Київський національний університет імені Тараса Шевченка
Факультет радіофізики, електроніки та комп'ютерних систем
Кафедра комп'ютерної інженерії

ЗАСТОСУВАННЯ КЛАСТЕРИЗАЦІЇ ДО ЗАДАЧІ ОПТИМІЗАЦІЇ РОЗВЕЗЕННЯ АВТОМОБІЛІВ ПО ДИЛЕРСЬКИХ ЦЕНТРАХ

Дипломна робота магістра
студента 2 року навчання
спеціальність: 123 «Комп'ютерна інженерія»
Михайлюка Олександра Олександровича

Науковий керівник:
к. ф.-м. н., асистент кафедри математики і
теоретичної радіофізики,
Іваненко Дмитро Олександрович

Рецензент:
Професор кафедри алгебри механіко-
математичного факультету КНУ
Шевченко Георгій Михайлович

До захисту допускаю: Завідувач кафедрою:

Юрій БОЙКО /

Ухвалено на засіданні кафедри "____" _____ 2022 р., протокол № ___

Київ - 2022

РЕФЕРАТ

Обсяг роботи за об’ємом складає 60 сторінок, містить 8 рисунків, та 7 додатків; використано 31 інформаційне джерело.

VRP, ЗАДАЧА МАРШРУТИЗАЦІЇ ТРАНСПОРТУ, АЛГОРИТМ ЗБЕРЕЖЕННЯ, АЛГОРИТМ СИМУЛЯЦІЇ ВІДПАЛУ, ЖАДІБНИЙ АЛГОРИТМ, C#, .NET.

Актуальність роботи полягає в необхідності доставки логістичними компаніями автомобілів в дилерські центри за найкоротші проміжки часу та з мінімальними затратами на сам процес доставки. Адже чим краще виконуються дані умови тим краща репутація у компанії, тим більший прибуток вона отримує.

Об’єктом даної роботи є автоматизація процесу пошуку оптимальних маршрутів. Предметом роботи є розроблений програмний засіб для вирішення ЗМТ.

Метою дослідження є дослідження та розробка алгоритму(одного або декількох), що дозволяють прокласти оптимальний маршрут доставки автомобілів за допустимий час.

Інструменти розробки:

- Microsoft Visual Studio – редактор коду з великою кількістю інструментів для тестування та налагодження програм
- Мова програмування: C#

Результат роботи: реалізація та аналіз результатів одного або декількох алгоритмів, що дають оптимальне рішення задачі VRP. Розроблені в ході роботи алгоритми можна використовувати в більш загальному контексті для

задач маршрутизації транспорту з аналогічними умовами, або на задачах зі схожою основою.

В даній роботі приведено порівняння підходів, інструментів реалізації, структур даних і аналіз щодо їх придатності для виконання поставленої задачі максимально ефективно. Також реалізовано алгоритми маршрутизації транспорту шляхом кластеризації дилерів, детально висвітлено логіку їх роботи та проаналізовано отримані рішення в результаті роботи алгоритмів. Запропоновано можливі варіанти покращення логіки для підвищення ефективності розроблених методів.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ТЗ – транспортний засіб

ООП – об’єктно-орієнтовне програмування

ЗМТ – задача маршрутизації транспорту

VRP – vehicle routing problem

SA – simulate annealing

HP – Holmes and Parker algorithm

CW – Clarke-Wright algorithm

ВСТУП

Транспортна логістика – система, що відповідає за організацію доставки тих чи інших ресурсів кінцевим користувачам. Оптимальним вважається маршрут, яким можна доставити логістичні об'єкти за найкоротший часовий проміжок, задіявши при цьому мінімальну кількість ТЗ, подолавши найкоротший шлях, а також завдавши мінімальної шкоди об'єкту доставки.

Шкодою для об'єкту доставки можна вважати негативний вплив на сам б'єкт, якого він зазнає під час транспортування, так зі сторони часового фактору при доставці об'єктів, що потрапляють в дану категорію.

Ознайомившись із цілями і функціями транспортної логістики, можна визначними основні задачі, які вирішує даний розділ логістики[1]:

- Аналіз пунктів доставки, властивостей вантажу і побудову попереднього маршруту
- Вибір відповідного виду транспорту
- Вибір перевізника і логістичних партнерів
- Побудова маршруту зі всіма ключовими точками
- Контроль вантажу під час транспортування
- Оптимізація показників транспортування

В даній роботі робиться акцент саме на побудові маршруту. На цьому етапі підбирається декілька детальних варіантів маршруту із уточненням виду транспорту, точок тимчасового зберігання і, якщо потрібно, відвантаження на інший тип транспорту. При цьому враховуються можливі ризики та затримки. Різні варіації маршруту відрізняються вартістю, термінами та затримками. Як правило перевага надається маршруту з

мінімальними затримками. Дані процеси описуються і вирішуються за допомогою задачі маршрутизації транспорту(ЗМТ, англ. VRP).

ЗМТ є широко відомим напрямком досліджень в області комбінаторної оптимізації, одним із найбільш важливих класів задач транспортної логістики і знаходить застосування практично у всіх областях транспортування: від роботи поштових служб до обробки багажу в аеропорту. Ціллю задач даного класу зазвичай є мінімізація вартості, відстані чи часу, пов'язаних з транспортуванням, за рахунок визначення оптимальної послідовності відвідування клієнтів для парку транспортних засобів(ТЗ), розміщених в умовному депо. Розробка і застосування методів системного аналізу, керування і обробки інформації для автоматизації рішення VRP вважається ефективним способом економії ресурсів виробництв. Підвищений інтерес до VRP визваний одночасно практичною значимістю і значною складністю – більшість задач даного класу мають складність NP.

Першовідкривачами даної задачі вважаються Г. Данциг і Дж. Рамсер. В 1959 році вони запропонували математичну постановку і алгоритмічний підхід до вирішення практичної задачі доставки бензину від кінцевої станції магістрального трубопроводу до великої кількості терміналів обслуговування. Через кілька років Кларк і Райт вперше додали використання більш, ніж одного ТЗ в постановку задачі, а також запропонували більш ефективну евристику на основі жадібного алгоритму. З того часу було запропоновано велику кількість моделей і алгоритмів, присвячених пошуку точного і наближеного вирішення багатьох варіантів даної задачі, в кінцевому результаті об'єднаних в загальну групу задач маршрутизації транспорту (Vehicle Routing Problem, VRP).

РОЗДІЛ 1.АНАЛІЗ ЗАДАЧІ МАРШРУТИЗАЦІЇ ТРАНСПОРТУ

1.1 Постановка класичної задачі маршрутизації транспорту

Класична VRP – задача комбінаторної оптимізації, в якій для парку однотипних транспортних засобів потрібно визначити оптимальний набір замкнутих маршрутів від єдиного депо до множини віддалених клієнтів. На практиці критерій оптимальності може виражатися будь-якими затратами на об’їзд клієнтів, але частіше за все відповідає довжині маршруту. На рис.1 наведено типовий приклад побудови маршрутів для парку із трьох ТЗ при мінімізації загальної довжини маршрутів. По центру малюнку розміщене депо, в якому з самого початку знаходиться парк із декількох однотипних ТЗ, а на деяких відстанях від нього розміщені клієнти, яких потрібно відвідати. Відстані між усіма пунктами вважаються відомими. Оптимальне рішення являє собою набір найкоротших маршрутів для ТЗ через всіх клієнтів із поверненням в депо.

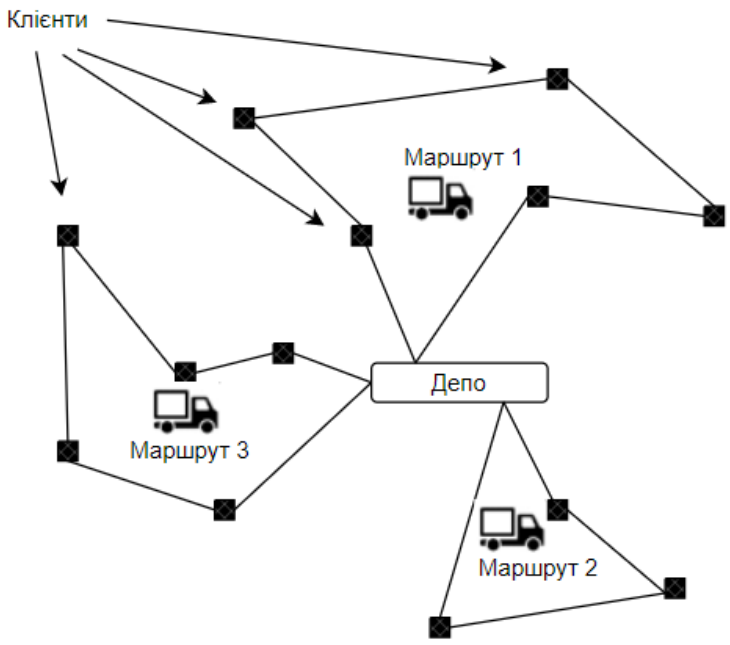

Рисунок 1 - Приклад рішення класичної VRP

Доцільно зазначити, що буд-яка задача класу VRP є узагальненням однієї з найважливіших в області комбінаторної оптимізації задачі комівояжера (Traveling Salesman Problem, TSP), в якій потрібно знайти для комівояжера найбільш вигідний шлях, що проходить через вказані міста по одному разу з майбутнім поверненням в початкове місто. VRP так само, як і TSP, може бути описана в термінах теорії графів, при цьому об'єктами комбінаторної оптимізації стають графові структури.

Класична VRP може бути представлена на графі $G = (V, E)$ з множиною вершин $V = \{0, \ldots, n\}$ і множиною ребер E. Вершини $i = 1,\ldots,n$ відповідають клієнтам, тоді як вершина 0 відповідає депо. Також задається матриця невід'ємних вартостей c_{ij}, пов'язаних із кожним ребром $(i , j) \in E$ і визначаючих витрати на пересування між вершинами i і j.

Таким чином, в теорії графів рішення класичної VRP зводиться до побудови кількох гамільтонових циклів мінімальної довжини на підграфах графа G з однією спільною вершиною-депо. Якщо відсутнє обмеження по вантажопідйомності і довільній кількості ТЗ k задача зводиться до побудови k циклів з загальною вершиною 0, які в сукупності містять всі вершини графа і мінімізують суму вартостей відвідувань клієнтів, тобто задача переростає в множинну задачу комівояжера (TSP з k ізольованими контурами із спільною точкою 0). Подібну задачу можна перетворити в звичайну TSP, додавши k-1 додаткових копій вершини 0 і суміжних з нею ребер.

1.2 Огляд наліз узагальнень і розширень задачі маршрутизації транспорту

В силу своїх обмежень, класична постановка задачі рідко зустрічається на практиці. На даний час існує велика кількість різновидів VRP, більша частина яких є комбінаціями декількох основних розширень класичного

варіанту. Всі вони відрізняються, в основному, комбінацією реальних обмежень, що накладаються на шукане рішення.

Asymmetric VRP (AVRP) – асиметрична задача маршрутизації транспорту. Даний різновид задачі відрізняється від симетричної тим, що моделюється за допомогою орієнтованого графа. Відповідно, матриця коштів є асиметрично. В більшості випадків, розглядається симетричний варіант задачі, тобто з симетричною матрицею кошту. Дане допущення не завжди є доцільним, адже найкоротший шлях між двома точками на множині доріг відрізняється в залежності від напрямку. Найбільш помітні дані відмінності при малих масштабах, наприклад, в межах одного міста. З іншої сторони, також суттєво збільшується простір для пошуку, що призводить до ускладнення пошуку оптимального рішення. Тому, в залежності від конкретної прикладної задачі, кошт(ваги дуг) для протилежних напрямків можуть бути зведені до певного середнього значення.

Capasited VRP (CVRP) – задача маршрутизації транспорту з обмеженням по кількості вантажу. Схожа на класичну, але відрізняється тільки тим, що об’єм вантажу на кожному маршруті не повинен перевищувати задану величину Q, однакову для всіх транспортних засобів. Фіксований парк ТС з однаковою місткістю із загального депо повинен із мінімальними витратами задовільнити попит на товар кожного клієнта і при цьому не перевищити власну вантажопідйомність. Попит і вантажопідйомність задаються цілими або дійсними значеннями. Цей найбільш вивчений варіант задачі маршрутизації транспорту був вперше описаний в роботі Данцинга і Рамсера[3], а в 1979 році з’явилось формулювання у вигляді задачі лінійного програмування[4].

Distance-Constrained VRP (DCVRP) – задача маршрутизації транспорту з обмеженням по відстані[4]. Довжина маршруту не може перевищувати задане значення. Такий варіант зручно використовувати, коли ТС може заправлятись тільки в депо і є необхідність врахувати обмежений об'єм бака. Обмеження по відстані зазвичай розглядається в поєднанні із обмеженням по вантажопідйомності.

VRP with Time Windows (VRPTW) – задача маршрутизації транспорту з часовими вікнами[5]. Використовується при обмеженому часовому діапазоні прийому чи вивозу товару. Для виконання замовлення кожного *i*-го клієнта існує відомий проміжок часу, що визначений інтервалом a_i,b_i. У випадку прибуття раніше нижньої границі інтервалу, враховується час її очікування. Прибуття після верхньої границі інтервалу недопустимо. В деяких випадках обслуговування клієнта у визначеному часовому вікні не є критично важливою умовою, але її порушення додає штрафне значення до цільової функції. Даний різновид задачі носить назву *VRP with Soft Time Windows* (VRPSTW). Також може бути врахований сервісний час, необхідний для обслуговування клієнта. Рішення даного типу задачі дозволяє підібрати час виїзду автотранспорту із депо і тим самим уникнути непотрібних очікувань в точках доставки. Постановка задачі є більш складною, але в деяких випадках більш точно описує реальний процес, адже в більшості практичних задач доставки товарів час прибуття до клієнта і час обслуговування клієнта грають суттєву роль.

Multi-Depot VRP (MDVRP) – задача маршрутизації транспорту з декількома депо[6]. В класичній моделі ЗМТ допускається наявність тільки одного депо, в якому повинні починатись і закінчуватись маршрути всіх транспортних засобів. Проте даний варіант не підходить для ряду випадків, таких, як розподіл продукції декількома постачальниками спільній групі

споживачів. MDVRP є узагальненням класичної ЗМТ, у якій існує більш, аніж одне депо, із яких відбувається обслуговування клієнтів, при цьому ТС починає і закінчує маршрут у власному депо.

Очевидно, що такий варіант задачі є більш складним, оскільки виникає необхідність розподілу споживачів по різних депо. Для цього доводиться визначати, які клієнти відносяться до кожного депо, або використовувати двофазні алгоритми, в яких граф спочатку розбивається на підграфи, а потім будується маршрут окремо для кожного депо. В ідеальному випадку більш ефективно виконання двох кроків одночасно, однак при вирішенні задач з великою кількістю вершин це стає важко.

Periodic VRP (PVRP) – періодична задача маршрутизації транспорту. На відміну від класичної ЗМТ, в задачах з періодичною маршрутизацією використовується розширений період планування до декількох днів. Для різних клієнтів потрібна різна кількість відвідувань у вказаний період, при тому дні обслуговування не визначені заздалегідь, але заданий список можливих дат відвідування для кожного клієнта. Таким чином задача маршрутизації вирішується для кожного дня планування[7]. Для ряду випадків ця особливість має важливе значення, наприклад, при вирішенні проблеми збирання відходів.

Існує також розширення задачі з вибірковим обслуговуванням – PVRP with Service Choice(PVRP-SC), де частота відвідувань клієнтів встановлюється в процесі вирішення задачі. Це дозволяє отримати більш оптимальні маршрути і дає переваги при обслуговуванні клієнтів [7].

VRP with Pickup and Delivery (VRPPD) – задача маршрутизації транспорту з вивезенням і доставкою[8]. Узагальнення задачі із обмеженням вантажопідйомності, в якому клієнти можуть як отримувати, так і

відправляти товари. При цьому, зазвичай, мається на увазі, що товари не перевозяться від одного клієнта до іншого, а з самого початку відправляються з депо, або в кінцевому результаті з'являються в депо.

VRPPD поділяється в залежності від порядку доставки і вивезення, які можуть здійснюватися послідовно, змішано і одночасно:

- Delivery-First, Pickup-Second VRP: всі товари повинні бути доставлені споживачам до того, як відбудеться будь-яке вивезення від постачальників. Таким чином, рішення задачі поділяється на дві фази. З практичної точки зору, дана умова пояснюється тим фактом, що всі завантаження, зазвичай, виконуються позаду транспорту і перестановка вантажів є неприпустимою чи затратною по часу. Водночас, росте ризик перевищення вантажопідйомності ТС, незалежно від рівня попиту на вивезення і доставку. Ця задача має і іншу назву – *VRP with Backhauls* (VRPB) – задача маршрутизації транспорту із зворотнім транзитом(із поверненням товару);
- *Mixed Pickup and Delivery VRP* (VRPMPD): вивезення і доставка можуть виконуватись в будь-якій послідовності по маршруту ТС, однак клієнти так само, як і в попередньому випадку розділені на споживачів та постачальників. Цей варіант дозволяє отримати більш оптимальні маршрути, але ускладнює завантаження і вивантаження товарів.
- *VRP with Simultaneous Pick-up and Delivery* (VRPSPD): одні і ті ж клієнти одночасно виступають як споживачами, так і постачальниками, вертаючи певний товар в депо. На практиці це потрібно, наприклад, в продуктові магазини, коли багаторазові піддони чи контейнери використовуються для транспортування товарів. Обидва останні варіанти можуть бути описані за допомогою загальної моделі:

VRPMPD як VRPSPD, в якій запит на доставку чи вивезення дорівнює нулю, а VRPMPD як VRPSPD, якщо кожного клієнта розбити на отримувача і постачальника.

В розглянутих вище варіантах VRPPD допускалось, що запити на вивезення і доставку відносяться як один-до-багатьох-до-одного (one-to-many-to-one-problem), коли товари транспортуються від депо до клієнтів і в зворотному напрямку. Однак деякі автори також відносять до VRPPD різновиди із зв'язками багато-до-багатьох (many-to-many problem), в яких будь-який клієнт може відправити і отримати будь-який товар, і зі зв'язками один-до-одного (one-to-one problem), в яких встановлюється попарна відповідність між пунктами завантаження і пунктами доставки у вигляді запитів. У першому випадку товар вивозиться від одного із багатьох постачальників і доставляється одному з багатьох споживачів. У другому ж кожен запит визначається пунктом завантаження, відповідним йому пунктом доставки і заданою кількістю вантажу, який потрібно транспортувати між вказаними пунктами. Широко відомими прикладами задач із парними зв'язками є Pickup And Delivery Problem (PDP), що застосовується при перевезенню товарів і Dial-A-Ride Problem (DARP), що застосовується при перевезенні людей. В даних випадках створюються додаткові обмеження і цілі.

Узагальнена задача вивезення і доставки *General Pick-up and Delivery Problem* (GPDP) дозволяє визначити різні задачі з вивезенням і доставкою як окремі випадки її формулювання.

Також варто окремо відмітити варіант VRP із стековим принципом «останнім прийшов – першим вийшов» - *VRP with LIFO* [10]. Аналогічно VRPPD, за винятком додаткового обмеження: розвантаження ТС може

відбуватись тільки в зворотному порядку завантаження, тобто першим завжди вивантажується останній завантажений товар. Зазвичай використовується при мультиноменклатурному вантажі для скорочення часу завантаження і відвантаження ТС, так, як відпадає необхідність переставляти товар.

Split Delivery VRP (SDVRP) – задача маршрутизації транспорту з роздільною доставкою[11]. В задачі знімається загальне для всіх VRP обмеження на багаторазове відвідування клієнта. Тобто один і той же клієнт може обслуговуватись декілька раз кількома ТС, якщо це дозволить зменшити витрати. Дана постановка особливо виправдана, якщо запити клієнтів перевищують вантажопідйомність ТС. Експериментально показано, що в залежності від характеристик задачі може бути отриманий різний виграш в довжині маршрутів при використанні реальної доставки[11]. Під час розгляду задачі, простим підходом є розбиття кожного клієнта на множину близько розміщених з меншими запитами, тим самим задача зводиться до звичайної VRP. Однак, як правило, для задачі маршрутизації з різними видами транспорту отримати рішення складніше, аніж для класичної ЗМТ.

Stochastic VRP (SVRP) – задача маршрутизації транспорту з випадковими даними[12]. Один чи декілька компонентів задачі можуть мати випадкову поведінку:

- *VRP with Stochastic Demands* (VRPSD): запит кожного клієнта пов’язаний з заданим розподілом, замість конкретного значення, а дійсне значення визначається тільки після прибуття ТС;
- *VRP with Stochastic Clients* (VRPSC): множина клієнтів точно не відома, кожен клієнт існує з певною ймовірністю;

- *VRP with Stochastic Travel Time* (VRPST): часи поїздок(відстані) між пунктами не детерміновані;
- *VRP with Stochastic Service Time* (VRPSST): час обслуговування кожного клієнта не детермінований.

SVRP відрізняється від класичної VRP цілим рядом аспектів. Методологія рішення є більш складною і поєднує в собі особливості стохастичного і цілочисельного програмування. Доволі часто задачі такого типу залишається невирішеними[13]. Тільки в деяких випадках вдається отримати оптимальне значення, тому розробка і застосування хороших евристик є непростим завданням. Вирішення SVRP, зазвичай, відбувається в два етапи. Перший дає рішення без врахування випадкових змінних. На другому етапі, коли стають відомими невідомі випадкові значення, відбувається корекція раніше отриманого рішення.

Fuzzy VRP (FVRP) – нечітка задача маршрутизації транспорту[14]. На практиці буває важко отримати точні значення запитів, часу шляху, кількості і місцезнаходжень клієнтів, меж і часових вікон, тощо, якщо вони підпорядковуються законам ймовірності. В деяких нових системах також важко описати параметри задачі, як випадкові величини через недостатню кількість даних для аналізу розподілу. Використання методів теорії нечітких множин дозволяє успішно моделювати задачі, що містять елементи невизначеності і суб'єктивності. Основними різновидами FVRP є:

- *VRP wit Fuzzy Demands* (VRPFD): нечіткий запит клієнтів на товар;
- *VRP with Fuzzy Travel Time* (VRPFT): нечіткий час поїздок(відстані) між пунктами;
- *VRP with Fuzzy Service Time* (VRPSST): нечіткий час обслуговування кожного клієнта;

- *VRP with Fuzzy Time Windows* (VRPFTW): нечіткі межі часових вікон, в межах яких можуть обслуговуватись клієнти.

Dynamic VRP (DVRP) – динамічна задача маршрутизації транспорту[15]. Допускається, що можуть відбуватись деякі зміни параметрів в процесі її вирішення. До таких, зазвичай, відносять аварії ТС, дорожні затори, нові замовлення, непередбачувані виклики, тощо. Найчастіше розглядається випадок, коли нові клієнти можуть з'являтись протягом дня, тобто після того, як ТС покине депо. Таким чином, по мірі обслуговування клієнтів умови задачі змінюються і необхідно динамічно перераховувати маршрути, враховуючи актуальну інформацію. Як правило, заздалегідь відома статистика появи нових замовлень, на основі якої створюються імітаційні моделі для розробки і випробувань алгоритмів.

Open VRP (OVRP) – відкрита задача маршрутизації транспорту[16]. Незамкнутий варіант задачі, в якому немає вимоги повертатися в депо в кінці маршруту. Таким чином, віддаленість останнього відвіданого клієнта від депо не впливає на загальну вартість рішення. Частіше за все OVRP використовується в тому випадку, коли компанія не володіє парком ТС, а підписує контракт з зовнішніми кур'єрами. Відповідно, наймані ТС не повинні повертатися в центр розподілу компанії(депо) і можуть закінчити маршрут в будь-якому місці. Рішення повинно забезпечити мінімальний набір ТС, яких потрібно задіяти для обслуговування всіх клієнтів з мінімальними витратами на дорогу.

Також потрібно відмітити, що зняття типової для всіх VRP умови повернення в депо не спрощує задачу, OVRP також залишається NP-важкою. При цьому є принципова відмінність: в OVPR будуються гамільтонові шляхи, що починаються в депо, тоді, як в класичній VRP – гамільтонові

цикли. Тому алгоритми вирішення замкнутих задач часто є неефективними при вирішенні незамкнутих. Тим не менш, незамкнутий варіант задачі зводиться до замкнутого шляхом зміни вартості дуг, що входять в депо, на 0.

Heterogeneous VRP (HVRP) – задача маршрутизації транспорту з урізноманітненим парком[17]. Узагальнення класичної VRP, в якому клієнти обслуговуються декількома типами ТС з відмінними характеристиками, такими як вантажопідйомність, швидкість, вартість використання, тощо. Як правило, парк ТС в моделях VRP є однорідним, що рідко відповідає реальній практиці. Дане допущення дозволяє спростити пошук рішення, але в логістичних операціях часто потрібно приймати до уваги характеристики кожного конкретного ТС.

Існує три версії задачі з урізноманітненим траснпортом:

- *Fleet Size and Mix VRP* (FSMVRP), *Vehicle Fleet Mix* (VFM) чи *Fleet Size and Composition VRP* (FSCVRP): змішаний парк ТС різних типів, що відрізняються вантажопідйомністю і вартістю використання, при тому кількість доступних ТС кожного типу необмежена;
- *Heterogeneous Fleet VRP* (HFVRP) чи *Mix Fleet VRP* (MFVRP): аналогічна попередній, але для кожного типу вводиться різна вартість пересування(за одиницю часу/відстані), тобто використовується множина матриць вартостей для всіх типів ТС;
- *Heterogeneous Fixed Fleet VRP* (HFFVRP): узагальнення попередніх версій з обмеженням кількості доступних ТС кожного типу (фіксований парк).

VRP with Satellite Facilities (VRPSF) – задача маршрутизації транспорту з додатковими складами[18]. В класичній VRP задачі кожен маршрут починається і закінчується в депо. Однією із причин повернення в депо є

обмежена вантажопідйомність – коли ТС розвозить всі товари, то повинен повернутись в депо за новою частиною товарів. Однак, в деяких випадках вигідніше влаштувати довантаження на маршруті в додаткових проміжних пунктах (додаткових складах) без повернення в депо. Даний варіант застосовується в ситуаціях, коли центральний постачальник повинен забезпечити товаром велику кількість споживачів на регулярній основі. VRPSF є доповнюваною частиною задачі розподілу товару – *Inventory Routing Problem* (IRP).

Розглянуті різновиди VRP для зручності можна піднести у вигляді ієрархічної схеми (рис. 2). З неї видно, що в основі більшості різновидів VRP є базове обмеження по вантажопідйомності(CVRP), яке в явному виді може не вказуватись при постановці. Очевидно, що за рахунок комбінацій розглянутих вище різновидів VRP можна отримати велику кількість варіантів і постановок задачі.

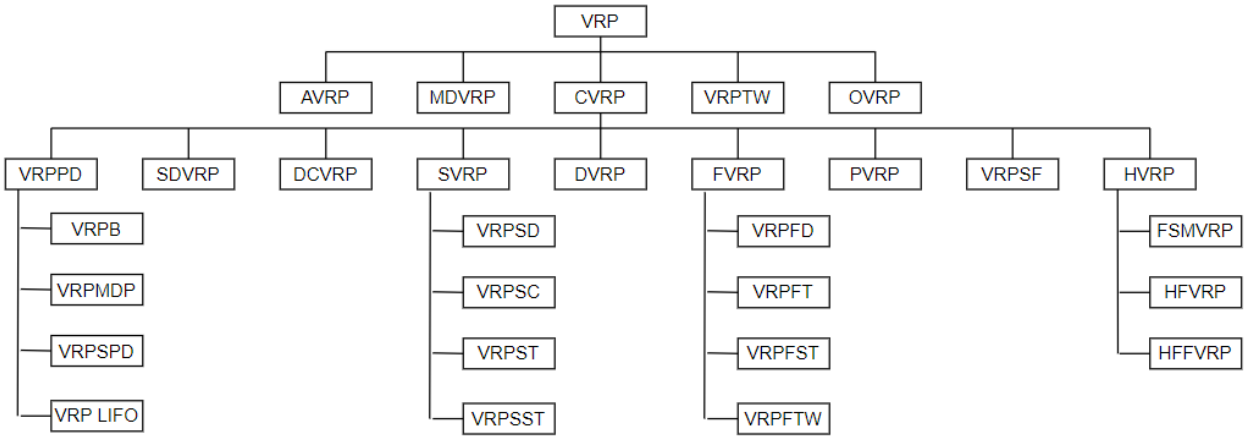

Рисунок 2 – ієрархічна схема узагальнень і розширень VRP

1.3 Підходи до вирішення ЗМТ

ЗМТ є задачею комбінаторної оптимізації, в якій число допустимих маршрутів росте експоненційно при збільшенні числа клієнтів і належить до класу складності NP. Це означає, що повний перебір можливих її рішень хоча

й дозволяє знайти оптимальний маршрут, але потребує колосального(неприйнятного) часу обчислень, навіть при відносно невеликій розмірності задачі (14 і більше). Оскільки повний перебір неефективний, то було придумано велику кількість алгоритмів, які, в цілому, можна виділити в три групи:

- точні
- евристичні
- метаевристичні.

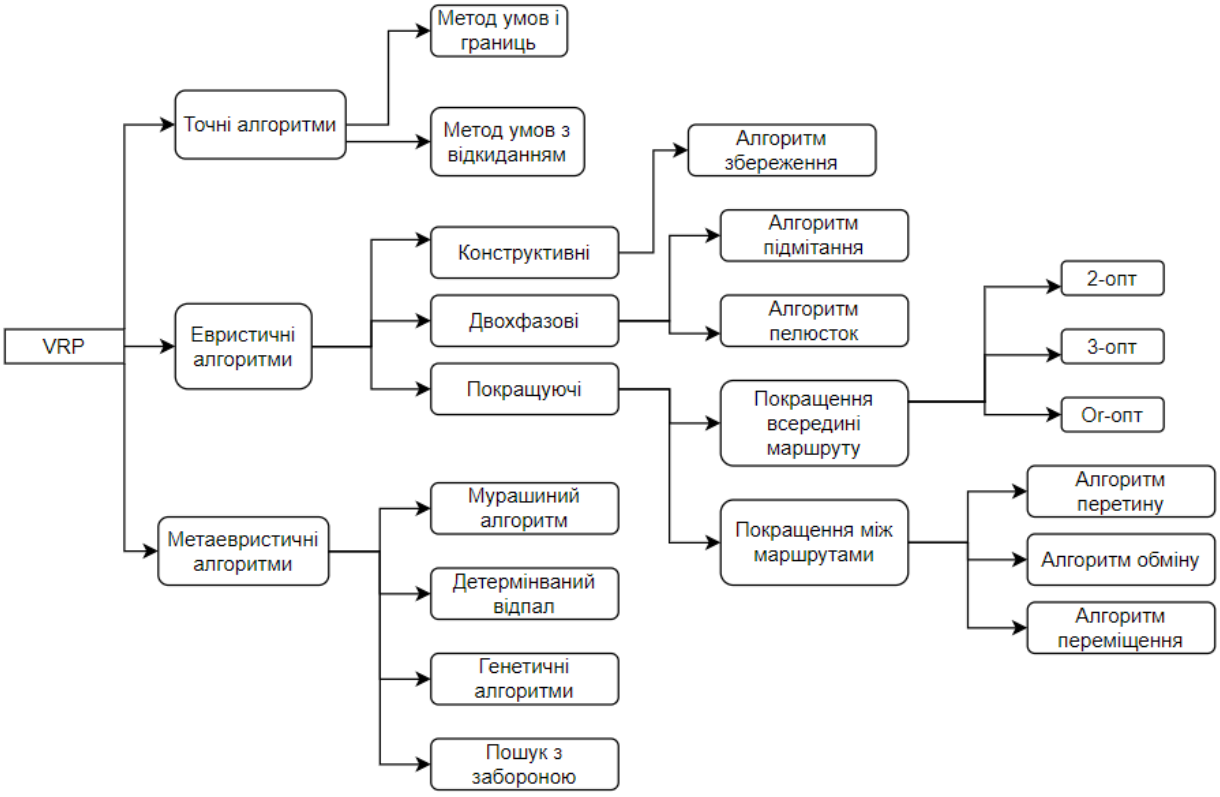

Рисунок 3 –класифікація алгоритмів ЗМТ

На рис.3 наведено класифікацію основних алгоритмів, які застосовуються до вирішення ЗМТ в більшості випадків. Оскільки існує велика кількість алгоритмів, то доцільно було б розглянути лише декілька з них, висвітливши кілька з кожної групи.

1.3.1 Точні алгоритми

Дані алгоритми перебирають всі можливі комбінації, поки не буде отримано найкращий з варіантів. Задача відноситься до NP-складної, тому точні алгоритми можуть бути застосовані тільки для випадків з малою кількістю вхідних даних. Таким чином, вони не підходять для вирішення реальних задач.

Метод гілок і меж[19] є одним з найбільш поширених в області дискретної оптимізації. Метод перебирає будує дерево рішень та дозволяє знайти глобальний мінімум на множині всіх можливих комбінацій. Спеціалізовані його варіації створюються для вирішення різних комбінаторних задач.

Метод гілок та меж складається з наступних кроків для визначення оптимального цілочисельного рішення:

1) знаходження оптимального рішення моделі лінійного програмування з релаксацією цілих чисел;
2) на першому вузлі нехай релаксоване рішення є верхньою межею, а заокруглене цілочисельне рішення – нижньою;
3) вибір змінною з найбільшою дробовою частиною для розділення. Створення двох умов для цієї змінної, що відображають значення секційного цілого. Результатом буде нове ≤ обмеження і нове ≥;
4) створення двох нових вузлів: один для ≤ обмеження і один для ≥;
5) вирішення вільної моделі лінійного програмування з новим обмеженням, доданим в кожному з цих вузлів;
6) релаксованим рішенням є верхня межа на кожному вузлі, а відповідне максимальне цілочисельне рішення(на будь-якому вузлі) є нижньою границею;

7) 7) якщо процес створює допустиме цілочисельне рішення з найбільшим значенням верхньої межі будь-якого кінцевого вузла, то досягається оптимальне цілочисельне рішення. Якщо можливе рішення не виникає, то відбувається перехід до вузла з найбільшою верхньою межею;
8) Повернення до кроку 3.

Для моделі мінімізації релаксовані рішення округлюються, а верхня і нижня межі змінюються на протилежні.

Ще одним точним методом є метод віток з відсівом[20]. В основі методу лежить наступна ідея: якщо нижня межа значень функції на підплощині *A* дерева пошуку більша, ніж верхня межа на будь-якій раніше переглянутій підплощині *B*, то *A* може бути виключена з майбутнього перегляду(правило відсіву). Зазвичай, мінімальну з отриманих верхніх оцінок записують в глобальну змінну *m*. Будь-який вузол дерева пошуку, нижня границя якого більше значення m, може бути виключений з множини подальшого розгляду.

Якщо нижня межа вузла дерева співпадає з верхньою, то це значення є мінімумом функції і досягається на відповідній підплощині.

Однак час роботи даних алгоритмів все одно росте надто швидко, а в гіршому випадку – як і при повному переборі.

1.3.2 Евристичні алгоритми

Більш розумним рішенням є пошук в напрямку наближених алгоритмів. Відомо достатньо шляхів вирішення VRP, більша частина яких – евристичні методи. Відомі підходи, зазвичай, орієнтуються на загальне формулювання VRP, в якій пропонується симетрична чи асиметрична

матриця відстаней, нечітко задана кількість ТЗ і відслідковується тільки обмеження по їх вантажопідйомності чи максимальній довжині маршруту.

Евристичні алгоритми – це алгоритми, які засновуються на деякому правилі(евристиці), що не завжди слідує із строгих математичних принципів; в більшості випадків видають рішення, близьке до точного. Тобто, жертвуючи точністю, можна значно скоротити час пошуку оптимального рішення. В спробах підвищити точність рішень, отриманих евристикою та наблизити її до точності рішень повного перебору було придумано багато евристичних алгоритмів. Вони поділяються на:

1) двохфазові алгоритми;
2) конструктивні алгоритми;
3) покращуючі алгоритми.

Конструктивні алгоритми вибудовують рішення крок за кроком, вираховуючи загальну вартість, що отримується в ході рішення, але не мають фази подальшого покращення.

Одним із найвідоміших та найефективніших конструктивних алгоритмів є алгоритм збереження(Clarke-Wright)[21], який запропонували двоє вчених, додавши до постановки задачі VRP можливість використовувати більш, ніж один ТЗ. Кроки, з яких складається алгоритм можна описати наступним чином

1) зробити n маршрутів: $v_0 \rightarrow v_1 \rightarrow v_0$, для кожного $i \geq 1$;
2) вирахувати збереження для злиття вершин доставки i та j, яке задається формулою $s_{ij} = d_{i0} + d_{0j} - d_{ij}$, для всіх $i,j \geq 1$, $i \neq j$;
3) відсортувати в порядку спадання збереження;

4) починаючи з верхнього(із тих, що залишились) маршруту між двома точками, об’єднуються зв’язані маршрути із найбільшим збереженням при умові, що:
   - дві точки вершини не знаходяться на одному і тому ж маршруті;
   - жодна точка доставки не є внутрішньою по відношенню до її маршруту, тобто обидві точки пов’язані з депо на відповідних маршрутах.
   - не порушуються накладені умови при об’єднанні маршрутів (накладені умови варіюються відносно варіанту задачі та описані у п. 1.3)
5) крок 3 повторюється поки не буде отримано множину маршрутів з максимальним збереженням.

До двохфазових алгоритмів належать алгоритми, які розбиваються на дві операції: групування вершин для кожного майбутнього маршруту і вирішення TSP для кожної отриманої вершини.

Представниками даного класу є алгоритм Фішера та Джайкумара [21], алгоритм пелюсток (Petal Algorithm) [22] та підмітання (Sweep Algorithm) [24].

Під час роботи алгоритму пелюсток створюються декілька маршрутів, що називаються пелюстками. А потім відбувається кінцевий вибір шляхом вирішення задачі розбиття форми.

Мінімізується

$$k \in S^{d_k X_k}, \quad (1)$$

при умові

$$k \in S^{d_k X_k} = 1, \quad (2)$$

де $i = 1, \ldots, n$, $x_k = 0$ or 1,

$k \in S$, S – множина маршрутів,

$x_k = 1$ тоді і тільки тоді, коли маршрут k належить рішенню,

a_{ik} – двійковий параметр, рівний 1, тільки якщо вершина i належить маршруту k, а d_k – вартість пелюстки k. Якщо маршрути відповідають суміжним секторам вершин, то ця проблема має властивість округлення і може бути вирішена за поліноміальний час.

Алгоритм підмітання є концептуально простим: в традиційних задачах ЗМТ вузли оточують центральне депо, то радіальна лінія з центральним депо в якось центральної точки починається на 0^0 і проходить по вузлах, оточуючих депо за часовою стрілкою, або проти. Коли зустрічається перший вузол, то перевіряється на обмеження пропускної здатності та присвоюється транспортному засобу. Якщо він підходить, то позначається, як відвіданий. Таким чином лінія прокручується до тих пір, поки ємність транспортного засобу не буде повною, або деякі обмеження не будуть перевищені. Після цього процедура повторюється для всіх наступних транспортних засобів, поки всі вершини не будуть відмічені, як відвідані.

Ідея покращуючих алгоритмів полягає в тому, що спочатку відбувається формування допустимого рішення(будь-якого), а потім покращення його шляхом послідовних невеликих змін. Покращуючі алгоритми всередині маршруту описані в термінах λ-опт операцій. В даних алгоритмах видаляються λ ребр із маршруту і відбувається поєднання λ сегментів, що залишились у всіх комбінаціях. Якщо перше покращуюче з'єднання знайдене, то нові зміни заносяться в маршрут. Якщо неможливо знайти більш вдалі варіанти замін, то робота завершується. Для перевірки λ-оптимальності потрібно часу $O(n^{\lambda})$.

Суть алгоритму 2-опт полягає в тому, щоб пройти маршрут, який себе пересікає, та впорядкувати його так, щоб маршрут був прямим, тобто себе не пересікав.

У процесі роботи 3-опт алгоритму відбувається видалення 3-х з'єднань в маршруті для створення 3 суб-маршрутів. Потім аналізується 7 різних способів повторного під'єднання маршруту для пошуку оптимального. Далі цей процес повторюється для іншого набору із 3 з'єднань, поки всі можливі комбінації не будуть перевірені. Він має складність $O(n^3)$.

Інші покращуючі алгоритми коротко описуються в [25].

1.3.3 Метаевристичні алгоритми

Метаевристика – метод вирішення задач шляхом комбінування існуючих процедур з відкритим інтерфейсом і закритою реалізацією, що призводить до максимально ефективного рішення. Особливістю метаевристичних алгоритмів є те, що вони не дають точного опису порядку дій для вирішення задачі, і кожен з них повинен бути додатково конкретизований шляхом підбору значень керуючих параметрів. На відміну від традиційних алгоритмів оптимізації і ітераційних методів, метаевристичні не гарантують, що в глобальному масштабі оптимальне рішення буде знайдене для деякого класу задач в силу стохастичності. Однак при пошуку на великих наборах допустимих рішень, алгоритми даного класу часто дозволяють знайти хороше рішення з меншими часовими та ресурсними витратами. Метаевристичні алгоритми не піддаються комбінаторному вибуху - явищу, коли час обчислення, необхідний для знаходження оптимального рішення NP-важких задач, зростає експоненційно.

Розробка та аналіз одного з таких алгоритмів, а саме метод симуляції відпалу буде розглянуто в наступній частині даної роботи.

В основі симуляції відпалу лежить процес кристалізації речовини для підвищення однорідності металу. У кожного металу є кристалічна решітка, яка описує геометричне розміщення атомів речовини. В контексті даного алгоритму, сукупність всіх атомів можна назвати станом системи. Кожному стану системи відповідає той чи інший рівень енергії. Ціллю даного методу є приведення системи в стан з найменшою енергією. Адже, чим нижче рівень енергії, тим «краща» кристалічна решітка, тим менше у неї дефектів і метал є більш стійким. В ході «відпалу» метал нагрівають до певної температури і починають повільне, контрольоване охолодження. Атоми кристалічної решітки покидають свої позиції та намагаються потрапити в стан з найменшою енергією. При цьому, з певною ймовірністю вони можуть перейти в стан із більшою. Перехід в гірший стан часто дозволяє віднайти стан з меншою енергією, ніж початкова. Процес закінчується, коли температура падає до попередньо заданого значення**[26].**

Покроково роботу алгоритму можна описати наступним чином:

1) на вході задається t_{max} (кінцева температура) та t_{min} (початкова)
2) генерується перше довільне рішення *s1*
3) $t_1 = t_{max}$
4) поки виконується умова $t_i > t_{min}$:
    - вираховується нове рішення на базі попереднього
    - для нового рішення вираховується кошт та різниця з попереднім
    - якщо різниця негативна, то нове рішення стає кращим

- якщо різниця позитивна, то присвоєння нового рішення в якості кращого відбувається з ймовірністю $P(\Delta E) = e^{-\Delta E/ti}$
- температура знижується на коефіцієнт α

5) В якості результату повертається останній стан.

Ймовірність, описана вище не є випадковою та слідує із законів термодинаміки**[26]**.

Тобто даний алгоритм займається пошуком глобального мінімуму(у випадку VRP). Водночас, він допускає, що локальний мінімум не є глобальним. На відміну він методу градієнтного спуску, симуляція відпалу з певною ймовірністю дозволяє розглядати гірші значення, припускаючи, що це приведе до кращого результату, ніж попередній. У ході роботи алгоритму, дана ймовірність поступово зменшується, поки не досягне критичного значення(аналогічно температурі під час процесу кристалізації).

Перепоною до практичної реалізації більшості метаевристичних алгоритмів є або їх абстрактний опис (просте перечислення найменувань етапів алгоритму), або опис, орієнтований на вирішення тільки однієї певної проблеми (наприклад, оптимізації числових функцій, або комбінаторної).

РОЗДІЛ 2. ОГЛЯД, АНАЛІЗ ТА ВИБІР ІНСТРУМЕНТІВ ДЛЯ РЕАЛІЗАЦІЇ ЗАДАЧІ

2.1 Інструменти для реалізації та вимоги до рішення.

На даний момент існує велика кількість мов програмування, як із узагальненими концепціями, так і адаптованих під специфічні потреби, тому вибір дуже великий. Уже довгий час можна спостерігати тенденції до використання мови програмування Python для аналізу даних, написання алгоритмів, тощо. Мова з інтерпретатором, динамічною типізацією та зручним синтаксисом з певними особливостями(табуляція замість дужок) є легкою в освоєнні, але повільнішою, порівняно з C#, C++, JAVA[27]. Особливості мови програмування Python, що пояснюють його повільнішу роботу у порівнянні з мовами, які використовують компілятори:

- GIL(Global Interpreter Lock, глобальне блокування інтерпретатора)
- Специфікація інтерпретованої мови
- Динамічна типізація

Також Python потребує багато ресурсів, адже під більшість типів даних з самого початку виділяє більше пам’яті, ніж потрібно. Пов’язано це з тим, що Python розглядає всі тип даних, як об’єкти. Наприклад, стандартний тип int займає 24 байти – в три рази більше, в порівнянні з int64_t з мовою C[27]. Числа з плаваючою комою в Python залежать від реалізації, але схожі на числа з подвійною точністю в C. Однак вони займають також в тричі більше, аніж еквівалентні типи в C. Таким чином вибір було зроблено в сторону C-подібних мов, а саме C#. Дана мова є об’єктно-орієнтованою, компільованою, має статичну типізацію. Завдяки розробникам Microsoft, вона є частиною великої екосистеми і пропонує велику кількість можливостей без встановлення додаткових пакетів\програм, а використання сучасної відкритої платформи .NET(раніше відомої, як .NET Core) робить

виконання коду програм на C# незалежним від операційної системи. Ще одним вагомим аргументом в сторону вибору C# була необхідність використання розробленого алгоритму у веб-застосунку на платформі .NET. У випадку з Python для цього можна було б використати реалізацію, орієнтовану на виконання в середовищі .NET(наприклад, IronPython), але це б наклало свій слід на швидкодію. Також, на даний момент, остання версія Python, яка підтримується IronPython – 2.7 в той час, коли актуальною є версія 3.10.4.

Даний алгоритм написаний на C#10.0 та з використанням платформи .NET 6.0 – останніх версій, актуальних на момент написання алгоритму. Шоста версія платформи пропонує декілька важливих функцій, таких, як покращена попередня компіляція, оптимізація на основі профілювання(PGO), та гаряче перезавантаження під час відлагодження, що значно скорочує час розробки та тестування написаного коду[28]. Також, у вище вказаному джерелі показано, що .NET 6 серії має значно менший час компіляції рішення, порівняно з попереднім випуском.

Для зручності розробки на C# використовувалась Microsoft Visual Studio 2022 – повноцінне інтегроване середовище розробки від Microsoft, що включає в себе компілятор, редактор коду з підтримкою технології IntelliSense і можливістю автоматичного рефакторингу коду на базовому рівні. Кінцеве рішення – композиція кількох бібліотек класів з розробленими алгоритмами та консольний застосунок для демонстрації ефективності того чи іншого алгоритму на тих чи наборах даних.

Щодо вимог до алгоритму, то вони наступні:

- час виконання – не більше 20 хвилин

- похибка у результаті не повинна перевищувати 5% від точного результату(отриманого методом повного перебору)
- алгоритм повинен вирішувати симетричну CVRP – задачу з обмеженням по вантажопідйомності транспорту, який доставляє дилерам автомобілі та має симетричну матрицю відстаней між дилерами.
- матриця відстаней є повною, тобто в кожного дилера прокладені маршрути до всіх інших.

Реалізації алгоритмів та аналіз їх ефективності описані в наступних частинах даної роботи.

2.2. Організація проекту

Для ефективної перевірки роботи алгоритму потрібна певна кількість наборів даних. Можна, звісно, згенерувати декілька невеликих наборів даних, але для якісного результату дослідження, рішення потрібно було протестувати на якомога більшій кількості наборів даних різної величини. Одним із джерел з даними для тестування була CVRP-бібліотека[29], що налічувала велику кількість варіантів вхідних даних задачі. Також дані були запозичені з інших відкритих ресурсів. Проблема полягала в тому, що дані з різних джерел мали своє форматування. Також дані, створені вручну, теж мали своє форматування, що не дозволяло уніфіковано зчитувати дані з різних ресурсів. Для прикладу, набір із одного з відкритого джерела був у звичайному форматі txt, мав табульоване форматування та секції, розділені ключовими словами. Приклад такого екземпляру даних наведено в додатку А. В той же час, ще один сет був у форматі xml а дані, створені власноруч під час розробки алгоритму зберігалися у json. Оскільки для зручності використання розробленого алгоритму потрібен уніфікований підхід до отримання вхідних даних, то було прийнято рішення написати «екстрактори»

для тих чи інших наборів. Екстрактором в даному контексті називається модуль, який зчитує дані та перетворює їх в унверсальний формат для подальшого використання алгоритмом VRP. Кожен з екстракторів реалізує інтерфейс IDataExtractor з бібліотеки VRP.Core.Common. Екстрактор повинен реалізувати метод Extract, що приймає стрім(для можливості отримувати дані не тільки з файлових джерел) та повертає результат у вигляді екземпляру класу InputData, який в свою чергу є вхідними даними для VRP алгоритму. Одна з реалізацій даного інтерфейсу наведена в додатку Б. В ході розробки даного рішення реалізовано три екстрактори:

- для специфічного формату json
- для текстового форматування з однієї з CVRP-бібліотек
- для формату xml(ще однієї з бібліотек)

Оскільки в даній роботі висвітлено дослідження кількох алгоритмів, то для зручності було створено інтерфейс IVRPAlgorithm(бібліотека VRP.Core.Common), що приймає у якості вхідних даних екземпляр класу InputData та оптимальний список маршрутів(відносно своєї логіки) для того чи іншого набору даних. Для кожного з підходів виділена своя бібліотека класів. У них знаходяться додаткові сутності, методи, потрібні для роботи відповідного алгоритму. Таким чином структура проекту наступна:

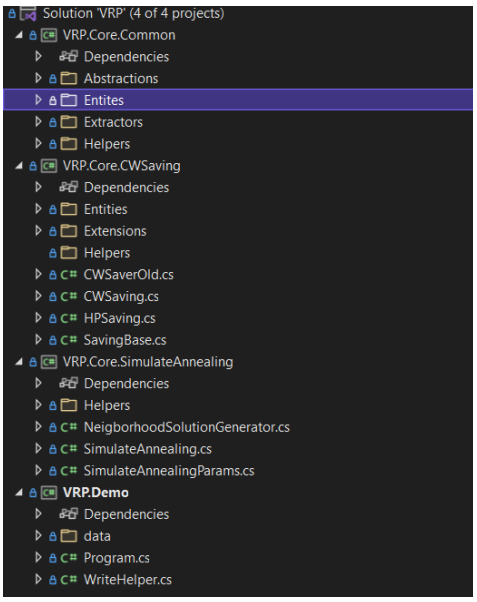

Рисунок 4 - Структура проекту вирішення задачі VRP

VRP.Demo – консольний застосунок для демонстрації роботи алгоритмів, VRP.Core.CWSaving та VRP.Core.SimulateAnnealing – бібліотеки в яких реалізовано рішення VRP за допомогою збереження та симуляції відпалу відповідно.

2.3 Алгоритм збереження (Clarke and Wright)

Першим алгоритмом для дослідження було обрано евристичний алгоритм збереження(алгоритм Кларка-Райта[21]). Це було зроблено з наступних міркувань:

- на відміну від точних алгоритмів, даний евристичний алгоритм є істотно швидшим
- на відміну від метаевристичних алгоритмів, даний алгоритм є абсолютно кертованим, тобто можна повністю проаналізувати хід його виконання.

Ідея алгоритму описана в п.1.3.2 теоретичної частини, в даному ж розділі буде розглянуто його реалізація. Код фінальної реалізації алгоритму збереження представлений в додатку В. Оскільки на вхід алгоритму

подається матриця коштів для кожного маршруту – доволі просто отримати матрицю відстаней. Оскільки алгоритм жадібний, то матрицю відстаней потрібно відсортувати в спадаючому порядку(в першій реалізації матриця була відсортована довільно, через що алгоритм відпрацьовував некоректно). А далі, ітеруючи матрицю збереження слід дотримуватись наступних правил:

- якщо перший дилер і другий не в кластері, і якщо місткість ТЗ охопить їх потреби, то перемістити дилерів в один кластер, якщо ж місткість ТЗ не дозволяє – то в два окремі.
- якщо один з дилерів уже в кластері, то в першу чергу потрібно перевірити чи він не є транзитним. Якщо він на початку, або в кінці маршруту, то знову потрібно перевірити обмеження на місткість ТЗ. В оптимістичному випадку вільний дилер додається в існуючий кластер, в негативному – для нього створюється окремий кластер
- якщо ітеровані дилери в кластерах і обмеження місткості дозволяє їх об’єднати, то вони об’єднуються

Таким чином, у результаті роботи алгоритму отримуємо список кластерів з впорядкованими маршрутами.

Оскільки у нас є 4 умови, то логіка алгоритму розростається. Як варіант, умови можна було винести в окремий файл, проте це не змінює ситуацію кардинально. Натомість, можна припустити, що з самого початку кожен дилер – це окремий кластер. В результаті, ітеруючи матрицю збереження, потрібно перевіряти тільки одну умову:

- якщо ітеровані дилери не транзитні і їх сумарна потреба не перевищує місткості ТЗ, то кластери об’єднуються в один, інакше виконується наступна ітерація

Таким чином, коду стає набагато менше. Після модифікації, алгоритм видавав на середніх та великих наборах даних(20 дилерів та більше) результат, на 4-5% кращий, аніж попередня його версія. Припускається, що даний варіант охопив більшу кількість варіацій умов, ніж було передбачено в попередньому випадку. Код останньої версії наведено в додатку В.

В пункті 2.6 проведено аналіз отриманих даних за допомогою збереження. Даний алгоритм показує хороший результат в багатьох випадках, особливо на невеликих наборах даних[x028]. Як уже раніше згадувалось, підхід є жадібним, що означає призначення оптимальним рішенням першого локального мінімуму на множині рішень. Проте такий локальний мінімум не завжди є глобальним. Тобто даний алгоритм не розуміє, що можна пропустити те чи інше рішення, допускаючи, що одне з наступних буде більш вдалим. Частково це виправляє розширення, описане в наступному пункті.

2.4 Розширення алгоритму збереження (Holmes and Parker)

Алгоритм гілок і меж базується на розгалуженні рішень, їх обчисленні та відсіюванні тих, що виходять за верхню чи нижню межі. Оскільки вартість кожного запропонованого варіанту все одно приходиться рахувати, а відсікається менша частина – алгоритм не є ефективним на великих наборах даних. Проте ідею з межами можна перенести на алгоритм збереження, що, власне і зробили Холмс та Паркер. Для більш детального розуміння варто розглянути наступний рисунок.

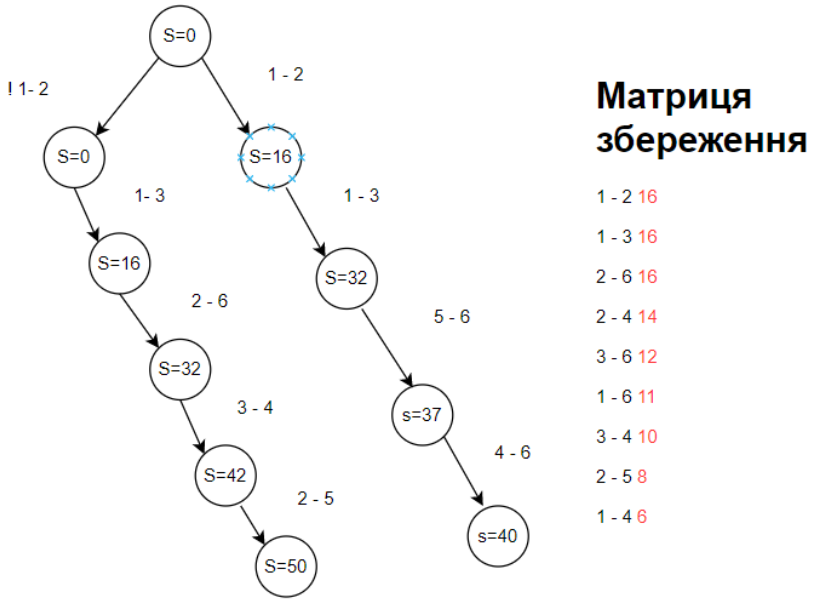

Рисунок 5 - Приклад роботи розширення Холмса і Паркера

На рис.5 праве розгалуження описує роботу вибору елементів збереження в класичному алгоритмі Кларка-Райта. Натомість, ліве розгалуження демонструє роботу розширення Холмса і Паркера, тобто на кожній ітерації відкидається елемент із найбільшим збереженням і алгоритм перераховує маршрут відповідно до зміненої матриці збереження. На практиці це реалізується доволі просто: в списку збереження елемент з найбільшим збереженням завжди перший, тобто на кожній ітерації потрібно видалити перший елемент, а потім для цієї матриці застосувати класичний алгоритм збереження. Результатом є найвигідніше з запропонованих рішень. Реалізація даного алгоритму наведена в додатку Г. Отже, в ході застосування даного розширення, можна отримати додаткову множину рішень, серед яких є ймовірність знаходження оптимального. Єдине, що із відношення даної множини до всієї множини рішень для того чи іншого набору даних слідує, що ймовірність знаходження оптимального рішення в даній множині є невисокою. Серед тестових наборів даних є такі, для яких розширення дає кращий результат, аніж класичне рішення, проте їх небагато.

Виникає потреба або спробувати інші розширення для алгоритму збереження, або вибрати інший алгоритм.

2.5 Алгоритм симуляції відпалу

Вибір нового алгоритму випав на імітацію відпалу в першу чергу через цікавість, викликану самою ідеєю роботи. Окрім цього, було знайдено дослідження інших метаевристичних алгоритмів, зокрема методу світлячків(аналогічного методу мурашиної колонії)[30], що показало задовільні результати в порівнянні з евристичним алгоритмом збереження. Розглянуті дослідження порівняння генетичного алгоритму та імітації відпаду саме в контексті VRP задачі([31], наприклад) допомогли остаточно визначитись на користь останнього.

Теоретичний опис роботи алгоритму відпалу наведений в секції 1.3.3, а фінальна його реалізація - в додатку Д.

Клас алгоритму реалізує інтерфейс IVRPAlgorithm, приймає у якості вхідних даних об’єкт InputData з початковими умовами задачі(матриця відстаней, максимальна місткість, список потреб для кожного дилера) та об’єкт параметрів, потрібних для роботи симуляції відпалу. Серед параметрів такі як початкова температура, кінцева температура, коефіцієнт охолодження та максимальна кількість ітерацій для кожного значення температури(код наведено в додатку Е). Згідно з загальним визначенням алгоритму:

- *T_Init* – температура, від якої почнеться процес охолодження.
- *T_Final* – температура, при якій завершується робота алгоритму.
- *Alpha* – коефіцієнт, на який буде зменшуватись поточна температура після кожної ітерації.
- *MaxIterationsCount* – значення, яке визначає кількість генерацій нових рішень для кожної нової ітерації температури.

Робота даного алгоритму представлена у вигляді рисунку нижче:

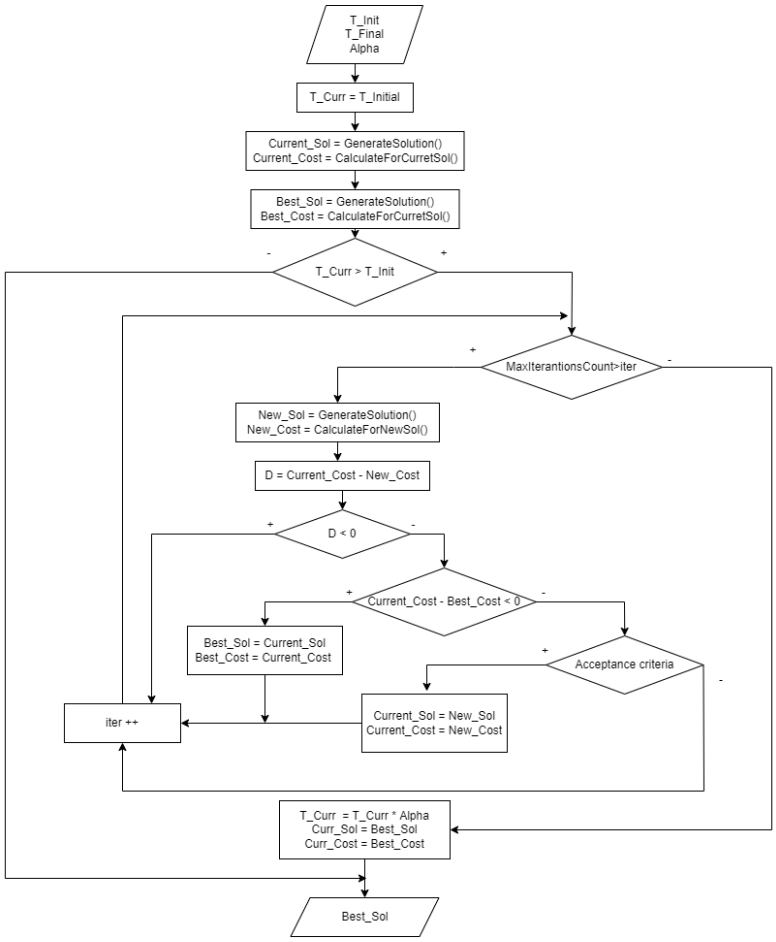

Рисунок 6 - Блок-схема реалізованого алгоритму імітації відпалу

Acceptance criteria – ймовірність переходу, наведена в секції 1.3.3 при описі роботи алгоритму відпалу. Логіка була написано відповідно до загального опису. Проте виникала потреба у коректній генерації нового рішення, адже від її ефективності залежало те, наскільки швидко буде знайдено оптимальне рішення. Недоліком симуляції відпалу є те, що у холодному стані робота алгоритму стає такою ж жадібною, як, наприклад, у алгоритму градієнтного спуску. Це означає, що за гарячу фазу оптимальне рішення має потрапити в ту ж область, де локальний мінімум буде рішенням задачі. Було вирішено генерувати новий список маршрутів на основі попереднього, тобто тимчасово визнаного найкращим. Для цього було визначено три функції: Swap, Insert та Reverse. Для отримання нового

рішення застосовувалась одна з функцій, відповідно до випадково згенерованого значення. Вони знаходяться у файлі NeigborhoodSolutionGenerator.cs у бібліотеці VPR.Core.SimulateAnnealing. Код наведено в додатку Є.

2.6Тестування роботи алгоритмів та аналіз отриманих даних

Усі тести проводились на лептопі з наступними характеристиками:

- Операційна система: Windows x64
- Процесор: Intel Core™ I7-8550U з архітектурою amd64
- Оперативна пам’ять: 16 Гб з частотою 2666 Мгц
- Пам’ять: твердотільний накопичувач об’ємом 256 Гб
- Середовище виконання коду: Microsoft Visual Studio 2022,
- Платформа: .NET 6 та версія C#: 10.0

Виміри часу виконання коду проводились за допомогою стандартних засобів C# класу Stopwatch(простір імен System.Diagnostics). Для більшої точності кожен алгоритм запускався кілька разів та на базі цього вираховувався середній час його виконання. Також доцільно зауважити, що алгоритм симуляції відпалу в силу своєї специфіки в деяких випадках видавав різні значення, тому тестувався по декілька разів. На основі отриманих результатів в якості фінального подавалося найкраще рішення з мінімально сумарною вартістю маршрутів. Оскільки алгоритм відпалу залежить від додаткових вхідних параметрів, таких, як початкова температура, кінцева, коефіцієнт охолодження та кількість ітерацій для значення температури, то із збільшенням складності вхідного набору VRP збільшувались і параметри. Дані налаштування підбирались відносно мінімальної межі їх значень, при яких алгоритм після багатократної перевірки генерував один і той самий результат, тобто знаходив глобальний

мінімум. Нижче наведена таблиця отриманих результатів. Таблиця з параметрами алгоритму симуляції відпалу наведена на рис. 7

		Вартість маршруту				Кількість потрібних ТЗ				Час виконання			Точність результату		
Назва набору даних	К-сть дилерів	CW	HP	SA	Absolute	CW	HP	SA	Absolute	CW	HP	SA	CW	HP	SA
data1.json	4	55	55	53	53	2	2	2	2	8ms	8ms	48ms	96%	96%	100%
data2.json	6	158	148	148	148	2	2	2	2	10ms	10ms	64ms	94%	100%	100%
data3.json	9	156	156	156	156	5	5	5	5	12ms	14ms	134ms	100%	100%	100%
A-n32-k5	38	863	863	843	831	6	6	6	6	68ms	898ms	2,6s	96%	96%	99%
A-n45-k7	44	1196	1196	1176	1146	7	7	7	7	88ms	1,9s	2,3s	96%	96%	97%
A-n55-k9	54	1138	1138	1088	1073	10	10	9	9	191ms	4,3s	4,5s	94%	94%	99%
A-n63-k10	62	1352	1352	1360	1314	10	10	10	10	116ms	6,1s	4,5s	97%	97%	97%
A-n80-k10	79	1863	1863	1863	1763	10	10	10	10	250ms	16,2s	2,9s	95%	95%	95%
B-n38-k6	37	834	834	820	805	6	6	6	6	105ms	1,9s	1,6s	97%	97%	98%
B-n43-k6	42	781	781	746	742	6	6	6	6	137ms	1,5s	1,6s	95%	95%	99%
B-n56-k7	55	755	754	719	707	7	7	7	7	214ms	4,4s	2,9s	94%	94%	98%
B-n67-k10	66	1104	1104	1088	1032	11	11	10	10	263ms	8,3s	2,5s	93%	93%	95%
B-78-k10	77	1270	1270	1259	1221	10	10	10	10	128ms	10,6s	2,2s	96%	96%	97%
E-n22-k4	21	388	388	381	375	4	4	4	4	72ms	470ms	1,7s	97%	97%	98%
E-n51-k5	50	584	584	527	521	6	6	5	5	302ms	2,5s	1,7s	89%	89%	99%
E-n76-k14	75	1054	1054	1063	1021	15	15	14	14	202ms	12s	3,1s	97%	97%	96%
E-n101-k14	100	1139	1139	1126	1067	14	14	14	14	158ms	46s	2,3s	94%	94%	95%
Golden_1	240	5688	-	5656	5623	9	-	9	9	1,6s	>10m	32s	99%	0%	99%
Golden_5	200	7265	-	6763	6460	5	-	5	5	932ms	>10m	31s	89%	0%	96%
Golden_9	255	663	-	615	579	14	-	14	14	1,1s	>10m	30s	87%	0%	94%

Рисунок 7 - Таблиця результатів тестування алгоритмів CW, HP та SA

На таблиці вище у колонці «Вартість маршруту» можна побачити вартість запропонованих маршрутів алгоритмами CW(класичний алгоритм збереження), HP(розширенням для CW) та SA(методом симуляції відпалу). Absolute – точна вартість маршруту, розрахована методом повного перебору (додається до наборів даних). Наступні колонки показують кількість задіяних ТЗ для кожного набору маршрутів, час виконання кожного алгоритму та точність у відсотках відносно точного рішення.

Перші три набори в форматі json – дані, згенеровані власноруч. Наступні умови задачі ЗМТ запозичені з відкритих джерел[28]. Набори з назвою Golden – дані, в яких велика кількість дилерів(більше 200). Дані, в умові яких до 20 дилерів вважаються малим набором, від 20 до 100 – середнім та 100 і більше – великим.

Результати дещо здивували, адже очікувалося, що на малих та середніх наборах даних алгоритм збереження дасть дещо гірший результат, аніж той,

який отриманий в ході тестування. На наборах даних, що містять до 50 дилерів його похибка складає до 7%. Варто відмітити швидкість роботи даного алгоритму, що на середніх та малих даних не перевищує 300ms. На більш складних наборах похибка результату росте, що очікувано, адже алгоритм є жадібним. На останніх наборах рішення відхиляється від точного більш, ніж на 10%. Це спровоковано великою кількістю дилерів – 200 і більше. Насправді, даний проект не планується використовувати в задачах, де дилерів більше 100, а вище описані набори включені в тестування з метод отримання більш якісної та загальної статистики. Проте похибка у більш, аніж 5% вважається неприпустимою. Даний алгоритм варто використовувати, де дозволяється дана неточність та дуже критичний час виконання.

Розширення для алгоритму збереження, запропоноване в п. 2.4 відпрацьовує значно повільніше, адже, на відміну від класичного алгоритму, він ітерує додатково низку рішень, кожного разу відсікаючи черговий елемент матриці збереження. Дана матриця прямопропорційно залежить від кількості дилерів, тому на таких наборах даних, де багато клієнтів даний алгоритм працює значно довше, інколи не виправдано довго. Щодо точності запропонованого результату, то в гіршому випадку, результат не відрізняється від результату класичного алгоритму збереження, а в одиницях випадків HP розширення пропонує дещо кращий результат. Це зумовлено тим, що до розгляду маршрутів додається список нових рішень, але оскільки їх небагато, порівнюючи з усією можливою кількістю можливих маршрутів, то ймовірність знайти точне, або більш наближене до точного рішення залишається доволі низькою. Звісно, можна придумати ще одне або кілька правил та включити їх до розгляду, але в такому випадку відбувається повільне наближення даного методу до методу повного перебору та,

відповідно, росте час роботи. Оскільки очікувалося, що відсоток кращого результату відносно CW буде більшим, то можна сказати, що даний алгоритм не виправдовує часу, який іде на його виконання.

Алгоритм симуляції відпалу показав хороші результати за розумний час. В багатьох випадках час виконання даного алгоритму був кращим за час роботи HP методу, проте все одно залишався гіршим за час CW. Це зумовлено тим, що SA ітерує велику кількість даних, порівняно з обробкою всього лише одного варіанту CW. Слід сказати, що під час роботи даних алгоритмів враховується і час перетворення структур C#, тому не виключено, що перписавши CW та SA на простіші структури, прибравши певні перетворення, ми отримаємо час виконання дещо кращий. Але оскільки алгоритми тестуються в однакових умовах, то вибір даного часу для порівняння цілком виправданий. Інтерес полягає швидше у порівнянні роботи одного алгоритму відносно іншого, а не у абсолютно точній швидкості роботи того чи іншого методу. Щодо результатів SA, то в найгіршому випадку відхилення від точного результату не перевищувало 5% на великих наборах даних. Це зумовлено тим, що при збільшенні дилерів, кількість рішень, які можна згенерувати стрімко збільшується, тому доводиться відповідно збільшувати початкову температуру та максимальну кількість ітерацій для кожної температури. Алгоритм не гарантує, що у гарячій фазі на великих наборах даних буде знайдено оптимальне рішення, яке є глобальним мінімумом, або наближеним до нього. Нижче наведено таблицю підібраних параметрів для кожного набору даних:

	Параметри SA			
Назва набору даних	T_Initial	T_Final	Alpha	Maximum iterations count
data1.json	100	0.1	0.99	50
data2.json	100	0.1	0.99	100
data3.json	1000	0.1	0.99	500
A-n32-k5	10000	0.1	0.99	500
A-n45-k7	10000	0.1	0.99	500
A-n55-k9	1000000	0.1	0.99	500
A-n63-k10	10000000	0.1	0.99	500
A-n80-k10	100000000	0.1	0.99	500
B-n38-k6	10000	0.1	0.99	500
B-n43-k6	10000	0.1	0.99	500
B-n56-k7	100000	0.1	0.99	500
B-n67-k10	1000000	0.1	0.99	500
B-78-k10	100000000	0.1	0.99	800
E-n22-k4	100000	0.1	0.99	500
E-n51-k5	100000	0.1	0.99	500
E-n76-k14	100000000	0.1	0.99	800
E-n101-k14	100000000	0.1	0.99	800
Golden_1	1000000000000	0.1	0.99	5000
Golden_5	1000000000000	0.1	0.99	5000
Golden_9	1000000000000	0.1	0.99	8000

Рисунок 8 Таблиця параметрів для SA відповідно до набору даних

Результати навіть в найгіршому випадку не перевищують відхилення в 5% відносно точного результату, час відведений на роботу є виправданий, тому даний алгоритм можна використовувати в цілях побудови маршрутів для розвезення автомобілів по дилерських центрах. Щодо майбутніх покращень, то можна спробувати більш ефективно підібрати механізм генерації маршрутів, або накласти більше обмежень для відсічення непридатних маршрутів, тим самим збільшивши кількість нових для розгляду та порівняння.

ВИСНОВКИ

Транспортна логістика відіграє сьогодні важливу роль в роботі практично будь-якого підприємства. Вартість доставки часто така еквівалентна вартості товару, тому велике значення надається впровадженню методів оптимізації маршрутів, що дозволяють іноді економити десятки відсотків від вартості товару. Крім доставки товарів, є ще безліч аспектів, в яких можуть бути застосовані методи оптимізації маршрутів: перевезення людей, туризм, управління супутниками, збір геному, кластеризація масивів даних, проектування телекомунікаційних мереж, з'єднання ряду пунктів кільцевими лініями енергопередачі і газопостачання, проектування топології інтегральних схем . Оптимальні маршрути потрібні ремонтним бригадам, тим, хто здійснює доставку різноманітної продукції, перевіряє різні мережі, тестує сайти і навіть роботів, тощо.

Дана робота направлена на пошук, дослідження та реалізацію алгоритму пошуку оптимального рішення методами кластеризації. В ході дослідження наведено опис постановок VRP та короткий опис алгоритмів, що дозволяють знайти точний чи наближений розв’язок даної задачі. Отримання точного результату навіть класичного варіанту VRP потребує повного перебору. Оскільки задача має складність NP, то кількість комбінацій із збільшенням дилерів зростає факторіально. Комбінацій виникає дуже багато, тому час їх перебору зі збільшенням точок, які потрібно відвідати зростає дуже стрімко. З точки зору бізнесу, час відведений на обчислення маршруту методом точного перебору є невиправданим. Тому, в ході дослідження даної проблеми часто застосовуються нестандартні рішення, зокрема евристичні та метаевристичні, що дають наближений результат за допустимий час. Перші базуються на певних математичних правилах, другі ж запозичені з природніх процесів та частково базуються на

стохастичності. Перші піддаються аналізу та повному контролю в той час, як ефективність других залежить від коректного підбору параметрів, потрібних метаевристчним алгоритмам. У випадку з методом симуляції відпалу, описаному в даній роботі, це - кількість ітерацій пошуку областей з локальним мінімумом у вибірці та кількість ітерацій для пошуку локального мінімуму у вибраній області.

Використання алгоримтів різних груп для вирішення задачі маршрутизації має як свої переваги, так і недоліки. Варіюється як точність, так і складність обчислень та час. В ході дослідження реалізовано та проаналізовано три алгоритми: евристичний метод збереження, його розширення та метаевристичний метод симуляції відпалу. Аналізуючи результати, можна побачити, що алгоритм потребують різного часу виконання та дають різний результат. В силу того, що перший та другий алгоритми жадібні – вони не завжди знаходять глобальний мінімум, зупиняючись на знайденому локальному. У випадку другого, результат залежить від ймовірності , що за час активного пошуку локального мінімуму оптимальне рішення буде саме в одній з досліджуваних областей. Підвищення кількості ітерацій збільшує відсоток цієї ймовірності, проте не завжди вдається знайти саме глобальний мінімум. В більшості випадків результат є дуже наближеним до точного(похибка у 5% і менше.)

З тих же отриманих результатів можна зробити висновок, що класичний алгоритм збереження на малих та середніх наборах даних є доволі ефективним і має похибку до 9%. Запропоноване розширення алгоритму вище в невеликій кількості випадків дає кращий результат, але обраховується помітно довше, що не завжди виправдано оптимальністю отриманого рішення. В більшості випадків похибка у 9% не влаштовує бізнес, адже веде до великих витрат. Наприклад, коли кошт доставки

автомобілів по дилерських центрах сягатиме 1 млн. грн. (умовно), то використання даного алгоритму приведе до втрат у розмірі 90 тис. грн. Із підвищенням складності набору даних(збільшенням набору дилерів, депо, маршрутів, тощо) похибка збільшується. Інший розглянутий алгоритм – алгоритм симуляції відпалу дає значно меншу похибку на малих, середніх та великих об'ємах даних(до 5% в найгіршому випадку). Час обчислення зазвичай більший у кілька разів, проте все ще залишається в допустимих рамках. В більшості випадків, він кращий за час обчислення НР розширення методу збереження. Тому використання алгоритму відпалу вважається доцільним на будь-яких наборах даних. Звісно, зі збільшенням набору даних цей алгоритм, як і попередній, втрачає свою ефективність, проте межа даного процесу набагато вища та точність отриманого рішення все ще залишається на допустимому рівні(5% для наборів із 200 і більше дилерів). Порівняння з іншими евристичним чи метаевристичними алгоримами не наводиться в силу неможливості реалізації їх за відведений час. А порівнняння даних алгоритмів на різних мовах програмування не можна вважати справедливим в силу специфіки типів даних, їх обробки, перетворень та інших нюансів кожної мови. Проте було досліджено інші роботи з даними порівнянням[30] та [31] під час вибору алгоритмів.

Код з реалізацією вище описаних алгоритмів та тестовими наборами даних можна знайти у відкритому доступі за посиланням **https://github.com/Anonim147/vrp-solving .**

ПЕРЕЛІК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Транспортная логистика [Електронний ресурс] – Режим доступу до ресурсу: https://www.axelot.ru/knowhow/press/detail_48008/ .
2. Швец А. Н. Задача коммивояжёра [Електронний ресурс] / Н. Швец – Режим доступу до ресурсу: http://mech.math.msu.su/~shvetz/54/inf/perlproblems/chCommisVoyageur.xhtml
3. Dantzig G.B., Ramser J.H. The Truck Dispatching Problem // Management science, Vol. 6, No. 1, 1959. pp. 80-91.
4. Christofides N., Mingozzi A., Toth P. The vehicle routing problem // In: Combinatorial Optimization. Wiley, 1979. pp. 315–338.
5. Solomon M.M. Algorithms for the vehicle routing and scheduling problems with time window constraints // Operations research, Vol. 35, No. 2, 1987. pp. 254-265.
6. Chao I.M., Golden B.L., Wasil E. A new heuristic for the multi-depot vehicle routing problem that improves upon best-known solutions // American Journal of Mathematical and Management Sciences, Vol. 13, No. 3-4, 1993. pp. 371-406.
7. Christofides N., Beasley J.E. The period routing problem // Networks, Vol. 14, No. 2, 1984. pp. 237-256.
8. Francis P., Smilowitz K. Modeling techniques for periodic vehicle routing problems // Transportation Research Part B: Methodological, Vol. 40, No. 10, 2006. pp. 872- 884
9. Belmecheri F., Prins C., Yalaoui F., Amodeo L. Belmecheri F. et al. Particle swarm optimization to solve the vehicle routing problem with heterogeneous fleet, mixed backhauls, and time windows // Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE International Symposium on., 2010. pp. 1-6.
10. Moura A., Oliveira J.F. An integrated approach to the vehicle routing and container loading problems // OR spectrum, Vol. 31, No. 4, 2009. pp. 775-800.
11. Dror M., Laporte G., Trudeau P. Vehicle routing with split deliveries // Discrete Applied Mathematics, Vol. 50, No. 3, 1994. pp. 239-254
12. Archetti C., Savelsbergh M.W.P., Speranza M.G. To split or not to split: That is the question // Transportation Research Part E: Logistics and Transportation Review, Vol. 44, No. 1, 2008. pp. 114-123.

13. Gendreau M., Laporte G., Séguin R. Stochastic vehicle routing // European Journal of Operational Research, Vol. 88, No. 1, 1996. pp. 3-12.
14. Chen D., Chen D., Yang Y. Nondeterministic Vehicle Routing Problem: A Review // Advances in Information Sciences and Service Sciences, Vol. 5, No. 9, 2013. pp. 485-493.
15. Garrido P., Riff M.C. DVRP: a hard dynamic combinatorial optimisation problem tackled by an evolutionary hyper-heuristic // Journal of Heuristics, Vol. 16, No. 6, 2010. pp. 795-834.
16. Sariklis D., Powell S. A heuristic method for the open vehicle routing problem // Journal of the Operational Research Society, Vol. 51, No. 5, 2000. pp. 564-573
17. Choi E., Tcha D.W. A column generation approach to the heterogeneous fleet vehicle routing problem // Computers & Operations Research, Vol. 34, No. 7, 2007. pp. 2080-2095
18. Bard J.F., Huang L., Dror M., Jaillet P. A branch and cut algorithm for the VRP with satellite facilities // IIE transactions, Vol. 30, No. 9, 1998. pp. 821-834.
19. Fischetti M., Vigo D., Toth P. A Branch-and-Bound Algorithm for the Capacited Vehicle Routing Problem on Directed Graphs. Operations Research. 1994. pp. 846–851.
20. John E. Mitchell, Branch-and-Cut Algorithms for Combinatorial Optimization Problems [Text] / E. Mitchell John // Mathematical Sciences Rensselaer Polytechnic Institute Troy, NY, USA. – 1999. – pp.19
21. Sorensen K., Cuervo D. P., Floria A. A critical analysis of the “improved Clarke and Wright savings algorithm”. Operational Research. 2017. P. 1–3.
22. Fisher M.L., Jaikumar R. A generalized assignment heuristic for vehicle routing // Networks, Vol. 11, No. 2, 1981. pp. 109-124.
23. D.M., Hjorring C., Glover F. Extensions of the petal method for vehicle routeing // Journal of the Operational Research Society, Vol. 44, No. 3, 1993. pp. 289-296.
24. Sweep Algorithm in Vehicle Routing Problem For Public Transport / G. W. Nurcahyo et al. Teknologi Maklumat. 2002. No. 2. pp. 51–64.
25. Ferrucci, Francesco, Pro-active Dynamic Vehicle Routing Operations Research. — 2013. — pp. 280.
26. Елизаров С. Введение в оптимизацию. Имитация отжига. [Електронний ресурс] / Habr. – Режим доступу до ресурсу: https://habr.com/ru/post/209610/.

27. ru_vds. Python – это медленно. Почему?. / Хабр. URL: https://habr.com/ru/company/ruvds/blog/418823/
28. CVRPLIB - All Instances. CVRPLIB - All Instances[Електронний ресурс] – Режим доступу до ресурсу:: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/
29. ЦВТ Ц. В. Т. –. Что нового в .NET 6?. / Хабр. – Режим доступу до ресурсу: https://habr.com/ru/post/573434/
30. Bala Sai Shankar R. Solution to a Capacitated Vehicle Routing Problem Using Heuristics and Firefly Algorithm / R. Bala Sai Shankar, D. Reddy, P. Venkataramaiah. // International Journal of Applied Engineering Research. – 2018. – pp. 15252–15254.
31. Hosny M. I. Comparing Genetic Algorithms and Simulated Annealing for Solving the Pickup and Delivery Problem with Time Windows. College of Computer and Information Sciences. 2011. Conference: Proceedings of the 2011 International Conference on Artificial Intelligence. pp. 6–7.

ДОДАТКИ

Додаток А

Приклад відкритих даних задачі CVRP(22 дилери, мінімальна кількість ТЗ – 4, оптимальна вартість маршруту – 375 од.

```
NAME : E-n22-k4
COMMENT : (Christophides and Eilon, Min no of trucks: 4, Optimal value: 375)
TYPE : CVRP
DIMENSION : 22
EDGE_WEIGHT_TYPE : EUC_2D
CAPACITY : 6000
NODE_COORD_SECTION
1 145 215
2 151 264
3 159 261
4 130 254
5 128 252
6 163 247
7 146 246
8 161 242
9 142 239
10 163 236
11 148 232
12 128 231
13 156 217
14 129 214
15 146 208
16 164 208
17 141 206
```

Додаток Б

Реалізація екстрактора, який обробляє дані із форматуванням, наведеними в дод. А.

```
public class ClassicTextDataExtractor : IDataExtractor
   {
      public InputData Extract(Stream stream)
      {
         ReadingStateEnum state = ReadingStateEnum.DATA;
         int dealersCount = 0;
         int capacity = 0;
         List<int> demands = new List<int>();
         List<Coord> coords = new List<Coord>();

         using StreamReader sr = new StreamReader(stream);
         while (!sr.EndOfStream)
         {
            var data = sr.ReadLine()!;
            if (string.IsNullOrWhiteSpace(data))
               continue;

            if (data.Contains("DIMENSION") && state == ReadingStateEnum.DATA)
            {
               data = data.Trim();
               dealersCount= int.Parse(data[(data.IndexOf(':') + 1)..]);
            }

            if (data.Contains("CAPACITY") && state == ReadingStateEnum.DATA)
            {
               data = data.Trim();
               capacity = int.Parse(data[(data.IndexOf(':') + 1)..]);
            }

            else if (data.Contains("NODE_COORD_SECTION") && state == ReadingStateEnum.DATA)
            {
               state = ReadingStateEnum.COORDS;
            }

            else if (data.Contains("DEMAND_SECTION") && state == ReadingStateEnum.COORDS)
            {
               state = ReadingStateEnum.DEMANDS;
            }

            else if (data.Contains("DEPOT_SECTION") && state == ReadingStateEnum.DEMANDS)
            {
               break;
            }


            else if (state == ReadingStateEnum.COORDS)
            {
               int id = int.Parse(data[..data.IndexOf(" ")]);
               data = data[(data.IndexOf(" ") + 1)..];

               int x = int.Parse(data[..data.IndexOf(" ")]);
               data = data[(data.IndexOf(" ") + 1)..];

               int y = int.Parse(data.Trim());

               coords.Add(new Coord(id, x, y));
            }
```

```
    else if (state == ReadingStateEnum.DEMANDS)
    {
        int id = int.Parse(data[..data.IndexOf(" ")]);
        data = data[(data.IndexOf(" ") + 1)..];

        int demand = int.Parse(data.Trim());

        demands.Add(demand);
    }
}
if (coords.Count != demands.Count)
    throw new ArgumentException("Coords and dealers quantities must be the same");

if (coords.Count != dealersCount)
    throw new ArgumentException("Not all coords is loaded. Check file and try again");

if (demands.Count != dealersCount)
    throw new ArgumentException("Not all demands is loaded. Check file and try again");

if (capacity < 1)
    throw new ArgumentException("Capacity should not be empty or negative");
return new InputData
{
    MaxCapacity = capacity,
    Demands = demands,
    DistanceMatrix = EuclidHelper.GetEuclidDistanceMatrix(coords)};}}
```

Додаток В.

Фінальний варіант алгоритму збереження

```
protected List<List<int>> ProcessSaving(List<SavingMatrixItem> savingItems, List<Dealer> dealers, int MaxCapacity)
    {
        List<Cluster> clusters = dealers
            .Skip(1)
            .Select(dealer => new Cluster(dealer))
            .ToList();

        foreach (var item in savingItems)
        {
            var firstCluster = GetClusterByDealerId(clusters, item.FirstDealer.Id);
            var secondCluster = GetClusterByDealerId(clusters, item.SecondDealer.Id);

            if (firstCluster != secondCluster
                && !firstCluster.IsDealerTransient(item.FirstDealer.Id)
                && !secondCluster.IsDealerTransient(item.SecondDealer.Id)
                && (firstCluster.WorkLoad + secondCluster.WorkLoad) <= MaxCapacity)
            {
                if (firstCluster.IsDealerFirst(item.FirstDealer.Id))
                {
                    firstCluster.ReverseDealers();
                }
                if (secondCluster.IsDealerLast(item.SecondDealer.Id))
                {
                    secondCluster.ReverseDealers();
                }
                firstCluster.AddCluster(secondCluster);
                clusters.Remove(secondCluster);
            }
        }

        List<List<int>> solutionList = clusters.Select(cluster =>
            cluster.Dealers
                .Select(dealer => dealer.Id)
                .ToList()
            ).ToList();

        solutionList.ForEach(solution =>
        {
            solution.Insert(0, 1);
            solution.Add(1);
        });

        return solutionList;
    }
```

Додаток Г.

Реалізація розширення алгоритму збереження

```
public class HPSaving : SavingBase, IVRPAlgorithm
   {
      public List<List<int>> Process(InputData input, object? parameters = null)
      {
         //you should remember that first dealer is always depot
         List<Dealer> dealers = input.Demands
            .Select((demand, id) => new Dealer(id + 1, demand))
            .ToList();

         List<SavingMatrixItem> savingItems = GetSavingMatrix(dealers.Skip(1).ToList(), input.DistanceMatrix)
            .OrderByDescending(si => si.SavingCost)
            .ToList();

         List<List<int>> bestSolution = new List<List<int>>();
         double bestPath=int.MaxValue;

         while(savingItems.Count > 0)
         {
            List<List<int>> newSolution = ProcessSaving(savingItems, dealers, input.MaxCapacity);
            double newPath = ResultHelper.GetTotalPath(input.DistanceMatrix, newSolution);

            if(newPath < bestPath)
            {
               bestSolution = newSolution;
               bestPath = newPath;
            };
            savingItems.RemoveAt(0);
         }

         return bestSolution;
      }
   }
```

Додаток Д.

Код алгоритму симуляції відпалу

```
{
    private static readonly FastRandom _rnd = new FastRandom();

    public static List<int> CurrentSolution { get; set; }
    public static double CurrentSolutionCost { get; set; }
    public static List<int> BestSolution { get; set; }
    public static double BestSolutionCost { get; set; }


    public static int MinVehicleCount { get; set; }
    public static List<int> Demands { get; set; }
    public static double[,] DistanceMatrix { get; set; }


    public List<List<int>> Process(InputData input, object? parameters = null)
    {
        SimulateAnnealingParams p = (SimulateAnnealingParams)parameters
            ?? throw new Exception("You must pass SA parameters!");
        double TInitial = p.TInitial;
        double TFinal = p.TFinal;
        double ALPHA = p.Apha;
        int MAX_ITERATIONS_COUNT = p.MaxIterationsCount;

        double currentTemp = TInitial;

        int maxCapacity = input.MaxCapacity;
        Demands = input.Demands;
        DistanceMatrix = input.DistanceMatrix;
        MinVehicleCount = GetMiVehiclesCount(input.Demands, input.MaxCapacity);

        InitializeSolution(maxCapacity);

        while (currentTemp >= TFinal)
        {
            for (int i = 0; i < MAX_ITERATIONS_COUNT; i++)
            {
                List<int> NewSolution = NeigborhoodSolutionGenerator.Generate(CurrentSolution);
                double NewSolutionCost = CalculateCost(NewSolution, maxCapacity, DistanceMatrix);

                double difference = NewSolutionCost - CurrentSolutionCost;

                if (difference < 0)
                {
                    CurrentSolution = new List<int>(NewSolution);
                    CurrentSolutionCost = NewSolutionCost;

                    if (CurrentSolutionCost < BestSolutionCost)
                    {
                        BestSolution = new List<int>(CurrentSolution);
                        BestSolutionCost = CurrentSolutionCost;

                        Console.WriteLine("Temperature = " + currentTemp.ToString("#.##") + " BestSolutionPath = " +
BestSolutionCost.ToString("#.##"));
                    }
                }
                else
                {
                    double r = _rnd.NextDouble();
                    double boltzman = Math.Exp(-difference / currentTemp); // boltzman acceptance criteria
```

```
            if (r < boltzman)
            {
               CurrentSolution = new List<int>(NewSolution);
               CurrentSolutionCost = NewSolutionCost;
            }
         }
      }

      currentTemp *= ALPHA;

      CurrentSolution = new List<int>(BestSolution);
      CurrentSolutionCost = BestSolutionCost;
   }

   return NormalizeSolution(BestSolution, maxCapacity, DistanceMatrix);
}

int GetMiVehiclesCount(List<int> demands, int maxCapacity)
{
   int TotalDemand = demands.Sum();
   int Nvehicle = (int)Math.Ceiling((double)(TotalDemand / maxCapacity));
   return (demands.Count - 1) + Nvehicle;
}

void InitializeSolution(int maxCapacity)
{
   CurrentSolution = RandomInitialSolution();
   CurrentSolutionCost = CalculateCost(CurrentSolution, maxCapacity, DistanceMatrix);

   BestSolution = new List<int>(CurrentSolution);
   BestSolutionCost = CurrentSolutionCost;
}

static List<int> RandomInitialSolution()
{
   List<int> sol = new List<int>();
   for (int i = 0; i < MinVehicleCount; i++)
   {
      int node = (i + 1);
      if (i >= (Demands.Count - 1)) node = 0;

      int pos = _rnd.Next(0, sol.Count);

      sol.Insert(pos, node);
   }
   return sol;
}

double CalculateCost(List<int> solution, int maxCapacity, double[,] distanceMatrix)
{
   double path = 0;
   int source = 0;
   int destination = 0;
   int workLoad = 0;

   for (int i = 0; i < solution.Count; i++)
   {
      destination = solution[i];
      workLoad += Demands[destination];

      if (destination == 0)
      {
         path += distanceMatrix[source, destination];
         source = destination;
         workLoad = 0;
```

```
        }
        else if (workLoad > maxCapacity)
        {
            path += distanceMatrix[source, 0];
            workLoad = 0;

            path += distanceMatrix[0, destination];
            workLoad += Demands[destination];
            source = destination;
        }
        else
        {
            path += distanceMatrix[source, destination];
            source = destination;
        }
    }

    if (source != 0)
    {
        path += distanceMatrix[source, 0];
    }
    return path;
}

//dealers must starts from 1, not from 0
static List<List<int>> NormalizeSolution(List<int> solution, int maxCapacity, double[,] distanceMatrix)
{
    double path = 0;
    int source = 0;
    int destination = 0;
    double workLoad = 0;

    List<List<int>> solList = new List<List<int>>();
    solList.Add(new List<int>());
    solList.Last().Add(1);

    for (int i = 0; i < solution.Count; i++)
    {
        destination = solution[i];
        if (destination == 0 && source == 0)
            continue;

        workLoad += Demands[destination];

        if (destination == 0)
        {
            path += distanceMatrix[source, destination];
            source = destination;
            workLoad = 0;

            solList.Last().Add(destination + 1);
            solList.Add(new List<int>());
            solList.Last().Add(1);
        }
        else if (workLoad > maxCapacity)
        {
            path += distanceMatrix[source, 0];
            workLoad = 0;

            solList.Last().Add(1);

            solList.Add(new List<int>());
            solList.Last().Add(1);
            solList.Last().Add(destination + 1);
```

```
                path += distanceMatrix[0, destination];
                workLoad += Demands[destination];

                source = destination;
            }
            else
            {
                path += distanceMatrix[source, destination];
                source = destination;
                solList.Last().Add(destination + 1);
            }
        }

        if (destination != 0)
        {
            solList.Last().Add(1);
        }

        return solList;
    }
}
```

Додаток Е.

Параметри алгоритму симуляції відпалу

```
public class SimulateAnnealingParams
   {
      public double TInitial { get; set; }
      public double TFinal { get; set; }
      public double Apha { get; set; }
      public int MaxIterationsCount { get; set; }

      public static SimulateAnnealingParams CreateDefault()
      {
         return new SimulateAnnealingParams()
         {
            TInitial = 10000,
            TFinal = 0.001,
            Apha = 0.99,
            MaxIterationsCount = 5000
         };
      }
   }
```

Додаток Є.

Методи для генерації набору маршрутів.

```
public static List<int> Generate(List<int> existedSolution)
        {
            List<int> newSolution = new List<int>(existedSolution);
            double p = _rnd.NextDouble();

            if (p < 0.337)
            {
                Swap(newSolution);
            }
            else if (p < 0.667)
            {
                Insert(newSolution);
            }
            else
            {
                Reverse(newSolution);
            }
            return newSolution;
        }

static void Swap(List<int> solution)
        {
            int N = solution.Count();
            int i = _rnd.Next(N - 1);
            int j = _rnd.Next(N - 1);
            int temp = solution[i];
            solution[i] = solution[j];
            solution[j] = temp;
        }
        static void Insert(List<int> solution)
        {
            int N = solution.Count();
            int i = _rnd.Next(N - 1);
            int j = _rnd.Next(N - 1);
            int tempJ = solution[j];
            solution.RemoveAt(j);
            solution.Insert(i, tempJ);
        }
        static void Reverse(List<int> solution)
        {
            int N = solution.Count();
            int i = _rnd.Next(N - 1);
            int j = _rnd.Next(N - 1);
            int index;
            int count;
            if (i > j)
            {
                index = j;
                count = i - j;
            }
            else
            {
                index = i;
                count = j - i;
            }
            solution.Reverse(index, count);
}
```