КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ РОЗВ'ЯЗАННЯ ЗАДАЧ ЛОГІСТИКИ

Виконав студент 4 курсу
Давидов Олексій Павлович __________
(підпис)

Науковий керівник:
доцент, кандидат технічних наук
Рябоконь Дмитро Ігорович _________
(підпис)

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

Студент __________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики

«____» ________________ 202_ р.,

протокол № ______________________

Завідувач кафедри

В. М. Терещенко __________
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 54 сторінки, 19 фрагментів коду, 4 ілюстрації, 8 джерел посилань

АНАЛІТИНІ АЛГОРИТМИ, ВІДОБРАЖЕННЯ РОБОТИ АЛГОРИТМІВ, МЕТАЕВРІСТИЧНІ АЛГОРИТМИ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, ОПТИМІЗАЦІЯ БЛИЗЬКОСТІ ПОЛІТИК, ПРОБЛЕМА МАРШРУТИЗАЦІЇ ТРАНСПОРТНОГО ЗАСОБУ

Об’єктом роботи є симуляція розв’язування VRP за допомогою PPO, та порівняння з аналітичними результатами, за допомогою програмних засобів. Предметом роботи є програмний засіб для симуляції розв’язування VRP за допомогою PPO, та аналітичних алгоритмів.

Метою роботи є створення програмного засобу для симуляції розв’язування VRP за допомогою PPO, та аналітичних алгоритмів, та отримання досвіду для виконання схожих робіт.

Методи розроблення: комп’ютерне моделювання, методи обчислювальної геометрії та комп’ютерної графіки, методи штучного інтелекту, розробка програмного продукту на основі ощадливої моделі. Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки обрано PyCharm Community edition 2023.1, мова програмування Python, бібліотеки Matplotlib , NetworkX, Numpy, Torch.

Результати роботи: виконано загальний огляд проблем маршрутизації транспортного засобу, проаналізовано переваги та недоліки використання різних варіантів задач для перевірки працездатності програмного засобу, розроблено програмний продукт, аналітичні алгоритми і метаеврістичні алгоритми до нього, розроблено спосіб наочно візуалізувати роботу алгоритмів.

Програмний продукт може застосовуватись для дослідження використання методів навчання з підкріпленням для розв’язання проблем маршрутизації транспортного засобу, та порівняння з аналітичними алгоритмами.

ЗМІСТ

ВСТУП 6
1 ВИБІР ЗАДАЧІ ДЛЯ СИМУЛЯЦІЇ 9
1.1 Розгляд вже існуючих задач, та їх вирішення 9
1.1.1 VRP 10
1.1.2 VRP з прибутком (VRPP) 15
1.1.3 VRP з отриманням і доставкою (VRPPD) 16
1.1.4 VRP з LIFO 17
1.1.5 VRP із часовими вікнами (VRPTW) 18
1.1.6 Ємна VRP (CVRP) 18
1.1.7 VRP з кількома поїздками (VRPMT) 18
1.1.8 Багато складська VRP (MDVRP) 19
1.1.9 VRP з трансферами (VRPWT) 20
1.2 Аналіз умов нової задачі для симуляції 22
1.3 Умова задачі для симуляції 25
2 ВИКОРИСТАНІ АЛГОРИТМИ ТА ЇХ ІМПЛЕМЕНТАЦІЯ 26
2.1 Попередній розрахунок 26
2.1.1 Створення випадкового графу 26
2.1.2 Розрахунок матрицю відстаней 28
2.2 Аналітичний алгоритм розв’язання задачі 30
2.3 Регуляризація ентропії 33
2.4 Proximal Policy Optimization (PPO) 36
3 СТРУКТУРА ПРОГРАМИ, РЕАЛІЗАЦІЯ ЗАДАЧІ В ЇЇ МЕЖАХ 38
3.1 Огляд складових програмного засобу 38
3.2 Середовище, його складові та імплементація 38
3.2.1 Поле 38

3.2.2 Депо 39

3.2.3 Кур'єр 40

3.2.4 Генератор запитів 42

3.2.5 Запити 42

3.2.6 Калькулятор поля або Візуалізатор поля 42

3.3 Колективний розум, його складові та імплементація 45

3.4 Запуск РРО у середовищі 47

4 РЕЗУЛЬТАТИ НАВЧАННЯ НА ПОСТАВЛЕНІЙ ЗАДАЧІ 48

ВИСНОВКИ 51

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 52

ДОДАТОК А Реалізація аналітичного алгоритму без використання біту перевезень 54

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

VRP – Vehicle Routing Problem

PPO – Proximal Policy Optimization

IDE – Integrated Development Environment, інтегрована середа розробки

LIFO – Last In – First Out, останнім прийшов – першим пішов

VRPP – Vehicle Routing Problem with Profits

VRPPD – Vehicle Routing Problem with Pickup and Delivery

VRPTW – Vehicle Routing Problem with Time Windows

CVRP – Capacitated Vehicle Routing Problem

VRPMT – Vehicle Routing Problem with Multiple Trips

OVRP – Open Vehicle Routing Problem

MDVRP – Multi-Depot Vehicle Routing Problem

VRPWT – Vehicle Routing Problem with Transfers

ILP – Integer Linear Programming

RL – Reinforcement Learning, навчання з підкріпленням

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. Сьогодні дороги є венами цивілізації, по яким тече усе необхідне для її існування, від зерна і води, до комплексних мікросхем. Усе потрібно кудись доставити, але асфальтне покриття і автомобіль це не все що потрібно для виконання цієї задачі, потрібно ще знати куди їхати.

Це і є однією з головних задач логістики – дізнатись як оптимально спрямувати набір транспортних засобів з центрального депо до споживачів, з урахуванням деяких обмежень. Така проблема називається Vehicle Routing Problem (Далі - VRP), і її можна виразити менш науковим чином.

Уявімо що є велика кількість кур’єрів, яким потрібно відвозити замовлення в різні частини міста. Як скоординувати їх так щоб хтось з не пішов через пів міста бо був єдиним вільним на той час, а через кілька хвилин звільняється людина в пішій доступності від замовника? Такі задачі є складними не тільки тому що використовуються важкі алгоритми, а ще й тому що ми обмежені нашою фантазією. Цю позицію можна продемонструвати на алгоритмах сортування. Так легко «довести» що Сортування бульбашкою є найбільш ефективним, бо іншими відомими алгоритмами є Випадкове сортування та Сталінське сортування.

Тому, навить якщо є існуючи алгоритми, які можуть вирішити подібні задачі логістики – скоріш за все ми не можемо довести з абсолютною гарантією, що не існує кращого рішення. Саме через це використовують методи штучного інтелекту, які показують значно кращі результати ніж аналітичні аналоги.

Актуальність роботи та підстави для її виконання. Методи штучного інтелекту вже використовують для вирішення найрізноманітніших задач логістики. Найбільше використовують саме генетичний алгоритм, але можливо інші засоби є не менш доречними в деяких випадках.

Тому є актуальним створення і вивчення можливих рішень задач логістики, використовуючи інші методи штучного інтелекту, які можуть показати кращі результати. Так вбачається доцільним використання Навчання з підкріпленням,

для вирішення VRP, бо виконання замовлень є тією дією яку ми хочемо заохочувати, незалежно від того яким шляхом ми дістались до замовника.

Мета й завдання роботи. Метою дипломної роботи є створення програмного засобу для дослідження розв'язування задачі типу VRP за допомогою методів штучного інтелекту, а саме алгоритму машинного навчання з підкріпленням Proximal Policy Optimization (Далі – PPO). Для досягнення цієї мети поставлено такі завдання:

- Розробити середовище для симуляції задачі, незалежно від алгоритму вирішення
- Розробити спосіб візуалізації середовища
- Розробити аналітичний алгоритм для вирішення задачі
- Розробити легкий спосіб створення нейромережі
- Показати працездатність програмного засобу на малих об'ємах даних

Об'єкт, методи й засоби розроблення. Об'єктом розроблення програмного засобу є симуляція розв'язування VRP за допомогою PPO, та порівняння з аналітичними результатами.

Під час розробки програмного продукту використовувалась Ощадлива модель заснована на таких принципах:

- Акцент на навчанні. Краще розробити простішу програму, використовуючи нові (для розробника) технології, ніж в котрий раз користуватись перевіреними методами, але досягти кращих результатів.
- Якомога відстрочене прийняття рішень. Рішення слід приймати не на основі припущень і прогнозів, а після відкриття істотних фактів.
- Короткі ітерації. Частий зв'язок з науковим керівником забезпечує високу мотивацію та допомогу при виникненні будь-яких затримок.
- Висока мораль. Головне не відбити бажання працювати над проектом, та підтримувати мотивацію, навіть якщо страждає ефективність.

В якості інструменту створення програмного засобу було обрано PyCharm Community edition 2023.1 – інтегроване середовище розробки (IDE) мовою

програмування Python, яке є безкоштовним, вільно поширюваним, з відкритим вихідним кодом.

Переваги Python полягають у його широкому спектрі бібліотек і фреймворків, які охоплюють різні сфери, серед них – аналіз даних і машинне навчання. Також він може похвалитися міжплатформною сумісністю, що дозволяє безперешкодно запускати код у різних операційних системах. Простота Python, високорівневі структури даних і велика стандартна бібліотека сприяють його продуктивності та швидкості розробки, дозволяючи ефективно писати код і виконувати складні завдання з меншою кількістю рядків коду.

Можливі сфери застосування. Програмний може застосовуватись для розробки нових алгоритмів руху бригад екстреної (швидкої) медичної допомоги, вантажних перевезень, кур’єрів служб доставки тощо.

1 ВИБІР ЗАДАЧІ ДЛЯ СИМУЛЯЦІЇ

1.1 Розгляд вже існуючих задач, та їх вирішення

Обрання задачі для симуляції є складним рішенням. Занадто проста – і аналітичний алгоритм може бути надто оптимальним для того щоб його міг обійти алгоритм побудований штучним інтелектом. Так банальна доставка вантажу з точки А в точку Б по зваженому графу, хоч і є складною з обчислювальної точки зору, але ефективність найкоротшого шляху, побудованого наприклад пошуком A* (читається «А зірочка» або англ. «A star»), можна тільки відтворити, але не перевершити, хоча, можливо і за коротший час. З іншої сторони занадто складна задача або зробить аналітичне рішення непрацездатним, або розтягне процес розробки і тренування на невідомий строк. Наприклад симуляція цілого міста з трафіком в залежності від часу доби та дня тижня, попиту в залежності від вдоволення міщан, та пропозиції в залежності від економічних показників, хоч і звучить цікаво та захоплююче, потребує дуже багато обчислювальних ресурсів та часу розробки.

Для того щоб обрати найкращий варіант, розглянемо декілька варіантів VRP:

- Звичайна VRP, яка звучить наступним чином: Який оптимальний набір маршрутів для парку транспортних засобів для доставки певному набору клієнтів?
- VRP з прибутком (VRPP) – проблема максимізації, коли відвідування всіх клієнтів не є обов'язковим. Мета полягає в тому, щоб відвідати клієнтів один раз, максимізуючи суму зібраного прибутку, дотримуючись обмеження часу транспортного засобу. Автомобілі повинні починати і закінчувати в депо. Існує багато підвидів цієї задачі, але тут ми розглядати їх не будемо.
- VRP з отриманням і доставкою (VRPPD) – деякі товари потрібно перемістити з певних місць отримання в інші місця доставки. Мета

полягає в тому, щоб знайти оптимальні маршрути для парку транспортних засобів для відвідування місць посадки та висадки.

- VRP з LIFO – те-ж саме що і VRP з отриманням і доставкою, за винятком додаткових обмежень щодо завантаження транспортних засобів: у будь-якому місці доставки товар, який доставляється, має бути останнім забраним. Ця схема скорочує час завантаження та розвантаження в місцях доставки, оскільки немає необхідності тимчасово вивантажувати товари, окрім тих, які потрібно вивантажити.
- VRP із часовими вікнами (VRPTW) – місця доставки мають часові вікна, протягом яких мають бути здійснені доставки (або відвідування).
- Ємна VRP (CVRP) – транспортні засоби мають обмежену вантажопідйомність щодо вантажів, які необхідно доставити.
- VRP з кількома поїздками (VRPMT) – транспортні засоби можуть виконувати декілька маршрутів.
- Відкрита VRP (OVRP) – транспортні засоби не зобов'язані повертатися в депо.
- Багато складська VRP (MDVRP) – існує кілька депо, з яких транспортні засоби можуть відправлятися та зупинятися.
- VRP з трансферами (VRPWT) – товари можна передавати між транспортними засобами в спеціально відведених перевантажувальних вузлах.

Важко одразу сказати яка задача підходить найкраще. Усі вони цікаві, але кожна презентує переваги і недоліки, тому треба розібратись з кожною з них окремо.

1.1.1 VRP

Класична задача VRP є дуже відомою та гарно дослідженою проблемою оптимізації. В цьому і є проблема для нашої симуляції, бо вже існують аналітичні алгоритми (хоча вони і не є ефективними за часом обчислення), які можуть її

вирішити оптимально. Перед тим як ми розглянемо ці алгоритми, спочатку треба математично змоделювати задачу одним з трьох найпопулярніших підходів:

1. Систематична організація руху транспортних засобів (Vehicle flow formulations) – тут використовуються цілі змінні, пов'язані з кожною дугою, які підраховують скільки разів транспортний засіб проходить ребро. Зазвичай використовується для базових VRP. Це добре для випадків, коли вартість рішення можна виразити як суму будь-яких витрат, пов'язаних з дугами. Однак його не можна використовувати для багатьох практичних застосувань.
2. Систематична організація руху товару (Commodity flow formulations) – додаткові цілочисельні змінні пов'язані з дугами або ребрами, які представляють потік товарів уздовж шляхів, якими рухаються транспортні засоби. Це нещодавно було використано для пошуку точного рішення.
3. Задача про розбиття (Set partitioning problem) – тут маємо експоненціальну кількість двійкових змінних, кожна з яких пов'язана з іншою можливою схемою. Тоді VRP формулюється як задача про розбиття, яка запитує, яка сукупність ланцюгів з мінімальною вартістю задовольняє обмеження VRP. Це враховує дуже загальні витрати на маршрут.

Щоб не поглиблюватись у деталі надто сильно, розглянемо перший варіант. Для цього розширимо задачу Комівояжера, яка полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу, сформульовану Джорджом Данцігом, Делбертом Реєм Фалкерсоном та Селмером Джонсоном, щоб створити формулювання двох індексних потоків транспортних засобів для VRP.

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

За умови що:

$$\sum_{i \in V} x_{ij} = 1, \qquad \forall j \in V\backslash\{0\} \tag{1}$$

$$\sum_{j \in V} x_{ij} = 1, \qquad \forall i \in V\backslash\{0\} \tag{2}$$

$$\sum_{i \in V\{0\}} x_{i0} = K \tag{3}$$

$$\sum_{j \in V\{0\}} x_{0j} = K \tag{4}$$

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S), \qquad \forall S \subseteq V\{0\}, S \neq \emptyset \tag{5}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall i, j \in V \tag{6}$$

Де c_{ij} є вартість переходу від вузла i до вузла j, x_{ij} є бінарною змінною, яка має значення 1 якщо дуга що йде від вузла i до вузла j є частиною рішення, інакше вона має значення 0, K є кількістю доступних транспортних засобів, $r(S)$ відповідає мінімальній кількості транспортних засобів, необхідних для обслуговування набору S, ми також припускаємо що вузол 0 є нашим депо.

Обмеження 1 і 2 стверджують, що рівно одна дуга входить і рівно одна виходить з кожної вершини, пов'язаної з клієнтом, відповідно. Обмеження 3 і 4 говорять про те, що кількість транспортних засобів, які виїжджають із депо, є такою ж, як кількість, що в'їжджає. Обмеження 5 – це обмеження пропускної спроможності, які накладають на те, що маршрути повинні бути з'єднані і що попит на кожному маршруті не повинен перевищувати пропускну здатність транспортного засобу. Нарешті, обмеження 6 є обмеженнями цілісності.

Такі, а також інші формулювання VRP можна вирішити оптимально за допомогою наступних підходів:

- Цілочисельне лінійне програмування (ILP) – VRP сформульовану як задачу ILP можна розв'язати за допомогою програм оптимізації. Цей метод підходить під задачу сформульовану як задачу про систематичну

організацію руху транспортних засобів, де ми визначили змінні рішення, обмеження та цільові функції, щоб змоделювати проблему математично.

- Метод гілок і меж – це техніка, яка систематично досліджує простір пошуку можливих рішень шляхом розгалуження на підпроблеми та обмеження цільової функції. Метод гілок і меж можна використовувати в поєднанні з формулюванням ILP для пошуку оптимального рішення. Він передбачає рекурсивний поділ проблеми на менші підпроблеми та використання меж для скорочення гілок, які гарантовано є неоптимальними.
- Динамічне програмування – динамічне програмування розбиває проблему на менші підпроблеми та вирішує їх рекурсивно, створюючи оптимальне рішення. Однак застосовність динамічного програмування обмежена конкретними випадками проблеми.

Окрім суто аналітичних алгоритмів, існує багато метаевристичних методів розв’язання VRP. Вони особливо корисні для цього, оскільки VRP, як відомо, є NP-складною, що означає, що пошук оптимального рішення за розумний час є обчислювально неможливим для великих екземплярів проблеми. До часто використовуваних метаевристичних алгоритмів для VRP вносять:

- Генетичні алгоритми – натхненні принципами еволюції та генетики, вони використовують популяцію кандидатів на рішення (хромосоми) і ітеративно застосовують такі генетичні оператори, як відбір, кросинговер і мутацію, щоб створити нове потомство. Оцінка придатності визначає якість рішень, і процес триває до конвергенції.
- Мурашиний алгоритм – ґрунтуючись на поведінці мурашок, які шукають їжу, цей метод використовує популяцію штучних мурах, які залишають сліди феромонів на маршрутах. Рівень феромонів впливає на вибір маршрутів іншими мурахами. Механізми навчання з підкріпленням в мурашиному алгоритмі покращують використання хороших рішень і дослідження простору рішень.

- Алгоритм імітації відпалу – натхненний процесом відпалу в металургії, цей алгоритм імітує охолодження матеріалу для зменшення дефектів. Він досліджує простір рішень, приймаючи гірші рішення з більшою ймовірністю на початку та поступово зменшуючи ймовірність з часом. алгоритм імітації відпалу збалансовує розвідку та експлуатацію, щоб не потрапити в пастку місцевих оптимумів.
- Табу-пошук – зберігає короткочасну пам'ять про нещодавно відвідані рішення, щоб уникнути їх повторного перегляду, цей лист відомий як «табу-список». Він досліджує околиці поточного рішення шляхом застосування збурень і рухів у просторі пошуку. Табу-пошук також пропонує стратегії диверсифікації, щоб уникнути локальних оптимумів.
- Метод рою часток – натхненний колективною поведінкою пташиних або рибних зграй, цей метод включає рій частинок, які рухаються через простір рішень. Кожна частинка представляє потенційне рішення, і вони коригують свій рух на основі власного досвіду та найвідомішого рішення в рої.

Ключовою перевагою метаевристичних алгоритмів є їх здатність ефективно досліджувати великі простори рішень і надавати якісні рішення для екземплярів VRP, навіть якщо вони можуть не гарантувати знаходження глобального оптимуму. Вони гнучкі, надійні та можуть обробляти динамічні або реальні варіації проблеми. Крім того, метаевристичні алгоритми можна комбінувати або гібридізувати, щоб використовувати сильні сторони різних підходів і покращувати їх загальну продуктивність.

Таким чином, можливо зробити висновок, що звичайна VRP вже є достатньо вивченою, та не є багато сенсу намагатися винайти алгоритми, які будуть краще ніж представлені, частково тому що ми будемо шукати не швидші, а якісніші рішення.

1.1.2 VRP з прибутком (VRPP)

VRP з прибутком (VRPP) — це варіант класичної VRP, який включає в себе поняття прибутку або доходу, пов'язаного з кожним відвідуванням клієнта. Це передбачає визначення оптимальних маршрутів для парку транспортних засобів для обслуговування набору клієнтів, максимізуючи загальний отриманий прибуток.

У VRPP кожен клієнт пов'язаний із значенням прибутку, яке представляє дохід, отриманий від відвідування цього клієнта. Мета полягає в тому, щоб знайти набір маршрутів, які відвідують усіх клієнтів, враховуючи обмеження пропускної спроможності транспортних засобів, обмеження часових вікон клієнтів і максимізуючи загальний отриманий прибуток.

Проблему можна визначити як набір ввідних та вихідних даних. Серед ввідних найчастіше є:

- Набір клієнтів: кожен клієнт може мати місцезнаходження, значення прибутку та часовий проміжок, що визначає час, протягом якого клієнт може бути обслугований.
- Парк транспортних засобів: кожен транспортний засіб може мати обмеження місткості та початкове місце.
- Матриця відстаней: матриця, яка представляє відстані між будь-якою парою місць.

Серед вихідних даних найчастіше є:

- Маршрути: набір маршрутів із зазначенням послідовності відвідувань клієнтів кожним транспортним засобом.
- Розклад: розклад із зазначенням часу прибуття до кожного клієнта з урахуванням обмежень у часовому вікні.
- Загальний прибуток: максимальний загальний прибуток, отриманий від відвідування всіх клієнтів.

Можна легко побачити що ця задача є дуже близькою до класичної VRP, має ті ж самі методи рішення, і звідси, ті самі проблеми.

1.1.3 VRP з отриманням і доставкою (VRPPD)

VRP з отриманням і доставкою (VRPPD) — це варіант класичної VRP, який передбачає доставку товарів із місць отримання до відповідних місць доставки за допомогою парку транспортних засобів. Це вимагає визначення оптимальних маршрутів для транспортних засобів, мінімізуючи загальну відстань або час подорожі.

У VRPPD існує два типи місць: місця отримання та місця доставки. Кожне місце отримання вказує на походження певної кількості товарів, а кожне місце доставки вказує на призначення цих товарів. Мета полягає в тому, щоб знайти набір маршрутів, які забирають товари з місць отримання, доставляють їх у відповідні місця доставки та задовольняють усі обмеження, мінімізуючи загальну відстань або час подорожі.

Проблему можна визначити як набір ввідних та вихідних даних. Серед ввідних найчастіше є:

- Набір місць отримання: кожне місце отримання може мати пропозицію (кількість товару), вікно часу отримання, відповідне місце доставки тощо.
- Набір місць доставки: кожне місце доставки може мати попит (кількість товару), вікно часу доставки тощо.
- Парк транспортних засобів: кожен транспортний засіб може мати обмеження місткості та початкове місце.
- Матриця відстаней: матриця, яка представляє відстані між будь-якою парою місць.

Серед вихідних даних найчастіше є:

- Маршрути: набір маршрутів із зазначенням послідовності місць отримання та доставки, відвіданих кожним транспортним засобом.

- Розклад: розклад із зазначенням часу прибуття в кожне місце отримання та доставки з урахуванням обмежень у часовому вікні.
- Результат: мінімальна загальна відстань або час, пройдений транспортними засобами.

Така задача може здатися значно більш цікавою з алгоритмічної точки зору, але ми можемо представити місця отримання та місця доставки не як дві різні точки, а як один відрізок, який ми обов’язково повинні пройти, і тоді ми фактично маємо класичну VRP, просто з відрізками, замість точок. Тобто ця задача нам також не підходить.

1.1.4 VRP з LIFO

VRP з LIFO, або VRP з принципом "останнім прийшов – першим пішов" — це варіант VRP з отриманням і доставкою (VRPPD), який запроваджує політику "останнім прийшов – першим пішов" (LIFO) для обслуговування клієнтів.

У цій задачі порядок відвідування клієнтів має значення, оскільки останній клієнт має обслуговуватися першим. Мета полягає в тому, щоб знайти оптимальні маршрути для транспортних засобів, враховуючи обмеження пропускної здатності, часові вікна та мінімізуючи загальну відстань або час у дорозі.

Набір ввідних та вихідних даних є аналогічним до VRP з отриманням і доставкою (VRPPD), так як ми лише додаємо обмеження, а не змінюємо задачу цілком.

Не дивлячись на те що ця задача є більш складною, ми так само можемо звести її до моделі цілочисельного лінійного програмування. Її доведення є за межами цього проекту, але ви можете ознайомитись з цією проблемою більш детально у статті «Pickup and delivery problem with LIFO, time duration, and limited vehicle number», від Andriansyah Andriansyah, Nissa Prasanti та Prima Denny Sentia [1].

1.1.5 VRP із часовими вікнами (VRPTW)

VRP з часовими вікнами (VRPTW) — це варіант класичної VRP, яка вводить обмеження часових вікон для відвідувань клієнтів. Це передбачає визначення оптимальних маршрутів для парку транспортних засобів для обслуговування набору клієнтів у визначені часові вікна, мінімізуючи загальну відстань або час подорожі.

Хоча ця задача рідко зустрічається самостійно, але часові обмеження часто додають до інших варіантів VRP, так серед вхідних даних як у VRPP, так і VRPPD вже вказані часові вікна як частина набору клієнтів та місць доставки відповідно, так само і в майбутніх варіаціях проблеми, обмеження часу буде частою додатковою умовою.

Звідси можна зробити висновок, що хоча ідея брати VRPTW як основну задачу для доведення працездатності програми не є вдалою, введення додаткового обмеження може бути цікавим доповненням.

1.1.6 Ємна VRP (CVRP)

Ємна VRP (CVRP)— це варіант класичної VRP, яка потребує врахування при цьому обмеження пропускної здатності транспортних засобів. Мета полягає в тому, щоб мінімізувати загальну відстань або вартість подорожі, задовольняючи попит усіх клієнтів і дотримуючись обмежень місткості транспортних засобів.

Так само як і VRPTW, цю задачу майже неможливо зустріти, окрім як доповнення до іншої задачі, тож і висновок є тим самим – додаткове обмеження ємності транспортного засобу може бути ще одним обмеженням, але не головною проблемою задачі.

1.1.7 VRP з кількома поїздками (VRPMT)

VRP з кількома поїздками (VRPMT) – це поєднання класичної VRP, VRPTW, та CVRP, з додатковою умовою що є можливість робити декілька поїздок, метою також є поєднання цих трьох задач, вона полягає в тому, щоб знайти оптимальні маршрути для транспортних засобів, враховуючи обмеження пропускної

здатності, часові вікна та мінімізуючи загальну відстань або час подорожі, дозволяючи транспортним засобам здійснювати кілька поїздок, якщо це необхідно.

Це додає дуже важливу особливість: збільшення кількості виконаних замовлень без збільшення розміру графа чи спрощення обмежень, але це ще не дозволяє нам вирішувати VRPMT в безперервному середовищі, так як кількість замовлень в нас все ще задана.

Якщо вирішити проблему обмеженості середовища, то VRPMT є досить гарним кандидатом як задача для симуляції, через свою комплексність та потенційну неперервність, що дозволить краще навчати нашу модель.

1.1.8 Багато складська VRP (MDVRP)

Багато складська VRP (MDVRP) є розширенням класичної VRP, де розглядаються декілька депо замість одного депо. У MDVRP є кілька складів, у кожному з яких є набір транспортних засобів і набір місць розташування клієнтів, які потрібно обслуговувати. Мета полягає в тому, щоб знайти оптимальні маршрути для транспортних засобів від депо до клієнтів, мінімізуючи загальну вартість або пройдену відстань, задовольняючи різноманітні обмеження.

У MDVRP ми поєднуємо декілька підходів із класичного VRP та VRPPD, за бажанням додаючи обмеження з інших варіацій, таких як VRPTW, CVRP тощо. Це дозволяє алгоритмічно ускладнити задачу, без накладання додаткових обмежень.

Проблему можна визначити як набір ввідних та вихідних даних. Серед ввідних найчастіше є:

- Набір депо: кожне депо може мати пропозицію (кількість товару), відповідне місце доставки тощо.
- Набір місць доставки: кожне місце доставки може мати попит (кількість товару), вікно часу доставки тощо.
- Парк транспортних засобів: кожен транспортний засіб може мати обмеження місткості та початкове місце.

- Матриця відстаней: матриця, яка представляє відстані між будь-якою парою місць.

Серед вихідних даних найчастіше є:

- Маршрути: набір маршрутів із зазначенням послідовності місць отримання та доставки, відвіданих кожним транспортним засобом.
- Розклад: розклад із зазначенням часу прибуття в кожне місце отримання та доставки з урахуванням обмежень у часовому вікні.
- Результат: мінімальна загальна відстань або час, пройдений транспортними засобами.

Легко побачити паралелі з VRPPD, і справді, фактично ця задача має ті самі умови, тільки замість багатьох місць отримання, ми маємо декілька депо, що додає рівень складності, але не робить задачу нерозв'язною.

1.1.9 VRP з трансферами (VRPWT)

VRP з трансферами (VRPWT) – це варіант класичної задачі маршрутизації транспортного засобу (VRP), яка передбачає перенесення вантажів з одного транспортного засобу на інший у певних пунктах пересадки. У VRPWT метою є визначення оптимальних маршрутів для парку транспортних засобів для обслуговування набору клієнтів, мінімізуючи загальну відстань або час подорожі, враховуючи обмеження пропускної здатності, часові вікна та пересадки між транспортними засобами.

Як і в інших задачах такого типу, у VRPWT кожен клієнт має попит (кількість товару), який повинен бути задоволений шляхом відвідування клієнта. Транспортні засоби мають обмежену здатність перевозити вантажі, і їм може знадобитися перевантажувати вантажі з одного автомобіля на інший у визначених пунктах перевантаження. Мета полягає в тому, щоб знайти набір маршрутів для транспортних засобів, які задовольняють попит усіх клієнтів, дотримуються обмеження пропускної спроможності та часового вікна, мінімізують загальну

відстань або час подорожі та визначають оптимальні точки пересадки та суми пересадок між транспортними засобами.

Проблему можна визначити як набір ввідних та вихідних даних. Серед ввідних найчастіше є:

- Набір клієнтів: кожен клієнт має місцезнаходження, попит і часовий проміжок, що визначає діапазон часу, протягом якого клієнт може обслуговуватися.
- Парк транспортних засобів: кожен транспортний засіб має обмеження місткості, початкове місце та максимальну відстань або часове обмеження.
- Пункти пересадки: місця, де можуть відбуватися пересадки між транспортними засобами.
- Матриця відстані: матриця, яка представляє відстані між будь-якою парою місць.

Серед вихідних даних найчастіше є:

- Маршрути: набір маршрутів із зазначенням послідовності відвідувань клієнтів кожним транспортним засобом, враховуючи пересадки між транспортними засобами.
- Розклад: розклад із зазначенням часу прибуття в кожне місце клієнта, включно з пунктами пересадки, враховуючи обмеження часових вікон.
- План трансферу: Визначення оптимальних точок трансферу та сум трансферу між автомобілями.
- Результат: мінімальна загальна відстань або час, пройдений транспортними засобами.

Вирішення VRPWT передбачає розгляд передачі вантажів між транспортними засобами. Це ускладнює проблему та вимагає спеціальних алгоритмів, які можуть ефективно обробляти перекази. Саме тому, якби ми були б обмежені вже існуючими задачами, ця напевно, була б найбільш практичною, але нашою задачею не є вирішення однієї практичної задачі, а створення інструменту

для симуляції багатого спектру таких задач, саме тому нам потрібно розробити свою проблему, а не брати вже існуючу.

1.2 Аналіз умов нової задачі для симуляції

Для початку треба зрозуміти які умови та обмеження та умови потрібні для доведення працездатності симуляції. Згідно з ощадливою моделлю розробки, потрібно концентруватись на створенні простого, рішення, але такого який буде покривати великий спектр можливостей, тому для початку розглянемо ті обмеження та умови які були зустріті під час розглядання вже існуючих задач:

1. Структура графу
2. Кількість кур'єрів
3. Кількість депо або кількість місць отримання
4. Кількість місць доставки
5. Обмеження ємності кур'єрів
6. Обмеження часу запитів
7. Прибуток за кожен виконаний запит
8. Принцип «останнім прийшов – першим пішов»
9. Можливість робити кілька поїздок
10. Можливість робити трансфери на спеціальних пунктах

Розглянемо кожен пункт окремо:

1. Нема необхідності надмірно ускладнювати структури графу. Він може бути неорієнтовним, не планарним, із випадковими вагами у визначених рамках.
2. Кількість кур'єрів буде статичною, та фактично гіперпараметром симуляції. Звісно є можливість ускладнення, та зміни кількості кур'єрів програмними методами, але це не є необхідним.
3. Кількість депо, так само як і кількість кур'єрів, буде статичною. Для задачі було обрано саме депо, ніж місця отримання. Це рішення буде пояснене пізніш.

4. Кількість місць доставки, а також їх розташування, є дуже важливими параметрами, так як в класичній VRP, та в її варіаціях, це є статичним показником, та головним фактором в визначені кінця роботи програми, але в багатьох реальних та практичних логістичних задачах ми не маємо чіткої кількості замовлень, так як вони виникають з плином часу. Наприклад виклики бригад екстреної (швидкої) медичної допомоги не є передбачуваними, так само як і онлайн-замовлення доставок товарів, тому буде краще якщо буде функціонал додавання місць доставки під час симуляції. Також це додає можливість робити симуляцію неперервною, що дозволить зменшити фактор таких гіперпараметрів як початкове розташування кур'єрів, запитів тощо.
5. Обмеження Ємності кур'єрів є ключовим параметром, без якого неможливо ефективно зробити задачу неперервною, бо кур'єри будуть брати нескінченну кількість товарів, що, очевидно, не є реалістичним. Для відповідності парадигмі, можна задати цю ємність як одиницю, це зробить процес навчання значно легшим, при цьому не роблячи задачу непрактичною.
6. Обмеження часу запитів може заплутати модель, але без цього, практичність є під великим питанням, тому було вирішено зійтись на компромісі: безпосередньо часового вікна не буде, але під час навчання будуть нараховуватись штрафи в залежності від часу роботи програми.
7. Прибуток за кожен виконаний запит є тим параметром на якому ґрунтується уся модель, адже саме він є винагородою для нашого алгоритму навчання з підкріпленням. Для відповідності ощадливої моделі розробки, можна задати цей прибуток як одиницю.
8. Принцип «останнім прийшов – першим пішов» буде зайвим ускладненням до усіх алгоритмів, та зробить навчання неймовірно повільним, тому цей принцип не буде входити в межі програми.
9. Можливість робити кілька поїздок є необхідною умовою для працездатності алгоритмів, особливо за умови ємності що дорівнює

одиниці, також це буде дуже важливим кроком для того щоб зробити програму неперервною.

10. Можливість робити трансфери на спеціальних пунктах є дуже цікавою темою, але нажаль виходить за рамки допустимого ощадливою моделлю розробки, бо додає дуже великий рівень складності алгоритмів, потребує мати різні види кур'єрів, та бажано буде розробити неочевидний метод винагороди за доставку вантажів до трансферних пунктів.

Також, окрім зазначених вище пунктів, є сенс розглянути декілька умов та можливостей, які не є частиною існуючих різновидів VRP, але є необхідними для розглядання, якщо ми хочемо зробити якісну та легко навчену модель. Усі ці умови стосуються однієї цілі: як зробити програму неперервною? Це потрібно для багатьох причин, частина з яких були розглянуті в пункті 4. Кількість місць доставки, а також їх розташування в списку вище. Серед цих причин можна виділити такі як зменшення впливу деяких гіперпараметрів, та наближення до практичних задач.

Для того щоб зробити задачу неперервною, головне що нам потрібно зробити, це додати спосіб генерування запитів, який буде простим, але досить реалістичним. На думку спадає два метода: підтримка статичної кількості замовлень, та випадкова генерація замовлень.

Підтримка статичної кількості замовлень є дуже простим, але дієвим методом. Все що ми робимо, це коли кур'єр виконує замовлення, ми одразу робимо ще одне на випадковій вершині графа. Це дозволяє дуже гарно контролювати середовище, але в цьому і проблема, адже з таким методом в нас ніколи не буде нуль замовлень, чи навпаки, дуже багато. Також за цих умов алгоритмічно дуже просто довести таку задачу до дуже оптимального рівня, адже ми можемо просто рухатись завжди до найближчого замовлення, яке ще не виконується іншим кур'єром.

Випадкова генерація замовлень з іншої сторони є набагато більш цікавим, але і більш складним методом. Імплементація звучить наступним чином: кожної

«миті» програми в кожної вершини графа є вірогідність замовити товар. Цей метод дозволяє не тільки покрити ситуації, коли в нас є різна кількість замовлень у графі, а і уникнути перенавчання, в обмін на необхідність знайти таку вірогідність, щоб швидкість утворюваних замовлень приблизно співпадала зі швидкістю їх виконання.

Серед цих двох методів, випадкова генерація є більш привабливою опцією, тому обрано було саме її. І тепер, можна сформулювати фінальну задачу, використовуючи яку ми будемо перевіряти працездатність нашої симуляції.

1.3 Умова задачі для симуляції

Є зважений граф з V вершинами, та вагами дуг w у межах $\{w|w \in \mathbb{N}, w \in [w_{min} = 25, w_{max} = 100]\}$, L з них – логістичні центри, де кур'єри можуть забирати товари. Сумарно на графі розташовано K кур'єрів, які за одну мить (англ. tick) програми проходять $\frac{1}{w}$ відстані кожного ребра, та можуть переносити $c = 1$ товар одночасно. Кожної миті програми, у кожної вершини графа є вірогідність p сформувати запит. Щоб виконати запит з вершини $d \in V$, один з кур'єрів $k \in K$ повинен мати товар, який він може отримати діставшись до одного з логістичних центрів $l \in L$, і, маючи товар, дістатись d. Задача стоїть в побудові алгоритму, який буде мати найбільшу швидкість виконання запитів.

Основна перевага цієї задачі – складність знаходження оптимального аналітичного рішення, через фактор випадковості. Так для теоретично найкращого рішення нам потрібно буде розглядати усі можливі комбінації усіх можливих виникнень замовлень на кілька тисяч митей наперед, що, очевидно не є практичним. Тому ми будемо використовувати евристичні та метаевристичні методи для її вирішення.

2 ВИКОРИСТАНІ АЛГОРИТМИ ТА ЇХ ІМПЛЕМЕНТАЦІЯ

2.1 Попередній розрахунок

Під час підготовки до виконання будь-яких алгоритмів, початковий крок включає генерацію випадкового графа та подальше визначення його матриці відстані. Цей важливий процес дозволяє нам кількісно визначити відстані між кожною парою вузлів на графіку та є одним з вхідних даних до більшості варіантів VRP.

2.1.1 Створення випадкового графу

Створення випадкового графу відбувається шляхом, додавання вузлів один за одним і приєднуючи їх до існуючих вузлів. Це забезпечує його зв'язність. Після цього більше ребер додається між випадковими вершинами, щоб досягти бажаної загальної кількості ребер на графі. Загальний алгоритм виглядає наступним чином:

1. Користувач задає два параметри: загальну кількість вузлів на графі і бажана кількість ребер на графі.
2. Створюємо порожній граф
3. Додаємо до нього перший вузол, його номер дорівнює нулю
4. Якщо кількість створених вузлів відповідає заданій – переходимо до пункту 8
5. Додаємо новий вузол
6. З'єднуємо його з випадковим існуючим вузлом
7. Повертаємось до пункту 4
8. Якщо кількість ребер більше або відповідає бажаній – переходимо до пункту 14
9. Обираємо випадковий вузол
10. Створюємо список усіх сусідів обраного вузла
11. Обираємо вузол який не входить до списку сусідів, та не є вже обраним вузлом

12.З’єднуємо його з вузлом, обраним в пункті 9

13.Переходимо до пункту 8

14.Випадковий граф отримано, закінчуємо алгоритм

Імплементація графа за допомогою NetworkX, популярної та потужної бібліотеки, яка широко використовується для вивчення та аналізу складних мереж і графів. Вона забезпечує потужну та інтуїтивно зрозумілу структуру для створення, маніпулювання та аналізу мережевих структур, що робить її важливим інструментом у різних областях. NetworkX пропонує широкий набір алгоритмів, функцій та інструментів, якими ми будемо користуватись [6]. Її універсальність, простота використання та велика документація роблять NetworkX дуже важливим інструментом для проекту.

Нижче приведено код мовою python, який імплементує даний алгоритм:

```
# Ця функція повертає випадковий граф із кількістю вузлів та ребер на графі.
def createRandomMapWithSetNodesAndEdges(amountOfNodes, amountOfEdges):
    # Створюємо порожній граф
    map = nx.Graph()

    # Додаємо до нього перший вузол, його номер дорівнює нулю
    map.add_node(0)

    # Якщо кількість створених вузлів відповідає заданій - завершуємо цикл
    for newNode in range(1, amountOfNodes):
        # Додаємо новий вузол
        map.add_node(newNode)

        # З’єднуємо його з випадковим існуючим вузлом
        map.add_edge(newNode, random.randint(0, newNode - 1))

    # Якщо кількість ребер більше або відповідає бажаній - завершуємо цикл
    for _ in range(amountOfNodes - 1, amountOfEdges):
        # Обираємо випадковий вузол
        firstNode = random.randint(0, amountOfNodes - 1)

        # Робимо список усіх вузлів
        allNodes = list(range(0, amountOfNodes))

        # Робимо список сусідів обраного вузла
        firstNodeNeighbors = list(map.adj[firstNode].keys())

        # Прибираємо обраний вузел зі списку усіх вузлів
        allNodes.pop(firstNode)

        # Робимо список усіх вузлів, які не є обраним вузлом, або його сусідами
        notNeighbors = [i for i in allNodes if i not in firstNodeNeighbors]

        # З'єднуємо обраний вузол із випадковим зі списку
        map.add_edge(firstNode, notNeighbors[random.randint(0, len(notNeighbors) - 1)])

    # Повертаємо отриманий граф
    return map
```

Після того як граф створено, потрібно для кожного ребра визначити випадкову вагу, та призначити колір, яким ми будемо демонструвати цю вагу при візуалізації графа, відповідно до ваги. Ми робимо це дуже простим шляхом: ітеруємо через усі ребра, графа, генеруємо випадкове число в заданих рамках, та в залежності від цього, додаємо ребру колір, як одну з властивостей.

Це реалізується у коді наступним чином:

```
# Ця функція встановлює випадкову вагу всім ребрам графа, та колір відповідно до ваги
def randomiseWeights(graph, minWeight, maxWeight):
    # Для усіх ребер
    for edge in list(graph.edges.keys()):
        # Генеруємо випадкову вагу
        weight = random.randint(minWeight, maxWeight)

        # Аналізуємо отриману вагу
        # Якщо вага є у перший третині області - обираємо зелений колір
        if weight <= (minWeight + (maxWeight-minWeight) / 3):
            color = 'green'
        # Якщо вага є у другій третині області - обираємо зелений жовтий
        elif weight <= (minWeight + 2*(maxWeight - minWeight) / 3):
            color = 'yellow'
        # Якщо вага є у третій третині області - обираємо червоний колір
        else:
            color = 'red'

        # Встановлюємо вагу та колір ребра відповідно
        graph[edge[0]][edge[1]]['weight'] = weight
        graph[edge[0]][edge[1]]['color'] = color
```

2.1.2 Розрахунок матрицю відстаней

Тепер, коли випадковий граф побудовано, потрібно розрахувати матрицю відстаней. Це буде робитися за допомогою алгоритму Дейкстри, який є широко використовуваний алгоритм для пошуку найкоротших шляхів між вузлами в графі. Він ефективно вирішує проблему найкоротшого шляху з одного джерела, ітеративно досліджуючи граф від початкового вузла до всіх інших вузлів, оновлюючи оцінки найкоротшого шляху на цьому шляху [2].

Загальний алгоритм виглядає наступним чином:

1. Створюється набір для зберігання невідвіданих вузлів.
2. Значення відстані для кожного вузла, спочатку встановленого на нескінченність, за винятком початкового вузла, який встановлений на 0.

3. Для поточного вузла розглядаються всі його сусідні вузли, які не були відвідані. Для кожного сусіда обчислюється орієнтовні відстань, додавши вагу поточного ребра до відстані поточного вузла.
4. Якщо ця орієнтовна відстань менша за попередньо записану відстань для сусіда, замінюємо стару відстань новою.
5. Після відвідування всіх сусідів поточного вузла, він позначається як відвіданий, та видаляється із набору невідвіданих вузлів.
6. Серед невідвіданих вузлів обирається вузол із найменшою орієнтовною відстанню та встановлюється як новий поточний вузол, а якщо невідвіданих вузлів немає, алгоритм завершено.
7. Повторюємо кроки з 3 по 6, поки не буде відвідано всі вузли або найменша орієнтовна відстань серед невідвіданих вузлів не стане нескінченністю. Якщо найменша орієнтовна відстань дорівнює нескінченності, це означає, що немає шляху від початкового вузла до решти невідвіданих вузлів.
8. Після завершення роботи алгоритму найкоротший шлях від початкового вузла до будь-якого іншого вузла можна знайти шляхом зворотного відстеження.

   Починаючи від вузла призначення, слідуйте по шляху зі зменшенням відстані до досягнення початкового вузла.

У бібліотеці networkx є функція single_source_dijkstra, яку ми будемо використовувати для реалізації цього алгоритму. Вона приймає в якості вхідних даних граф і початковий вузол і обчислює найкоротший шлях від цього вузла до всіх інших вузлів у графі. Функція повертає пару словників, де ключами є цільові вузли, а значення – найкоротша відстань шляху від початкового вузла, та безпосередньо найкоротший шлях. За допомогою функції single_source_dijkstra в нас э зручний і надійний метод пошуку найкоротших шляхів на графах.

Застосувавши алгоритм Дейкстри до усіх вузлів графа, ми отримуємо шукану матрицю відстаней, цей процес має часову складність:

$$\Theta(|V|(|E| + |V|\log|V|))$$

Де $|V|$ - кількість вершин, а $|E|$ - кількість ребер.

Це реалізується у коді наступним чином:

```
# Ця функція розраховує усі найкоротші шляхи у графі
def calculateAllShortestPaths(self):
    # Дістаємо граф із поля
    graph = self.fieldVisualiser.field.map

    # Підготовуємо змінну для запису результатів
    result = dict()

    # Для усіх вершин
    for i in range(graph.number_of_nodes()):
        # Використовуємо single_source_dijkstra
        length, path = networkx.single_source_dijkstra(graph, i)

        # Записуємо результат
        result[i] = (length, path)
    # Повертаємо результат
    return result
```

2.2 Аналітичний алгоритм розв'язання задачі

Хоча знаходження точного алгоритму розв'язання задачі є дуже складним завданням, існує досить очевидне рішення, яке буде показувати гарні результати. Воно полягає в тому, що кур'єр буде рухатись до найближчого депо чи запиту, до якого ще не йде кур'єр, в залежності від того чи несе він замовлення чи ні. Очевидно, що це не буде найбільш оптимальним рішенням, але воно дасть приблизну планку, до якої треба буде дотягнутись алгоритму машинного навчання з підкріпленням.

На вхід аналітичного алгоритму подається те-ж саме, що і буде подаватись на вхід нейромережі: масив, розмір якого дорівнює $5|V| + 1$, де $|V|$ - кількість вершин. Цей масив складається з п'яти частин розмірами $|V|$ кожна, та одного біту перевезення. Перша частина є нормовані відстанями до кожного вузла графу, друга частина – булеві індикатори приналежності вершин до запитів, третя – булеві індикатори приналежності вершин до депо, четверта – розташування кур'єрів, п'ята – цілі кур'єрів. Біт перевезення позначає чи перевозить кур'єр наразі замовлення, чи ні.

Перед тим як можна буде застосувати алгоритм, потрібно проаналізувати масив, підготувати дані для алгоритму та розрахувати поточне положення кур'єра. Для цього ми спочатку визначаємо чи містить початковий масив біт перевезення,

шляхом пошуку остачі ділення кількості елементів в масиві на п’ять. Якщо вона дорівнює одиниці – масив містить біт перевезення, інакше ні. Це робиться для цілі сумісності з старим кодом, в якому біту перевезення не існувало. Після визначення наявності біту перевезень, він виймається з масиву, та запам’ятовується, а решта ділиться на п’ять рівних частин, та записується як масив масивів. Поточне положення кур’єру відповідає індексу елементу зі значенням нуль в першому підмасиві, бо саме там зберігається нормовані відстані до усіх вузлів. Після цього, застосовується реалізація алгоритму відповідно до наявності біту перевезень.

Це реалізується у коді наступним чином:

```
# Ця функція готує дані та запускає відповідний алгоритм
def complexAnalyticalAgent(observation):
    # Перевіряємо наявність біту перевезень
    if len(observation) % 5 == 1:
        # Якщо він існує - прибираємо його з масиву, та запам'ятовуємо значення
        carryingBit = observation.pop(-1)
    else:
        # Якщо його не існує - запам'ятовуємо це
        carryingBit = -1
    # Розділяємо масив на п'ять підмасивів
    splited = [list(array) for array in np.array_split(observation, 5)]

    # Індекс елементу зі значенням нуль відповідає пото
    currentNode = splited[0].index(0)

    # Запускаємо алгоритм відповідно до наявності біту перевезень
    if carryingBit == -1:
        return Hivemind.ComplexAnalyticalWithoutCarryingBit(splited, currentNode)
    else:
        return Hivemind.ComplexAnalyticalWithCarryingBit(splited, currentNode,
                                                         carryingBit)
```

Сам алгоритм є дуже простим як в теорії, так і в реалізації. Тут і далі розглядається варіант з бітом перевезень, інший варіант можна проглянути в додатку А.

Алгоритм розділено на дві частини, в залежності від значення біту перевезень. Якщо кур’єр перевозить товар, то йде перебір усіх вузлів, та ті, на яких є запит, та які не є цілями інших кур’єрів, записуються як потенційні цілі. Після чого ми знаходимо такий вузол зі списку потенційних цілей, який є найближчим до поточної позиції кур’єра, та встановлюємо його як ціль.

Якщо ж кур'єр не перевозить товар, то ми так само перебираємо усі вузли, але тепер в якості потенційних цілей записуються ті, на яких є депо. Після чого ми повторюємо другий крок попередньої реалізації для знаходження цілі.

Це реалізується у коді наступним чином:

```
# Ця функція аналітично визначає ціль, для кур'єра
def ComplexAnalyticalWithCarryingBit(splited, currentNode, carryingBit):
    # Якщо кур'єр перевози ть товар
    if carryingBit == 1:
        # Підготовуємо дані
        possibleTargets = list()
        request = 0

        # Для усіх вузлів
        for isRequest in splited[1]:
            # Якщо на вузлі є запит, та він не є вже ціллю кур'єра
            if splited[4][request] == 0 and isRequest == 1:
                # Додаємо вузол як потенційну ціль
                possibleTargets.append(request)
            request += 1
        # Якщо потенційних цілей немає - кур'єр залишається на поточній позиції
        if not possibleTargets:
            return currentNode

        # Встановлюємо перший потенційний вузол як ціль
        target = possibleTargets[0]

        # Запам'ятовуємо відстань до цілі
        closest = splited[0][target]

        # Для кожної потенційної цілі
        for possibleTarget in possibleTargets:
            # Якщо відстань до потенційної цілі менше ніж до поточної
            if splited[0][possibleTarget] < closest:
                # Встановлюємо нову відстань та ціль
                closest = splited[0][possibleTarget]
                target = possibleTarget
        # Повертаємо ціль
        return target
    # Якщо кур'єр не перевозить товар
    else:
        # Підготовуємо дані
        possibleTargets = list()
        hub = 0

        # Для усіх вузлів
        for isHub in splited[2]:
            # Якщо на вузлі є депо
            if isHub == 1:
                # Додаємо вузол як потенційну ціль
                possibleTargets.append(hub)
            hub += 1

        # Встановлюємо перший потенційний вузол як ціль
        target = possibleTargets[0]

        # Запам'ятовуємо відстань до цілі
        closest = splited[0][target]

        # Для кожної потенційної цілі
        for possibleTarget in possibleTargets:
            # Якщо відстань до потенційної цілі менше ніж до поточної
            if splited[0][possibleTarget] < closest:
                # Встановлюємо нову відстань та ціль
                closest = splited[0][possibleTarget]
                target = possibleTarget
        # Повертаємо ціль
        return target
```

2.3 Регуляризація ентропії

Регулярізація ентропії — це техніка, яка використовується в навчанні з підкріпленням (RL), щоб стимулювати дослідження та покращувати процес навчання. У RL агент вчиться приймати послідовні рішення в середовищі, щоб максимізувати поняття сукупної винагороди. Поведінка агента керується політикою, яка є відображенням від станів до дій.

Ентропія розподілу ймовірностей вимірює невизначеність або випадковість, пов'язану з цим розподілом. У RL ентропія політики представляє ступінь дослідження або випадковості в діях агента. Висока ентропія означає, що агент виконує різноманітні дії, досліджуючи різні можливості. Низька ентропія вказує на те, що дії агента більш детерміновані та зосереджені на використанні відомих хороших дій.

Регулярізація ентропії дозволяє сприяти розвідці, зазвичай шляхом додавання ентропійного члена до цільової функції RL. Цільова функція зазвичай спрямована на максимізацію очікуваної кумулятивної винагороди, але з регуляризацією ентропії мета стає компромісом між максимізацією очікуваної винагороди та максимізацією ентропії політики.

Замість звичайної реалізації, ми імплементуємо свою. При низькому значенні сили регуляризації, вона буде тим більше збільшувати вірогідність дії, чим менше була її початкова вірогідність, що і потрібно.

Алгоритм приймає два параметри: множина P розміру $|P|$, яка представляє розподіл ймовірностей, і величину e, яка визначає силу регулярізації. Якщо величина менша або дорівнює 0 або більша за 1, алгоритм просто повертає вхідний розподіл, без виконання будь-якої регулярізації.

Створюється також множина з розподілом не ймовірностей, шляхом віднімання кожного елемента в масиву розподілу ймовірностей від 1. Далі створюється його нормалізована версія, шляхом ділення кожного елемента на суму усіх елементів.

Для кожного елемента оригінального розподілу, ймовірність змінюється шляхом множення його на $(1-e)$ і додавання значення, пропорційно його не вірогідності так, щоб сума фінального розподілу мало значення в одиницю. Це ефективно виконує регуляризацію ентропії шляхом перерозподілу ймовірностей на основі величини регуляризації. Строго це визначається наступним чином:

$$\forall i \in \mathbb{Z}, i \in [1, |P|], r_i \in R, p_i \in P:$$

$$\begin{cases} r_i = p_i * (1-e) + e\dfrac{(1-p_i)}{\sum_{j=1}^{|P|}(1-p_j)}, \qquad e \in (0,1] \\ r_i = p_i, \qquad e \notin (0,1] \end{cases}$$

Доведемо що сума елементів результуючої множини R дорівнює одиниці:

$$\sum_{i=1}^{|P|} r_i = \sum_{i=1}^{|P|}\left(p_i(1-e) + e\frac{(1-p_i)}{\sum_{j=1}^{|P|}(1-p_j)}\right) =$$

$$= \sum_{i=1}^{|P|} p_i(1-e) + \sum_{i=1}^{|P|} e\frac{(1-p_i)}{\sum_{j=1}^{|P|}(1-p_j)} =$$

$$= (1-e)\sum_{i=1}^{|P|} p_i + \frac{e}{\sum_{j=1}^{|P|}(1-p_j)}\sum_{i=1}^{|P|}(1-p_i) =$$

$$= (1-e) + e\frac{\sum_{i=1}^{|P|}(1-p_i)}{\sum_{i=1}^{|P|}(1-p_i)} = 1 - e + e = 1$$

Знайдемо при якій силі регуляризації результуючий розподіл буде випадковим, незалежно від вмісту оригінального розподілу:

$$\exists e \in (0,1]: \forall r_i, r_u \in R, r_i = r_u$$

$$p_i * (1-e) + e\frac{(1-p_i)}{\sum_{j=1}^{|P|}(1-p_j)} = p_u(1-e) + e\frac{(1-p_u)}{\sum_{j=1}^{|P|}(1-p_j)}$$

$$(p_i - p_u)(1-e) + (p_u - p_i)\frac{e}{\sum_{j=1}^{|P|}(1-p_j)} = 0$$

$$(p_i - p_u)\left(1 - e - \frac{e}{\sum_{j=1}^{|P|} 1 - \sum_{j=1}^{|P|} p_j}\right) = 0$$

$$(p_i - p_u)\left(1 - e - \frac{e}{|P| - 1}\right) = 0$$

$$(p_i - p_u)\left(\frac{|P| - 1 - e|P| + e - e}{|P| - 1}\right) = 0$$

$$(p_i - p_u)\left(\frac{|P| - 1 - e|P|}{|P| - 1}\right) = 0$$

$$(p_i - p_u)\left(\frac{1 - \frac{1}{|P|} - e}{|P| - 1}\right) = 0$$

$$\left[\begin{array}{l} p_i = p_u \\ e = 1 - \frac{1}{|P|} \end{array}\right.$$

Отже при значенні сили регуляризації що дорівнює різниці одиниці та оберненої кількості елементів в множині розподілу ймовірностей, результуючий розподіл ймовірностей буде рівномірним.

Це реалізується у коді наступним чином:

```
# Ця функція регуляризує розподіл ймовірностей, на основі сили регуляризації
def entropyRegularization(distTensor, magnitude):
    # Якщо сила регуляризації не відповідає крітерію, повертаємо початковий розподіл
    if magnitude <= 0 or magnitude > 1:
        return distTensor

    # Готуємо дані
    detached = distTensor.detach()
    arrayed = detached.numpy()
    successChances = arrayed.copy()

    # Рахуємо розподіл не ймовірності
    unsuccessfulChance = [1-chance for chance in successChances[0]]

    # Нормалізуємо розаоділ не ймовірності
    summ = sum(unsuccessfulChance)
    unsuccessfulChanceNormalized = [float(i) / summ for i in unsuccessfulChance]

    # Ітеруємо для кожного елементу розподілу
    i=0
    for chance in successChances[0]:
        # Зменшуємо вірогідність відповідно до сили регуляризації
        chance *= (1-magnitude)

        # Збільшуємо вірогідність відповідно до не ймовірності
        chance += unsuccessfulChanceNormalized[i]*magnitude

        # Зберігаємо результат
        detached[0][i] = chance
        i+=1
    return distTensor
```

2.4 Proximal Policy Optimization (PPO)

Теорія, яка стоїть за PPO, є дуже складною темою, що вимагає ретельного вивчення, представити яке тут немає змоги, але загальна концепція наступна: зазвичай алгоритми діяча-критика є дуже чутливими до будь-яких змін, і часто виникають ситуації, коли агент «падає зі скелі». PPO різними методами обмежує те, наскільки змінюється нейромережа, головним з яких є орієнтування на співвідношення нової політики до старої. Очевидно, так само як і в інших подібних алгоритмах, береться до уваги те, в наскільки гарному стані ми опинилися, величина яке це визначає називається перевагою. Це може створити проблему що функція втрат починає бути завеликою, тому в PPO ця функція підрізається [3, 4, 5].

Ми імплементуємо PPO використовуючи дві нейромережі, одна для діяча, інша для критика. Це не є найбільш оптимальним варіантом, але він оптимальний достатньо для нашої задачі. Критик буде оцінювати стани, а діяч обирати що робити відповідно до стану. На виході отримуємо розподіл ймовірностей, до якого застосовується регуляризація ентропії, що і забезпечує розвідку.

Нажаль у PPO багато гіперпараметрів, серед них довжина пам’яті, розмір партії, кількість епох і швидкість навчання, а також декілька інших. Тож навіть коли все зроблено, доведеться налаштовувати декілька параметрів вручну.

З математичної точки зору правило оновлення для діяча виглядає наступним чином:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

Де r_t це відношення нової політики до старої, та дорівнює $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$, $\hat{A}_t$ – перевага, ϵ – гіперпараметр визначаючий підрізання відношення політик. Ця реалізація дозволяє легко контролювати дальність кроків, що і береже модель від «падіння зі скелі».

Якщо трохи спростити визначення переваги, то вона оцінюється як сума нагород та оціночної ефективності наступного стану, де кожен наступний стан впливає все менше, щоб мотивувати робити більш рішучі дії:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$

$$\text{де } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Тут γ – знижка (зазвичай 0.99), λ – параметр згладжування (зазвичай 0.95) , $V(s_t)$ – оціночне значення стану.

Функція втрат критика визначається як середньоквадратична похибка оцінки стану критика на основі поточної нейромережі, та суми переваги з оцінкою стану критика з пам’яті.

Звичайно реалізація в коді є дуже комплексною, тому нижче приведено фрагменти коду, які імплементують вказані вище методи.

Підрахунок переваги:

```
for t in range(len(reward_arr) - 1):
    discount = 1
    a_t = 0
    for k in range(t, len(reward_arr) - 1):
        a_t += discount * (reward_arr[k] + self.gamma * values[k + 1] *
                           (1 - int(dones_arr[k])) - values[k])
        discount *= self.gamma * self.gae_lambda
    advantage[t] = a_t
advantage = T.tensor(advantage).to(self.actor.device)
```

Підрахунок функції втрат Діяча:

```
new_probs = dist.log_prob(actions)
prob_ratio = new_probs.exp() / old_probs.exp()
weighted_probs = advantage[batch] * prob_ratio
weighted_clipped_probs = T.clamp(prob_ratio, 1 - self.policy_clip,
                                 1 + self.policy_clip) * advantage[batch]
actor_loss = -T.min(weighted_probs, weighted_clipped_probs).mean()
```

Підрахунок функції втрат Критика:

```
returns = advantage[batch] + values[batch]
critic_loss = (returns - critic_value) ** 2
critic_loss = critic_loss.mean()
```

3 СТРУКТУРА ПРОГРАМИ, РЕАЛІЗАЦІЯ ЗАДАЧІ В ЇЇ МЕЖАХ

3.1 Огляд складових програмного засобу

Для вирішення поставленої задачі, потрібно реалізувати декілька напів-автономних частин. Головні з них це середовище, та колективний розум – прошарок між агентом та середовищем.

Середовище складається з настуаних частин:

1. Поле – об’єднує та зберігає більшість елементів середовища до передачі їх в калькулятор або візуалізатор.
2. Депо – статичні об’єкти розташовані на мапі, з яких кур’єри можуть забирати товари.
3. Кур’єри – рухомі об’єкти, які будуть відвозити товари в залежності від рішень колективного розуму.
4. Генератор запитів – функція що визначає правило створення запитів.
5. Запити – тимчасові об’єкти, які створюються в залежності від параметрів генератора, та видаляються після отримання товарів.
6. Калькулятор поля або Візуалізатор поля – головний менеджер середовища, який відповідає за спостереженням над полем, передачі наказів колективного розуму, ітерації середовища тощо.

Колективний розум складається з різних аналітичних алгоритмів, які можуть використовуватись для керування кур’єрами, але головна його задача, це визначення способів взаємодії середовища з агентом.

Окрім середовища та колективного розуму, наявні допоміжні елементи, які було розглянуто у главі 2.

3.2 Середовище, його складові та імплементація

3.2.1 Поле

При створенні поля потрібно вказати мапу, яка повинна бути графом з бібліотеки network, після чого, можна додавати пасивні депо та пасивних кур’єрів, так само як і генератори запитів.

Додавання пасивних кур’єрів записує дані необхідні для створення кур’єрів, але не створює кур’єрів напряму. Так само з пасивними депо. Також є функціонал для додавання одразу декількох кур’єрів, та депо на випадкові вузли мапи.

В коді це реалізується наступним чином:

```
class Field:
    def __init__(self, map):
        self.map = map
        self.passiveCouriers = list()
        self.passiveHubs = list()
        self.requestGenerators = list()

    # Додавання пасивного кур'єра записує необхідні дані для його створення
    def addPassiveCourier(self, courierName, aiName, node, oneStepBehind=False):
        self.passiveCouriers.append((courierName, aiName, node, oneStepBehind))

    # Додавання пасивного депо записує необхідні дані для його створення
    def addPassiveHub(self, node, services):
        self.passiveHubs.append((node, services))

    # Додавання генератора запитів дозволяє зручно їх зберігати
    def addRequestGenerator(self, requestGenerator):
        self.requestGenerators.append(requestGenerator)

    # Ця функція додає декілька пасивних депо на випадкові вузли мапи
    def addRandomPassiveHubs(self, amount, services):
        # Стільки разів, скільки необхідно депо
        for _ in range(amount):
            # Намагаємось знайти вузол, на якому ще нема депо
            while True:
                # Створюємо список усіх вузлів
                allNodes = list(self.map.adj.keys())

                # Обираємо серед них випадковий
                randomNode = allNodes[random.randint(0, len(allNodes) - 1)]

                # Якщо цього вузла нема серед тих на якому вже є депо - вузол знайдено
                if randomNode not in self.passiveHubs:
                    break
            # Додаємо пасивне депо
            self.addPassiveHub(randomNode, services)

    # Ця функція додає декілька пасивних кур'єрів на випадкові вузли мапи
    def addCouriers(self, courierName, aiName, amount, oneStepBehind=False):
        # Стільки разів, скільки необхідно кур'єрів
        for _ in range(amount):
            # Створюємо список усіх вузлів
            allNodes = list(self.map.adj.keys())

            # Обираємо серед них випадковий
            randomNode = allNodes[random.randint(0, len(allNodes) - 1)]

            # Додаємо пасивного кур'єра
            self.addPassiveCourier(courierName, aiName, randomNode, oneStepBehind)
```

3.2.2 Депо

При створенні депо вказується вузол, на якому воно розташоване, які товари воно надає, кількість часу, який потрібен щоб забрати товар, та два

необов’язкових параметри візуалізації, які використовуються тільки якщо потрібно показати депо на екрані. Серед функцій депо, першою є отримання товару, яке повертає успішність операції, та час, який це зайняло, а другою – видалення Депо.

В коді це реалізується наступним чином:

```
class Hub:
    def __init__(self, node, services, pos=None, pyplotAx=None, ticksToReceive=60):
        self.services = services
        self.node = node
        self.ticksToReceive = ticksToReceive

        # Дивимось чи вказані параметри необхідні для візуалізації
        if node is None or pos is None or pyplotAx is None:
            # Якщо ні - позначаємо що депо є неанімованим
            self.animated = False
        else:
            # Якщо так - позначаємо що депо є анімованим
            self.animated = True

            # Дізнаємось координати вузла, де знаходиться депо
            self.x = pos[node][0]
            self.y = pos[node][1]

            # Встановлюємо де буде намальоване депо
            self.ax = pyplotAx

            # Малюємо депо
            self.dot, = self.ax.plot(self.x, self.y, marker='o', color='black', zorder=0,
                                     lw=5, markersize=23, markeredgewidth=3)

    # Ця функція повертає успішність намагання забрати товар, та час який це зайняло
    def getService(self, serviceToGet):
        return serviceToGet in self.services, self.ticksToReceive

    # Ця функція видаляє депо
    def deleteRequest(self):
        self.dot.remove()
```

3.2.3 Кур’єр

Кур’єр складається з трьох частин: одна відповідає за рух кур’єра, друга за поведінку, третя поєднує дві попередні та зберігає деякі важливі дані. Головна задача частини яка відповідає за рух кур’єра, це рахувати скільки митей програми залишилось до кінця руху, другорядна – переміщати іконку кур’єра, якщо задані параметри візуалізації. Друга частина має змогу давати кур’єрам різні поведінки, але наразі вона виконує скоріше функцію посередника між кур’єром та колективним розумом. Третя частина є головною, в ній визначається чи є кур’єр анімованим, який формат спостереження він веде, чи перевозить він товар, тощо.

Головна функція третьої частини – ітерація кур’єра, де застосовується запропонована дія, віддається команда підрахунку маршруту, потім команда слідування маршруту, і наостанок команда отримання або доставки товару. Найбільш неочевидна частина, це команда видалення виконаних запитів, вона присутня для того щоб критик міг побачити, який стан приносить нагороду, тому потрібно щоб ця команда була віддана кур’єром після того як відбулося спостереження.

Найбільш важливі частини в коді реалізуються наступним чином:

```
class Courier:
    def __init__(self, courierName, aiName, pos=None, currentNode=0,
                 pyplotAx=None, hivemind=None, oneStepBehind=False,
                 fieldCalculator=None, id=0):
        # Калькулятор або візуалізатор поля, в якому знаходиться кур'єр
        self.fieldCalculator = fieldCalculator

        # Ім'я, яке ніде не використовується
        self.name = courierName

        # Перевірка чи є кур'єр анімованим, чи ні
        if pos is None or pyplotAx is None:
            self.courierMovement = CourierMovement()
        else:
            self.courierMovement = CourierMovement(pos[currentNode][0],
                                                   pos[currentNode][1], pyplotAx)
        # Поведінка кур'єра
        self.courierAi = CourierAi(aiName, self, currentNode, hivemind)

        # Товар, ячкий кур'єр перевозить
        self.carryingService = None

        # Формат видалення виконаних запитів
        self.oneStepBehind = oneStepBehind

        # Нагорода та Покарання
        self.reward = 0
        self.punishment = 0

        # Унікальний ідентифікаційний номер
        self.id = id

    def iterateCourier(self, action=None):
        # Якщо кур'єр не зайнятий
        if self.courierAi.freeze <= 0:
            # Ітеруємо "інтелект" кур'єра, пропонуємо дію, якщо вона є
            self.courierAi.iterateAi(action)

            # Якщо необхідно, кур'єр власноруч видаляє виконані запити
            if self.oneStepBehind:
                self.fieldCalculator.killRequests(self.id)

            # Прораховуєм команди для руху
            self.courierAi.hivemind.moveToFinalNode(self)

            # Рухаємо кур'єра
            courierMovement = self.courierMovement.iterateMovement()
            self.courierAi.endOfTheMoveCheck()

            # Отримуємо або доставляємо товар
            self.courierAi.provideOrGetService()
        else:
            # Зменшуємо кількість часу що залишилось бути зайнятим
            courierMovement = False
            self.courierAi.freeze -= 1

        # Повертаємо чи рухався кур'єр чи ні
        return courierMovement
```

3.2.4 Генератор запитів

Генератор запитів є дуже простою структурою, в якої задається шанс згенерувати запит та випадково обирається один можливий вид товару із списку. Реалізація також є примітивною:

```
class RequestGenerator:
    def __init__(self, chanceToSpawn, outOf, possibleServices):
        self.chanceToSpawn = chanceToSpawn
        self.outOf = outOf
        self.possibleServices = possibleServices

    # Ця функція створює запит з певною ймовірністю
    def generateRequest(self, node, pos=None, ax=None):
        # Генеруєм випадкове число та перевіряємо чи воно менше ніж необхідне
        if random.randint(1, self.outOf) <= self.chanceToSpawn:
            # Беремо випадковий товар зі списку
            service = self.possibleServices[random.randint(0,
                                                len(self.possibleServices)-1)]

            # Створюємо і повертаємо запит
            return Request(node, service, pos, ax)
```

3.2.5 Запити

Запити мають схожу структуру з депо, але замість видачі товару, вони його приймають, та повертають кількість митей, необхідних для обробки товару, або мінус один, якщо товар не замовлявся. Також з'явився параметр відстроченого видалення, який ідентифікує який кур'єр повинен видалити цей запит.

Таким чином розгляду заслуговує тільки одна функція серед запитів:

```
# Ця функція повертає час який зайняло доставка товару, або -1 якщо товар не підійшов
def receiveService(self, receivedService):
    if receivedService == self.serviceRequest:
        return self.ticksToServe
    else:
        return -1
```

3.2.6 Калькулятор поля або Візуалізатор поля

Калькулятор або Візуалізатор є найважливішою складовою середовища. Його головна задача – ітерація митей. Калькулятор і Візуалізатор мають декілька різниць, одна з яких очевидна з назви: калькулятор на має жодного графічного представлення, що дозволяє йому працювати на дуже великих швидкостях, в той

час як візуалізатор дає гарне картинку, але швидкість не дозволяє на ньому робити навчання.

Спочатку розглянемо функції спільні для обох варіантів. Існує декілька допоміжних функцій, які не потребують багато пояснень, чи пояснень взагалі, до них відносяться наприклад підрахунок усіх найкоротших шляхів, додавання запитів та депо вручну, тощо. Цікавою ж є функція спостереження. Вона приймає номер кур'єра як параметр, та створює лист зі списком нормованих дистанцій до усіх вузлів мапи з позиції кур'єра, та списками вузлів, на яких є запити, депо, кур'єри та цілями кур'єрів, закодовані унітарно для усіх вузлів, таким чином в кінці отримуємо список з розміром що дорівнює п'ятикратній кількості вузлів. На пізніх стадіях розробки було додано також біт перевезень, який вказує на те перевозить кур'єр товар чи ні.

В коді розрахунок спостереження ведеться наступним чином:

```
def observeForCourier(self, courierId):
    # Знаходимо кур'єра по номеру
    courier = self.activeCouriers[courierId]

    # Формуємо список найкоротших шляхів
    dictOfShortestPaths = self.allShortestPaths[courier.courierAi.currentNode][0]

    # Сортуємо список за номером вузлів
    myKeys = list(dictOfShortestPaths.keys())
    myKeys.sort()
    sorted_dict = {i: dictOfShortestPaths[i] for i in myKeys}

    # Записуємо довжину найкоротших шляхів
    listOfShortestPaths = list(sorted_dict.values())

    # Нормуємо список найкоротших шляхів
    maxx = max(listOfShortestPaths)
    listOfShortestPathsNormalizedAgainstTheMaximum = [float(i) / maxx for i in
listOfShortestPaths]

    # Записуємо вузли на яких знаходяться важливі речі
    nodesContainingRequests = [activeRequest.node for activeRequest in
self.activeRequests]
    nodesContainingHubs = [activeHub.node for activeHub in self.activeHubs]
    nodesContainingCouriers = [activeCourier.courierAi.currentNode for activeCourier in
self.activeCouriers]
    nodesThatAreTargets = [activeCourier.courierAi.finalTargetNode for activeCourier in
self.activeCouriers]

    # Кодуємо та записуємо спостереження
    observation = list()
    observation.extend(listOfShortestPathsNormalizedAgainstTheMaximum)
    observation.extend(self.encodeNodes(nodesContainingRequests))
    observation.extend(self.encodeNodes(nodesContainingHubs))
    observation.extend(self.encodeNodes(nodesContainingCouriers))
    observation.extend(self.encodeNodes(nodesThatAreTargets))

    # Додаємо біт перевезень, якщо він є
    if self.addCarryingBit:
        if not courier.carryingService:
            observation.append(0)
        else:
            observation.append(1)
    return observation
```

Тепер розглянемо Калькулятор поля. Спочатку його треба запустити щоб створити усіх кур'єри та депо, які були пасивні, потім в ньому можна робити кроки. Один крок це видача однієї команди кур'єру, після чого, поки потреб в командах нема, крок робить миті, доки потреба в команді не з'явиться. Тоді він повертає спостереження з нагородою, та чекає наступної команди щоб зробити крок. Нагорода вираховується індивідуально для кур'єра, який повинен виконати крок, та є різницею його успіхів з кумулятивною кількістю митей, зроблених з моменту попереднього успіху. Успіхами вважається отримання та доставка товарів.

Найбільш важливі функції реалізовані в коді наступним чином:

```
def tickField(self):
    # Ітеруємо усіх кур'єрів
    for _ in range(self.courierId, len(self.activeCouriers)):
        courier = self.activeCouriers[self.courierId]
        # Якщо кур'єр потребує команди - повертаємо спостереження
        if courier.noPathAndMovement():
            return self.observeForCourier(self.courierId)
        courier.iterateCourier()
        self.courierId += 1
    self.courierId = 0

    # Застосовуємо генератор запитів для усіх вершин
    self.generateRequests()
    self.totalTicks += 1
    return None

def step(self, action):
    # Застосовуємо команду до поточного кур'єра, та продовжуємо симуляцію
    courier = self.activeCouriers[self.courierId]
    courier.iterateCourier(action)
    self.courierId += 1
    while True:
        # Якщо мить не поверне спостереження - в команді нема потреби
        observation = self.tickField()
        if observation is not None:
            # Якщо мить повернула спостереження - рахуємо нагороду
            reward = self.evaluate()

            # Повертаємо результати кроку
            return observation, reward, False, "nothing", self.totalTicks

def evaluate(self):
    # Підраховуємо деякі альтернативні параметри
    self.previousTicks = self.totalTicks
    self.previousReceivedRequests = self.totalReceivedRequests
    self.previousVisitedHubs = self.totalVisitedHubs

    # Рахуємо нагороду для кур'єра
    courier = self.activeCouriers[self.courierId]
    reward = courier.reward - courier.punishment
    if courier.reward != 0:
        courier.punishment = 0
        courier.reward = 0

    return reward
```

Головна відмінність Візуалізатора поля від Калькулятора, окрім додавання параметрів для анімації, це відсутність необхідності тренувати агента, бо в цьому нема сенсу коли ми не можемо проводити симуляції швидко, тому замість очікування команди, Візуалізатор зберігає агента та одразу використовує його щоб обрати дію відповідно до спостереження. Також нема потреби в підрахунку нагороди.

Таким чином можна позбутися кроків та залишити тільки миті:

```
@Profiling.timeTracker(Profiler=profiler)
def tickField(frame):
    # Для сумісності зі старим кодом йде перевірка наявності агента
    if not self.hasAi:
        # Якщо агента нема, ітеруємо усіх кур'єрів
        for courier in self.activeCouriers:
            changedCouriers.append(courier.iterateCourier())
    else:
        i = 0
        # При наявності агента, ми ітеруємо через усіх кур'єрів
        for courier in self.activeCouriers:
            if courier.noPathAndMovement():
                # Якщо кур'єр потребує команди - робимо спостереження
                observation = self.observeForCourier(i)

                # Обираємо команду в залежності від спостереження
                action = self.agent.choose_action(observation)

                # Ітеруємо кур'єра, даючи йому команду
                changedCouriers.append(courier.iterateCourier(action=action))
            else:
                # Якщо кур'єр не потребує команди - просто ітеруємо його
                changedCouriers.append(courier.iterateCourier())
            i += 1

    # Застосовуємо генератор запитів для усіх вершин
    self.generateRequests()

    # Ітеруємо профілювання
    profiler.tickProfiling()
```

Безпосередньо візуалізація робиться за допомогою Matplotlib – потужної бібліотеки візуалізації даних для Python, яка надає широкий спектр інструментів і функцій, які зазвичай застосовують для створення високоякісних графіків, діаграм і малюнків [7]. Але його можна адаптувати і для анімацій, що і було зроблено.

3.3 Колективний розум, його складові та імплементація

Колективний розум зберігає аналітичні алгоритми, якими можуть користуватися кур’єри, також він проводить отримання та доставки і виступає прошарком між PPO алгоритмом, та кур’єрами.

Найбільш цікава частина колективного розуму, яку ми ще не розглянули, є механіка отримання і доставки. Вона працює таким чином, що перевіряє чи перевозить кур'єр товар, і в залежності від цього, якщо так – намагається його доставити, якщо ні – отримати. Ці дві функції є дуже схожими, тож є сенс розглянути тільки одну, наприклад функцію яка відповідає за доставку. Все що потрібно зробити це перевірити чи знаходиться кур'єр на запиті, чи приймає запит товар який перевозить кур'єр, та якщо так, оновити статус кур'єра та поля щоб відобразити це.

У коді це реалізовано наступним чином:

```
def provideService(self, courier):
    # Створюємо лист вузлів з активними запитами
    nodesContainingRequests = [activeRequest.node for activeRequest in
                               self.fieldVisualiser.activeRequests]

    # Перевіряємо чи знаходиться кур'єр на вузлі з запитом
    onTheRequest = courier.courierAi.currentNode in nodesContainingRequests
    providedService = -1

    # Якщо кур'єр знаходиться на вузлі з запитом
    if onTheRequest:
        # Беремо запит на якому знаходиться кур'єр
        request = self.fieldVisualiser.activeRequests[
            nodesContainingRequests.index(courier.courierAi.currentNode)]

        # Намагаємося доставити товар, якщо він не підходить, отримуємо -1
        providedService = request.receiveService(courier.carryingService)

        # Якщо товар доставлено
        if providedService != -1:
            # Перевіряємо чи видаляє кур'єр запити власноруч
            if courier.oneStepBehind:
                # Якщо так - відмічаємо запит для видалення
                request.delayedDeletion=courier.id
            else:
                # Якщо ні - одразу видаляємо запит
                self.fieldVisualiser.deleteRequest(request)

            # Прибираємо товар з кур'єра
            courier.carryingService = None

            # Подаємо команду що запит виконано
            self.fieldVisualiser.receiveRequest(request)

            # Даємо кур'єру нагороду
            courier.reward+=1

            # Затримуємо кур'єра на небхідну кількість митей
            courier.courierAi.freeze += providedService

    # Повертаємо успішність доставки
    return providedService != -1
```

3.4 Запуск PPO у середовищі

Для того щоб запустити PPO у середовищі, треба спочатку створити як агента, так і середовище, задавши усі необхідні гіперпараметри, включаючи кількість кроків у грі, та кількість ігор. Далі на початку кожної гри середовище перезапускається, і кожен крок проводиться спостереження, агент обирає дію та отримує нагороду, раз в кілька кроків агент вчиться, коли зроблено усі кроки у грі, вона починається заново, та виводиться результат гри, а якщо середній результат останніх сотні ігор є найкращим – він записується, а модель зберігається. По завершенню усіх ігор, виводиться графік рахунків кожної гри.

В коді ітерація гри реалізована наступним чином:

```
# Перезапускаємо середовище, та отримуємо перше спостереження
observation = env.reset()

# Обнулюємо дані
done = False
score = 0
totalTicks = 0

# Повторюємо, поки кількість кроків не дорівнює заданій
while not done:
    # Обираємо дію в залежності від спостереження
    action, prob, val = agent.choose_action(observation)

    # Отримуємо нове спостереження з середовища
    observation_, reward, done, info, totalTicks = env.step(action)

    # Ведемо облік
    n_steps += 1
    score += reward
    agent.remember(observation, action, prob, val, reward, done)

    # Якщо кількість кроків кратна заданій - проводимо навчання
    if n_steps % N == 0:
        agent.learn()
        learn_iters += 1
    observation = observation_

    # Якщо кількість кроків дорівнює заданій - закінчуємо гру
    if n_steps % numberOfSteps == 0:
        done = True

# Записуємо історію рахунків
score_history.append(score)

# Рахуємо середнє за останні 100 ігор
avg_score = np.mean(score_history[-100:])

# Якщо воно найкраще - зберігаємо модель
if avg_score > best_score:
    best_score = avg_score
    agent.save_models()

# Виводимо результат гри
print('episode', i, 'score %.1f' % score, 'Relative score',
      1000 * (score + totalTicks / 1000) / totalTicks,
      'avg score %.1f' % avg_score,
      'time_steps', n_steps, 'learning_steps', learn_iters)
```

4 РЕЗУЛЬТАТИ НАВЧАННЯ НА ПОСТАВЛЕНІЙ ЗАДАЧІ

Процес та результати навчання дуже залежать від реалізації та гіперпараметрів. Так якщо прорахунок нагороди працює неправильно, нейромережа намагається максимізувати не ті параметри, через що можемо отримати наступний процес навчання:

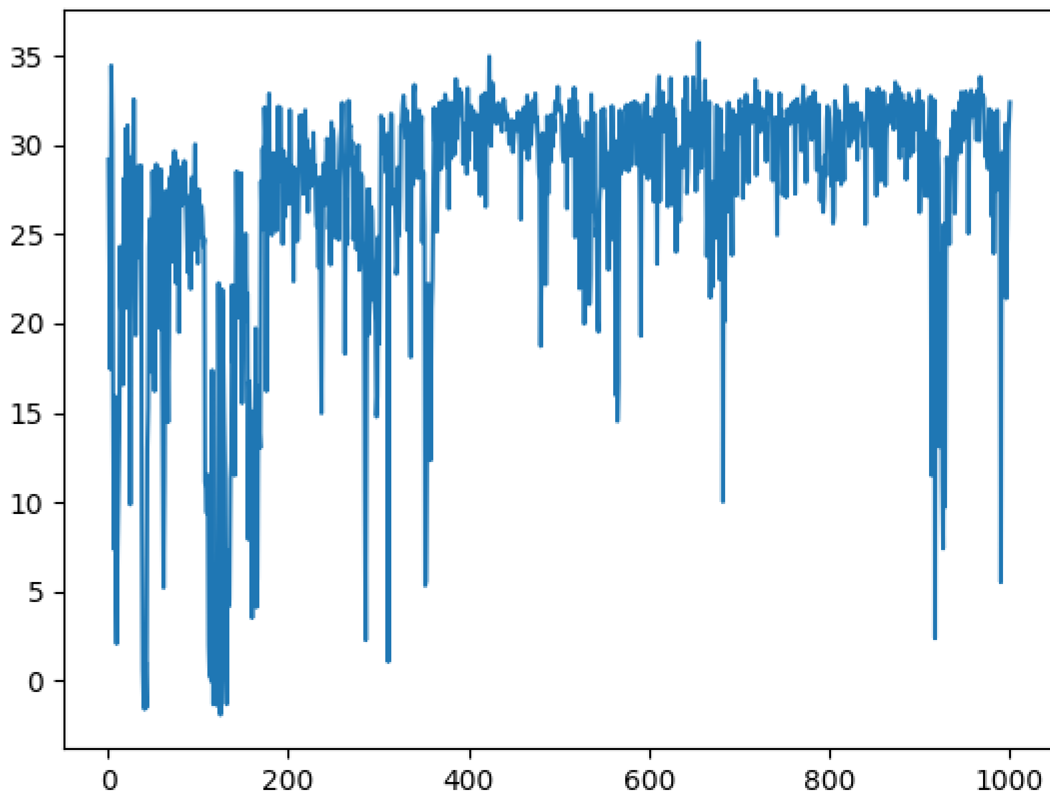

Рисунок 4.1

Тут і далі, по вісі абсцис позначено ігри, по вісі ординат – рахунок гри

Другий важливий гіперпараметр – темп навчання. Так при високому темпі нейромережа вчиться швидше, але набагато більш шумно.

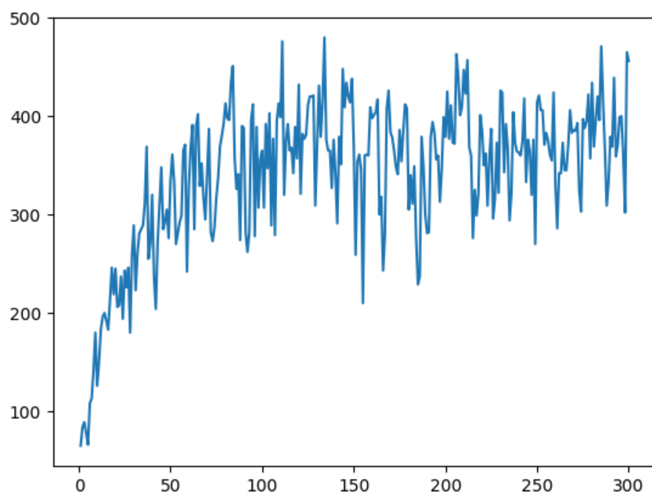

Рисунок 4.2 процес навчання з високим темпом

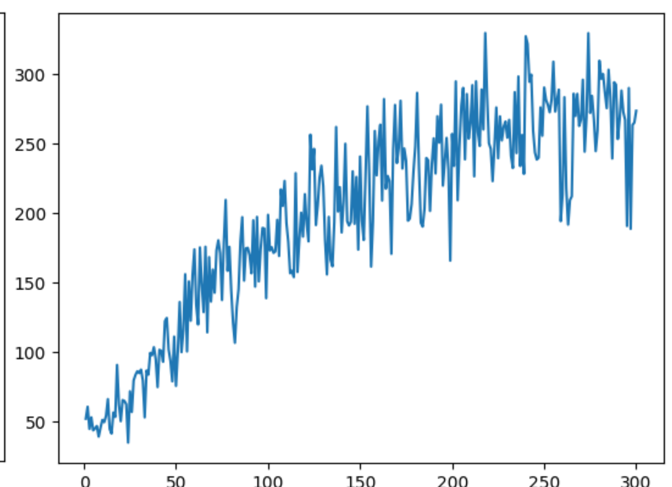

Рисунок 4.3 процес навчання з низьким темпом

Якщо ж параметри підібрати правильно, можна отримати досить красивий та продуктивний результат:

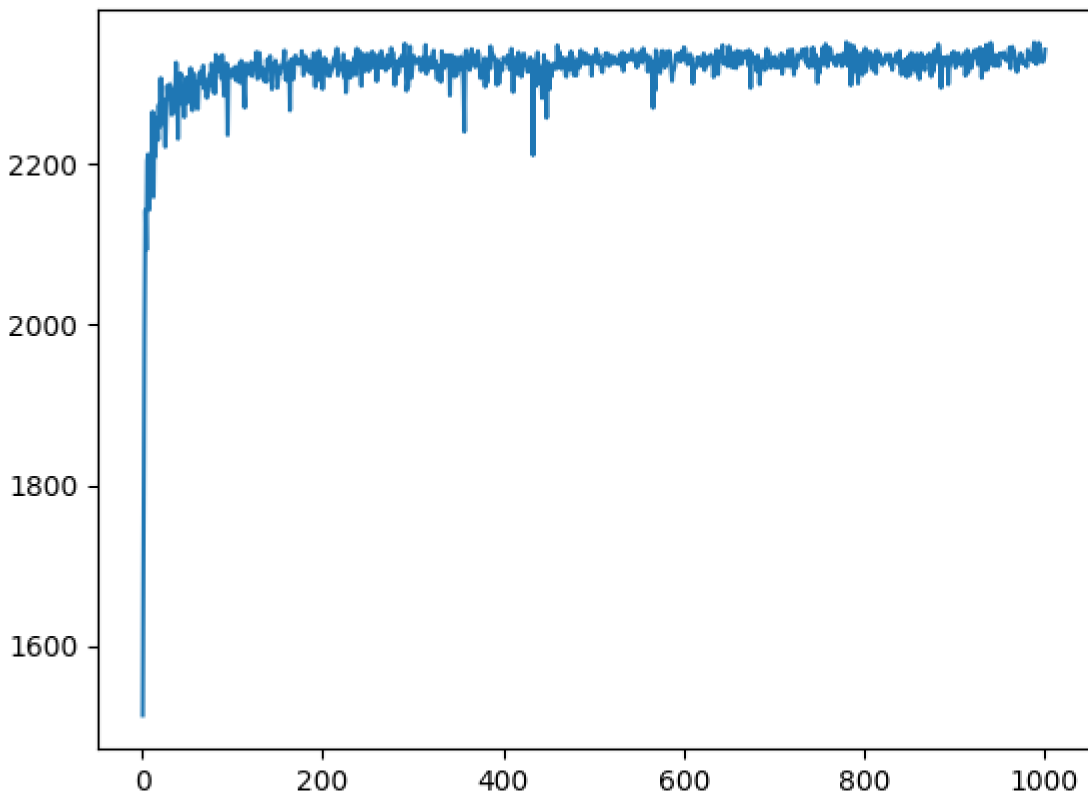

Рисунок 4.4 процес навчання з правильно підібраними гіперпараметрами

Але постає інша проблема: рахунок дуже залежить від того наскільки багато ми робимо кроків в кожній грі, тому можна помітити що на рисунках 4.1, 4.2 і 4.4 вісь ординат має зовсім різний вигляд. Щоб з цим боротися, після проведення гри вираховується відносний рахунок, шляхом ділення рахунку гри на кількість митей, та множенню на тисячу для зручності сприйняття. Фактично це значення є швидкістю роботи агента або алгоритму, і хоча агент це значення не бачить, PPO, завдяки знижці, повинно власноруч прагнути найшвидшого рішення.

Щодо безпосередньо результатів, на десяти вузлах, двох депо та двох кур'єрах, аналітичний алгоритм досяг відносного рахунку, що дорівнює в середньому 13.158, в той час як агент, всього за тисячу ігор, зміг досягти 11.378, в локальному оптимумі. Якщо це візуалізувати, то можна побачити, що агент використовує лише одне депо, що не є оптимальним. Щоб обійти це можна або

дати більше часу, і сподіватися що регуляризація ентропії в якийсь момент дасть нейромережі ідею користуватися іншим депо, або можна змінити гіперпараметри, такі як наприклад темп навчання, або змінити алгоритм прорахунку нагороди вцілому.

ВИСНОВКИ

В процесі виконання даної роботи вдалося розробити середовище для симуляції та вирішення задач пошуку оптимального маршруту транспортного засобу, імплементувати можливість використання різних алгоритмів в ньому, включаючи деякі аналітичні алгоритми, та алгоритм навчання з підкріпленням Proximal Policy Optimization.

Незалежно від алгоритму вирішення, є можливість візуалізовати середовище, що дозволяє не тільки використовувати це для того щоб відлагоджувати програму, а і для отримання нових ідей, створених агентом.

Для запуску агента все що потрібно зробити, це задати гіперпараметри, все інше програма робить самостійно. Це дозволяє легко коригувати елементи програмного засобу, не хвилюючись що процес навчання постраждає.

Хоча результати роботи штучного інтелекту на малих об’ємах даних не є кращими за аналітичні, вдалося показати що усі частини програми є працездатними.

Завдяки Ощадливій моделі розробки, було досліджено багато нових технологій та методів, включаючи бібліотеки networkx, numpy, matplotlib та torch [8], методи навчання з підкріпленням такі як vanilla policy gradient, natural policy gradient, trust region policy optimization та безпосередньо proximal policy optimization.

Таким чином виконання цієї роботи призвело до створення програмного засобу, і до того ж покращило навички розробників у сферах штучного інтелекту, навчання з підкріпленням та методів обчислювальної геометрії та комп’ютерної графіки.

Повний код програмного засобу можна переглянути тут: https://github.com/AlexRodry007/SolvingLogisticsUsingAI

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Andriansyah A. Pickup and delivery problem with LIFO, time duration, and limited vehicle number [Електронний ресурс] / A. Andriansyah, N. Prasanti, D. S. Prima // ResearchGate. – 2018. – Режим доступу до ресурсу: https://www.researchgate.net/publication/327794861_Pickup_and_delivery_problem_with_LIFO_time_duration_and_limited_vehicle_number.
2. Pound M. Dijkstra's Algorithm - Computerphile [Електронний ресурс] / Mike Pound // Computerphile. – 2017. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=GazC3A4OQTE.
3. van Heeswijik W. Proximal Policy Optimization (PPO) Explained [Електронний ресурс] / Wouter van Heeswijik // Towards Data Science. – 2022. – Режим доступу до ресурсу: https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b.
4. Tabor P. Proximal Policy Optimization (PPO) is Easy With PyTorch | Full PPO Tutorial [Електронний ресурс] / Phil Tabor // Machine Learning with Phil. – 2020. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=hlv79rcHws0.
5. Proximal Policy Optimization Algorithms [Електронний ресурс] / [J. Schulman, F. Wolski, P. Dhariwal та ін.] // Cornell University. – 2017. – Режим доступу до ресурсу: https://arxiv.org/abs/1707.06347.
6. Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008
7. J. D. Hunter, "Matplotlib: A 2D Graphics Environment," in Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, May-June 2007, doi: 10.1109/MCSE.2007.55.Abstract: Matplotlib is a 2D graphics package used for Python for application development, interactive scripting,and publication-

quality image generation across user interfaces and operating systemsURL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4160265&isnumber=4160244

8. PyTorch [Електронний ресурс] / [A. Paszke, S. Gross, S. Chintala та ін.] // Meta AI. – 2016. – Режим доступу до ресурсу: https://pytorch.org/.

ДОДАТОК А

Реалізація аналітичного алгоритму без використання біту перевезень

В цьому варіанті ми просто дивимось чи не знаходиться кур’єр на вузлі з депо, якщо ні – повертаємось в депо, якщо так – відправляємось к найближчому запиту.

```
def ComplexAnalyticalWithoutCarryingBit(splited, currentNode):
    if splited[2][currentNode] == 1:
        possibleTargets = list()
        request = 0
        for isRequest in splited[1]:
            if splited[4][request] == 0 and isRequest == 1:
                possibleTargets.append(request)
            request += 1
        if not possibleTargets:
            return currentNode
        target = possibleTargets[0]
        closest = splited[0][target]

        for possibleTarget in possibleTargets:
            if splited[0][possibleTarget] < closest:
                closest = splited[0][possibleTarget]
                target = possibleTarget
        return target
    else:
        possibleTargets = list()
        hub = 0
        for isHub in splited[2]:
            if isHub == 1:
                possibleTargets.append(hub)
            hub += 1
        # print(currentNode)
        # print(splited[0])
        # print(possibleTargets)

        target = possibleTargets[0]
        closest = splited[0][target]
        for possibleTarget in possibleTargets:
            # print(possibleTarget, splited[0][possibleTarget])
            if splited[0][possibleTarget] < closest:
                closest = splited[0][possibleTarget]
                target = possibleTarget
        # print(target)
        # input()
        return target
```