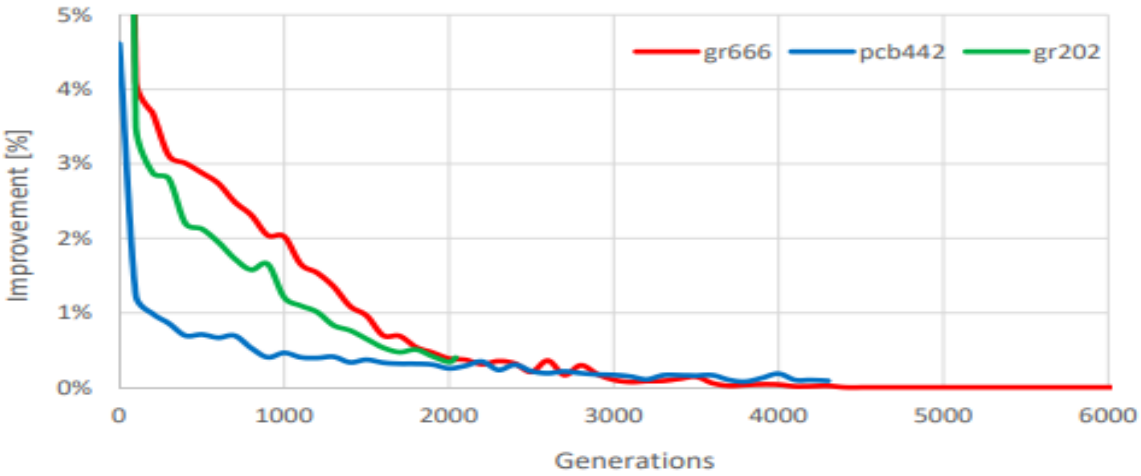


Рисунок 12 – Покращення результатів завдяки SA

РОЗДІЛ 3. ПРИКЛАДИ ЗАСТОСУВАННЯ ЗАДАЧІ КОМІВОЯЖЕРА І ПОБУДОВИ ШЛЯХІВ ПЕРЕСУВАННЯ В УМОВАХ НАДЗВИЧАЙНИХ СИТУАЦІЙ

Часова спроможність

Евристичні алгоритми надають прийнятні рішення для більшості випадків, але їх оптимальність не може бути гарантована. Це означає, що може бути порушено принцип оптимальності Беллмана, який стверджує, що будь-яке майбутнє рішення, незалежно від початкового стану системи, повинно залишатися оптимальним [3]. Іншими словами, оптимальність рішення повинна зберігатися на всіх підзадачах - це властивість, яку розуміють як тимчасову спроможність. Зменшення розмірності задачі сприяє більш швидкому знаходженню оптимального результату. Тому евристичний алгоритм є кращим варіантом для знаходження оптимального рішення. Оскільки TSP полягає в послідовності відвідування точок транспортом, до неї можна застосувати оцінку тимчасової спроможності [4].

Позначимо:

P - набір тестових прикладів для завдання; $p \in P$ - окремий приклад з цього набору; $s(p)$ - набір рішень, отриманих за допомогою евристичного алгоритму для задачі. Кожне рішення представляє собою послідовність міст, що відвідуються кожним транспортним засобом протягом періодів T.

Вихідне рішення $s(p)$ можна розбити на частини, відповідні кожному періоду $t = 1, 2, \dots, T-1$. Кількість вузлів, пройдених протягом перших t періодів, обчислюється за формулою:

$$n\big(t, s(p)\big) = \frac{n_0 \cdot t}{T}, \tag{32}$$

де n_0 - кількість вузлів у тестовій задачі, $s^+(t,p)$- частина маршруту $s(p)$, що відповідає порядку обходу вузлів після періоду t, а $s^-(t,p)$- частина

маршруту $s(p)$ до періоду t [5]. Також $s(p) = s^{+}(t,p) \cup s^{-}(t,p)$. Розглянемо поточну задачу $p\big(s^{-}(p,t)\big)$, вона відрізняється від початкової TSP тим, що скорочена багата кількість вузлів, що вже входять в частину маршруту $s^{-}(t,p)$.

Через $s(p(s^{-}(p,t)))$, позначимо розв'язок поточної задачі, що був знайдений за допомогою евристичного алгоритму SA. Припустимо, що $s(p)-$рішення, яке ми будемо вважати заможним за часом, якщо для кожного $t = 1, 2, \dots, T-1$ виконується наступна нерівність

$$f(s^{+}(t,p)) \le f(s\big(p(s^{-}(p,t))\big)). \tag{33}$$

Для кожного рішення $s(p)$, необхідно провести N обчислювальних експериментів, які будуть спрямовані на перевірку часової ефективності рішення. Позначимо $b(s(p),t)$ як кількість експериментів, в яких ця властивість порушується після періоду t.

Якщо рішення є оптимальним, то за критерієм оптимальності Беллмана виконується рівність:

$$\sum_{k=1}^{T-1} b(s(p),t) = 0\,. \tag{34}$$

Динамічна адаптація методу імітації відпалу

У статті [5] авторами було детально розглянуто метод динамічної адаптації алгоритмів для задачі комівояжера. Я розгляну ідеї з цієї статті до випадку методу імітації відпалу. Динамічну адаптацію тут можна описати таким чином: при початку методу генерується N рішень для всіх вузлів в заданій TSP, після чого обирається рішення, у якого цільова функція має найменше значення, тобто оптимальне. Далі кожен маршрут розбивається на T періодів, після кожного розбиття зменшується кількість вузлів. Після зміни

періоду алгоритм повторюється. Маршрут потрібно змінити, коли настає чергова ітерація алгоритму, і знайдене рішення буде найкращим на поточний момент. Прогрес у виконанні методу досягається завдяки тому, що на кожному кроці кількість розглянутих вузлів зменшується [6].

Загальний алгоритм методу імітації відпалу при використанні динамічної адаптації можна описати наступним чином:

- Сформувати N результатів виконання алгоритму SA для TSP;
- Визначити оптимальний маршрут серед знайдених N результатів;
- $t = 1$ до $t < T$ умова початку циклу алгоритму DSA;
    - Формуємо наступні n результатів вирішення TSP;
    - Обирається результат з найменшим значенням цільової функції $f(p') \to min$;
    - Якщо $f(p') \leq f(p)$, то приймається результат p' як оптимальний, інакше алгоритм продовжує виконання;

Моделювання надзвичайної ситуації

При моделюванні таких небезпеки, такої, як пожежа або повінь, є кілька методів, які можна застосувати. Коротко розглянемо деякі з них:

- Моделювання на основі параметрів розподілу: цей метод використовує розподіл ймовірностей для визначення рівня небезпеки на різних ділянках. Наприклад, при моделюванні пожежі можна використовувати розподіл ймовірності виникнення пожежі в окремих точках або розподіл ймовірності поширення пожежі по території. Це дозволяє врахувати випадковість і неоднорідність небезпеки.
- Моделювання на основі гідродинамічних рівнянь: гідродинамічні рівняння, такі як рівняння Нав'є-Стокса, можна використовувати для моделювання повеней та інших небезпек. Ці рівняння описують рух

рідин і розподіл води у просторі та часі. Моделі можуть враховувати топографічні характеристики, географічні особливості та гідрологічні параметри.

- Агентно-орієнтоване моделювання: цей підхід моделює окремих агентів, які взаємодіють у небезпечних ситуаціях. Наприклад, моделювання евакуації може використовувати агентів, які реагують на небезпеку і визначають маршрути евакуації. Цей підхід може враховувати поведінку та взаємодію окремих агентів у небезпечних ситуаціях.
- Моделі систем масового обслуговування: моделі систем масового обслуговування можна використовувати для моделювання небезпек та їхнього впливу на системи обслуговування, такі як пожежні та медичні служби. Ці моделі можуть аналізувати час реагування, ефективність і пропускну здатність системи при виникненні небезпеки.

Ці методи не є вичерпним переліком, і вибір конкретного методу буде залежати від типу загрози, доступної інформації та цілей моделювання.

Реалізація методу імітації відпалу

Реалізація динамічної адаптації з умовою невизначеності стану ситуації виявилась досить складною задачею, та потребує більше часу на освоєння та розуміння доступних алгоритмів. Можливо, правильним кроком в вирішенні проблем динамічної адаптації стане застосування методів на базі штучного інтелекту, такі як Q-learning та Deep Learning.

Мовою програмування python мною було розроблено програму, що знаходить оптимальний маршрут комівояжера в задачі розміром в 100 міст, методом симуляції відпалу.

Основні кроки методу відпалу для задачі комівояжера такі:

Ініціалізація: Починаємо пошук зі стартового розташування міст (потенційний маршрут). Це може бути будь-яка початкова послідовність міст або випадково згенерований маршрут.

Вибір нового розв'язку: Здійснюється випадкова зміна маршруту, наприклад, шляхом обміну двох міст місцями або перестановки декількох міст.

Обчислення енергії системи: Розраховується різниця довжини між попереднім та новим маршрутами.

Прийняття або відхилення нового розв'язку: Якщо маршрут-кандидат є кращим, то він приймається і стає поточним. Інакше прийняття нового стану відбувається з деякою ймовірністю

Охолодження: Після кожної ітерації температура зменшується, що зменшує ймовірність прийняття гіршого розв'язку. Цей процес дозволяє алгоритму виходити з локальних оптимумів та продовжувати пошук кращих розв'язків.

Повторення: Кроки 2-5 повторюються до досягнення критерію зупинки, тобто поки значення температури більше за мінімальне задане значення.

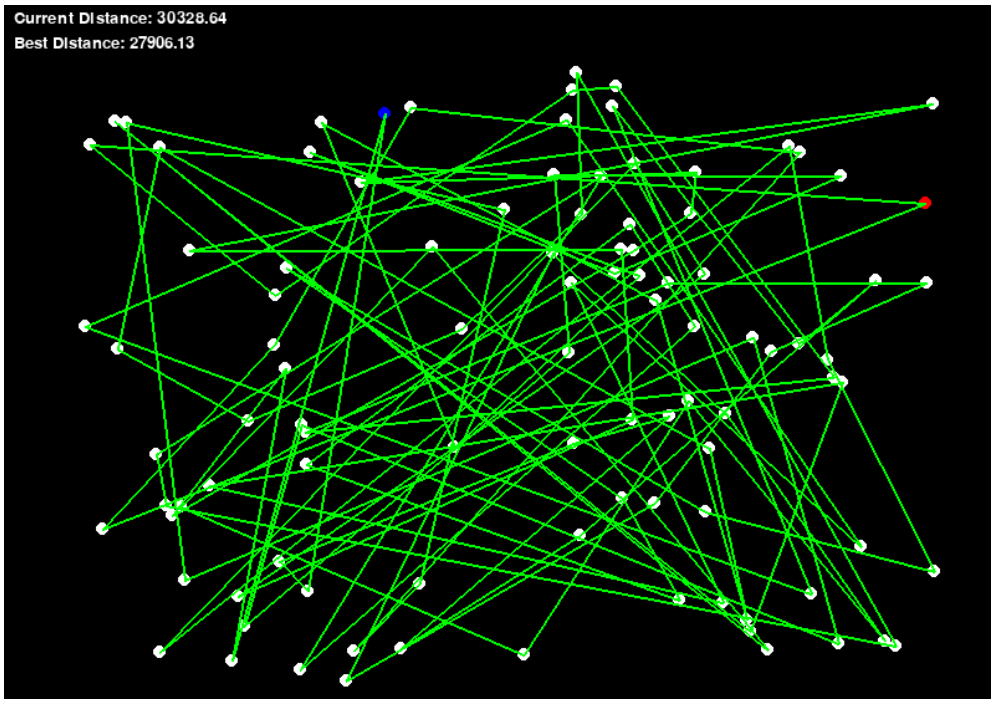


Рисунок 13 – Виведення на дисплей результатів роботи програми пошуку алгоритмом SA для TSP.

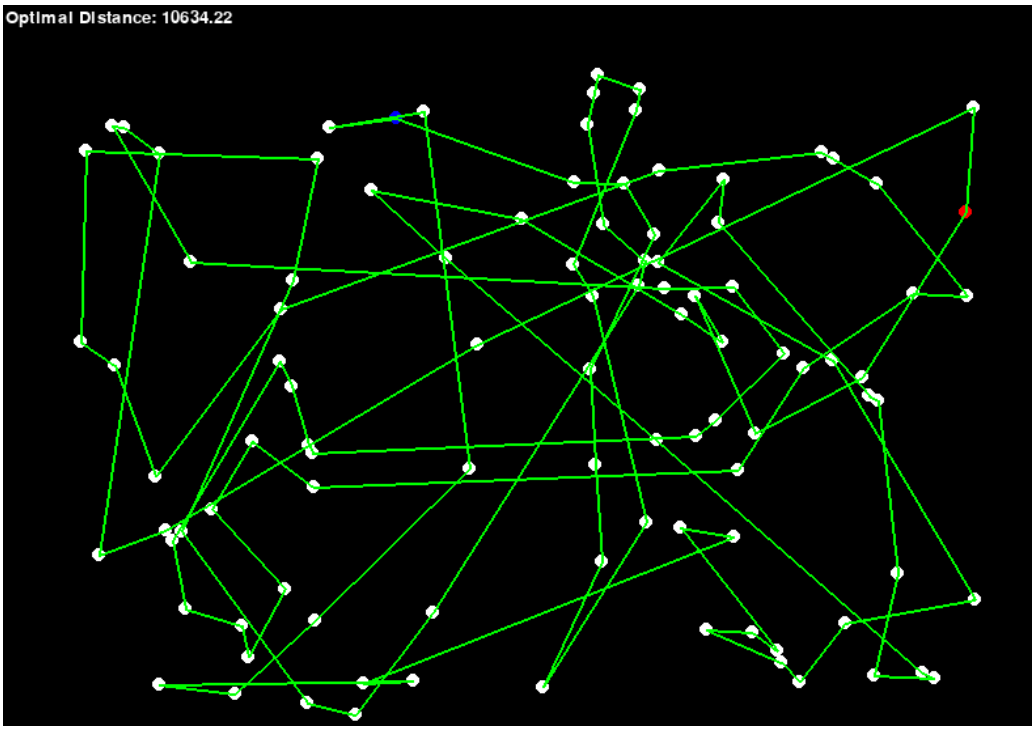


Рисунок 14 - Оптимальний маршрут TSP

Результом роботи програми, є знаходження оптимального шляху задачі комівояжера розміром в 100 міс, що генеруються кожного разу випадковим чином, в режимі реального часу, та візуалізація процесу пошуку на екрані. Знайдений оптимальний маршрут приведений на *рисунку 14* .

ВИСНОВКИ

В роботі було проведено дослідження існуючих методів пошуку оптимального шляху виходу в графі, а також в його динамічній адаптації. Наведені методи включення та виключення ребер з графу, також наведені алгоритми, що базуються на цих модифікаціях. Було наведено та розібрано декілька адаптацій алгоритмів пошуку гамільтонова циклу в задачі комівояжера, а також деякі динамічні адаптації алгоритмів, що застосовуються для вирішення динамічної задачі комівояжера. Тобто були розглянуті алгоритми, що при здійсненні пошуку оптимального шляху адаптуються до змін, що зазнають вхідні данні з плином часу.

Були розглянуті можливі моделі імітації небезпеки, як модель клітинного автомату. Також було розроблено мовою програмування python найпростішу модель КА, що імітує модель розповсюдження пожежі.

Тема є досить складною, та все ще безліч авторів в своїх наукових роботах зазначають, що це ще не кінець, та всі алгоритми можна продовжувати модифікувати, щоб знайти більш оптимальний результат. Так як рішення динамічної задачі комівояжера може знайти застосування в логістиці та проблемах з транспортом, в маркетингу та продажі, в налаштуванні маршрутизації дронів тощо.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. C. J. Eyckelhof and M. Snoek «Ant systems for a dynamic TSP» Conference: Ant Algorithms, Third International Workshop, ANTS 2002, Brussels, Belgium, 2002
2. P. Stodola, K. Michenca «Hybrid Algorithm Based on Ant Colony Optimization and Simulated Annealing Applied to the Dynamic Traveling Salesman Problem», науковий журнал “Entropy”, 2020
3. Р. М. Трохимчук «Теорія графів», Київ, 1998
4. В. В. Мезеря, П. І. Щербатюк «Механічний підхід до моделювання поведінки натовпу», подана до журналу «Актуальні проблеми автоматизації та інформаційних технологій», 2018
5. Shirokikh, V. A., Zakharov, V. V. Dynamic Adaptive Large Neighborhood Search for Inventory Routing Problem, 2015
6. D. Helbing, I. Farkas, T. Vicsek «Simulating dynamical features of escape panic», Nature Journal, 2000
7. Мезеря В. В., Щербатюк П. І. «Метод калібрування коефіцієнтів математичної моделі симуляції поведінк натовпу», подано на Всеукраїнську науково-практичну конференцію «Наукова Україна: проблеми сучасності та перспективи майбутнього», 2018
8. Wolfram S. «New kind of science», науковий журнал “Wayback Machine”, 2007
9. Hodicky, J.; Prochazka, D.; Prochazka, J. «Training with and of Autonomous System—Modelling and Simulation Approach. In Proceedings of the Modelling and Simulation for Autonomous Systems», Italy, 2017
10. Li, W. «A Parallel Multi-Start Search Algorithm for Dynamic Traveling Salesman Problem. In Proceedings of the International Symposium on Experimental Algorithms», Greece, 2017
11. O. Bandman «Cellular automata composition techniques for spatial dynamics simulation. Simulating Complex System by Cellural Automata», Berlin, 2010

ДОДАТОК

Код програми що реалізує алгоритм SA для TSP

```
import pygame
import math
import random
import sys

# Константи для вікна
WIDTH = 800
HEIGHT = 600

# Константи для кольорів
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Клас для представлення міста
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self, screen, color):
        pygame.draw.circle(screen, color, (self.x, self.y), 5)

# Функція для обчислення відстані між двома містами
def distance(city1, city2):
    x_diff = city1.x - city2.x
    y_diff = city1.y - city2.y
    return math.sqrt(x_diff ** 2 + y_diff ** 2)

# Функція для обчислення загальної відстані маршруту
def calculate_distance(cities, route):
    num_cities = len(route)
    total_distance = sum(distance(cities[route[i]], cities[route[(i + 1) %
num_cities]]) for i in range(num_cities))
    return total_distance

# Функція для генерації випадкових міст
def generate_cities(num_cities):
    cities = []
    for _ in range(num_cities):
        x = random.randint(50, WIDTH - 50)
        y = random.randint(50, HEIGHT - 50)
        cities.append(City(x, y))
    return cities

# Функція для візуалізації маршруту
def draw_route(screen, cities, route):
    for i in range(len(route)):
        city1 = cities[route[i]]
        city2 = cities[route[(i + 1) % len(route)]]
        pygame.draw.line(screen, GREEN, (city1.x, city1.y), (city2.x,
city2.y), 2)

# Функція для охолодження
def cool(temperature, cooling_rate):
    return temperature * cooling_rate
```

```
# Функція для виконання методу відпалу
def simulated_annealing(cities, initial_route, temperature, cooling_rate,
screen):
    current_route = initial_route.copy()
    current_distance = calculate_distance(cities, current_route)

    best_route = current_route.copy()
    best_distance = current_distance

    while temperature > 1:
        new_route = current_route.copy()
        i = random.randint(0, len(new_route) - 1)
        j = random.randint(0, len(new_route) - 1)
        new_route[i], new_route[j] = new_route[j], new_route[i]

        new_distance = calculate_distance(cities, new_route)

        if new_distance < current_distance or random.random() <
math.exp((current_distance - new_distance) / temperature):
            current_route = new_route
            current_distance = new_distance

        if current_distance < best_distance:
            best_route = current_route
            best_distance = current_distance

        temperature = cool(temperature, cooling_rate)

        screen.fill(BLACK)

        # Відображення міст
        for city in cities:
            if city == cities[0]:
                city.draw(screen, RED)
            elif city == cities[-1]:
                city.draw(screen, BLUE)
            else:
                city.draw(screen, WHITE)

        # Відображення поточного маршруту
        draw_route(screen, cities, current_route)

        # Виведення енергії поточного маршруту та оптимального маршруту
        font = pygame.font.Font(None, 20)
        current_distance_text = font.render("Current Distance:
{:.2f}".format(current_distance), True, WHITE)
        best_distance_text = font.render("Best Distance:
{:.2f}".format(best_distance), True, WHITE)
        screen.blit(current_distance_text, (10, 10))
        screen.blit(best_distance_text, (10, 30))

        pygame.display.flip()

    return best_route, best_distance

# Головна функція
def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Travelling Salesman Problem")

    clock = pygame.time.Clock()
```

```
    cities = generate_cities(100)
    num_cities = len(cities)

    initial_route = list(range(num_cities))
    random.shuffle(initial_route)

    best_route, best_distance = simulated_annealing(cities, initial_route,
10000, 0.999, screen)

    print("Optimal Route:", best_route)
    print("Optimal Distance:", best_distance)

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
                pygame.quit()
                sys.exit()

        screen.fill(BLACK)

        # Відображення міст
        for city in cities:
            if city == cities[0]:
                city.draw(screen, RED)
            elif city == cities[-1]:
                city.draw(screen, BLUE)
            else:
                city.draw(screen, WHITE)

        # Відображення найкращого маршруту
        draw_route(screen, cities, best_route)

        # Виведення енергії поточного маршруту та оптимального маршруту
        font = pygame.font.Font(None, 20)
        best_distance_text = font.render("Optimal Distance:
{:.2f}".format(best_distance), True, WHITE)
        screen.blit(best_distance_text, (10, 10))

        pygame.display.flip()
        clock.tick(60)

if __name__ == "__main__":
    main()
```

Код програми, що моделює пожежу КА

```
import pygame
import random

# Константи
WIDTH = 800
HEIGHT = 600
CELL_SIZE = 10
GRID_WIDTH = WIDTH // CELL_SIZE
GRID_HEIGHT = HEIGHT // CELL_SIZE
FIRE_CHANCE = 0.01  # Ймовірність спалаху пожежі в кожній клітині

# Кольори
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
```

```
RED = (255, 0, 0)

# Ініціалізація Pygame
pygame.init()

# Створення вікна
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Моделювання пожежі")

# Створення початкового стану клітин
grid = [[False] * GRID_HEIGHT for _ in range(GRID_WIDTH)]

# Встановлення початкового положення пожежі
fire_x = random.randint(0, GRID_WIDTH - 1)
fire_y = random.randint(0, GRID_HEIGHT - 1)
grid[fire_x][fire_y] = True

# Функція для оновлення стану клітинного автомата
def update_cells():
    new_grid = [[False] * GRID_HEIGHT for _ in range(GRID_WIDTH)]
    for x in range(GRID_WIDTH):
        for y in range(GRID_HEIGHT):
            # Перевірка чи пожежа вже є в поточній клітині
            if grid[x][y]:
                new_grid[x][y] = True
                # Розповсюдження пожежі на сусідні клітини
                for i in range(-1, 2):
                    for j in range(-1, 2):
                        if i == 0 and j == 0:
                            continue
                        if x + i >= 0 and x + i < GRID_WIDTH and y + j >= 0
and y + j < GRID_HEIGHT:
                            # Ймовірність поширення пожежі в сусідню клітину
                            if random.random() < FIRE_CHANCE:
                                new_grid[x + i][y + j] = True
    return new_grid

# Функція для візуалізації стану клітин
def draw_cells():
    for x in range(GRID_WIDTH):
        for y in range(GRID_HEIGHT):
            color = RED if grid[x][y] else WHITE
            pygame.draw.rect(screen, color, (x * CELL_SIZE, y * CELL_SIZE,
CELL_SIZE, CELL_SIZE))

# Основний цикл програми
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Оновлення стану клітин
    grid = update_cells()

    # Очищення екрана
    screen.fill(BLACK)

    # Візуалізація стану клітин
    draw_cells()

    # Оновлення вікна
    pygame.display.flip()
```

```
# Завершення роботи Pygame
pygame.quit()
```