Одеський національний університет імені І.І.Мечникова
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття ступеня вищої освіти «магістр»
(освітньо-кваліфікаційний рівень)

на тему

Методи кооперативного пошуку шляхів у мультиагентному середовищі

Methods of cooperative wayfinding in a multi-agent environment

Виконав: студент денної форми навчання
спеціальності 123 – Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
Коган Владислав Владиславович
(прізвище, ім'я, по-батькові)
Керівник к.т.н., доц. Пенко В.Г.
(науковий ступінь, вчене звання, прізвище та ініціали)
Рецензент к.ф.-м.н., доц. Шпінарева І.М.
(науковий ступінь, вчене звання, прізвище та ініціали)
Рецензент д.т.н., проф. Ковальчук Л.В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___»_____ 2023 р.

Завідувач кафедри
_____ Євгеній МАЛАХОВ
(підпис) (ім'я, прізвище)

Захищено на засіданні ЕК № ___
протокол № __ від «___» ____ 2023 р.
Оцінка _____/_____/_____
(за національною шкалою, шкалою ECTS, бали)
Голова ЕК
_____ Алла Кобозєва
(підпис) (ім'я, прізвище)

Одеса – 2023

АНОТАЦІЯ

Пошук шляху у мультиагентних системах залишається областю досліджень, для якої може бути запропоновано безліч рішень зі своїми перевагами та недоліками. Метою мультиагентного пошуку шляху (multi-agent pathfinding, MAPF) є пошук шляху для декількох агентів, розташованих у спільному просторі. В таких випадках важливо забезпечити ефективне управління рухом агентів, враховуючи їх взаємодію та запобігання конфліктів, уникнення зіткнень агентів між собою тощо. Багато реальних практичних задач можна розглядати як завдання мультиагентного пошуку шляху, такі як планування руху, логістика та прийняття рішень.

Алгоритми пошуку, як правило, є затратними у випадках великих графів. Тому існує необхідність у прискоренні алгоритмів пошуку шляху, коли це можливо. Традиційні одноагентні алгоритми пошуку шляху, такі як алгоритм Дейкстри, практично не піддаються подальшій оптимізації. Модифікації алгоритму Дейкстри для мультиагентних систем, напроти, мають цей потенціал.

Метою кваліфікаційної роботи є прискорення та підвищення ефективності наявного алгоритму мультиагентного пошуку шляху. У роботі проведено аналіз можливості використання векторизації окремих етапів алгоритму пошуку за допомогою AVX-інструкцій. Порівняно ефективність векторизації за допомогою компіляторів із мануальною реалізацією векторного коду. В результаті кваліфікаційної роботи реалізовано та продемонстровано прискорення за допомогою AVX-інструкцій напівкооперативного алгоритму пошуку шляху для декількох агентів, які уникають взаємних конфліктів.

Ключові слова: теорія графів, пошук шляху, мультиагентні системи, конфлікти, векторизація

ABSTRACT

Pathfinding in multi-agent systems remains an area of research for which many solutions can be proposed with their advantages and disadvantages. The goal of multi-agent pathfinding (MAPF) is to find a path for several agents located in a shared space. In such cases, it is important to ensure effective management of agent movement, taking into account their interaction and conflict prevention by avoiding agent collisions. Many real-world practical problems can be viewed as multi-agent pathfinding problems, such as motion planning, logistics, and decision making.

Traditional single-agent pathfinding algorithms, such as Dijkstra's algorithm, are practically not amenable to vectorization. Modifications of the Dijkstra algorithm for multi-agent systems, on the other hand, have this potential.

The aim of the qualification work is to speed up and improve the efficiency of the existing multi-agent pathfinding algorithm. The work studied the possibility of speeding up some stages of the search algorithm via vectorization, investigated the technical aspects of integrating AVX instructions in the context of the algorithm and compared the efficiency of vectorization using compilers with manual implementation of vector code. As a result of the qualification work, a hybrid pathfinding algorithm for multiple agents that avoids mutual conflicts has been improved and accelerated by AVX instructions.

Keywords: pathfinding, algorithms, graph theory, vectorization

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ. УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ 5
ВСТУП 6
1 ОГЛЯД НАЯВНИХ ПІДХОДІВ 8
1.1. Алгоритм Дейкстри 8
1.2. Алгоритм A* 9
1.3. Алгоритм CA* 10
1.4. Проблеми вирішення задачі MAPF 11
2 ОГЛЯД НАПІВКООПЕРАТИВНОГО АЛГОРИТМУ ДЕЙКСТРИ 13
2.1. Опис алгоритму 13
2.2. Демонстрація принципу роботи 15
3 ПРОМІЖНІ УДОСКОНАЛЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ 17
3.1. Перехід до пріоритетної черги вершин 17
3.2. Зміна топології графу 18
3.3. Демонстрація роботи алгоритму на матриці суміжності 21
4 ВЕКТОРИЗАЦІЯ ОКРЕМИХ ЕТАПІВ АЛГОРИТМУ 28
4.1. Аналіз функції імітації плину часу 28
4.2. Залежності даних у алгоритмі Дейкстри 30
4.3. Можливості пришвидшення алгоритму 31
4.4. Параметри компіляції 34
4.5. Прискорення функції імітації часу 34
5 ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕЗУЛЬТАТІВ ПРИСКОРЕННЯ 38
5.1. Порівняння швидкодії 38
ВИСНОВОК 42
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 43
ДОДАТОК А. Код програми напівкооперативного алгоритму Дейкстри із та без векторизації 46

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

Скорочення

A* – A-star

AVX – Advanced Vector Instructions

CA* – Cooperative A*

FPU – Floating Point Unit

MAPF – multi-agent path-finding

NP – non-deterministic polynomial time

SIMD – single instruction multiple data

SSE – Streaming SIMD Extensions

ВСТУП

В сучасному науковому та технічному середовищі вирішення проблем мультиагентного пошуку шляху (MAPF) залишається актуальним завданням, оскільки воно знаходить широке застосування у великому спектрі високотехнологічних областей, таких як логістика, робототехніка, навігація, комп’ютерні ігри, виробничі процеси, моделювання систем тощо.

MAPF визначається як пошук шляху для декількох агентів, розташованих у спільному просторі, з обов'язковим врахуванням їх взаємодії та уникненням конфліктів. Розвиток та оптимізація алгоритмів в цій галузі мають високий потенціал застосування у різних сферах життя.

Поєднання різних проблем і середовищ призводить до багатьох викликів у цій області досліджень. Ці проблеми включають довгий час пошуку, велике споживання пам'яті, неоптимальні шляхи. Необхідність вирішення цих проблем привертає чимало інтересу. Хоча вже існує багато досліджень щодо методів прискорення алгоритмів пошуку шляхів, ця проблема далека від того, щоб бути вичерпаною.

Традиційні алгоритми пошуку шляху, спрямовані на одного агента, такі як алгоритм Дейкстри, мають обмежені можливості до подальшого прискорення. У контексті MAPF, модифікації алгоритму Дейкстри мають значний потенціал до зростання їх ефективності. Це становить великий інтерес для подальшого дослідження, адже оптимізація може потенційно прискорити швидкодію алгоритмів, у який вона використовується.

Об’єктом дослідження є методи і алгоритми MAPF. Предметом дослідження є методи прискорення напівкооперативного алгоритму MAPF.

Метою кваліфікаційної роботи є прискорення та підвищення ефективності наявного алгоритму мультиагентного пошуку шляху. Для досягнення цієї мети необхідно виконати наступні задачі:

а) проаналізувати наявний напівкооперативний алгоритм MAPF з точки зору більш універсального представлення середовища агентів;

б) розробка методів врахування послідовності дій агентів в часі;

в) дослідити та реалізувати деякі методи прискорення напівкооперативного алгоритму MAPF;

г) провести експериментальний аналіз результатів роботи поліпшеного алгоритму порівняно з попередньою версією.

1 ОГЛЯД НАЯВНИХ ПІДХОДІВ

1.1 Алгоритм Дейкстри

Алгоритм Дейкстри – це алгоритм пошуку найкоротших шляхів у вагованому графі від одного заданого вузла до всіх інших. Розроблений Едсгером Дейкстрою, він ефективно вирішує задачу за умови, що граф не має циклів з від'ємними вагами. Класичний алгоритм Дейкстри має часову складність O(V^2), де V – кількість вузлів у графі. Алгоритм розширюється за допомогою пріоритетної черги, щоб зменшити складність до O((V + E) * log V), де E – кількість ребер у графі [1].

Використання алгоритму можна розглянути на прикладі графу у вигляді двовимірної сітки (рис. 1.1.), де кожна вершина представляє певну локацію, а ребра визначають вагу шляху між локаціями. Під час виконання алгоритму, вузли обробляються послідовно, і ваги найкоротших шляхів до них оновлюються. Завдяки алгоритму Дейкстри можна знайти оптимальний шлях від одного вузла до всіх інших у графі. Синіми клітинами відзначені вершині, які алгоритм обчислив спочатку, блакитними – в кінці, сірими – вершини, що не відвідувалися.

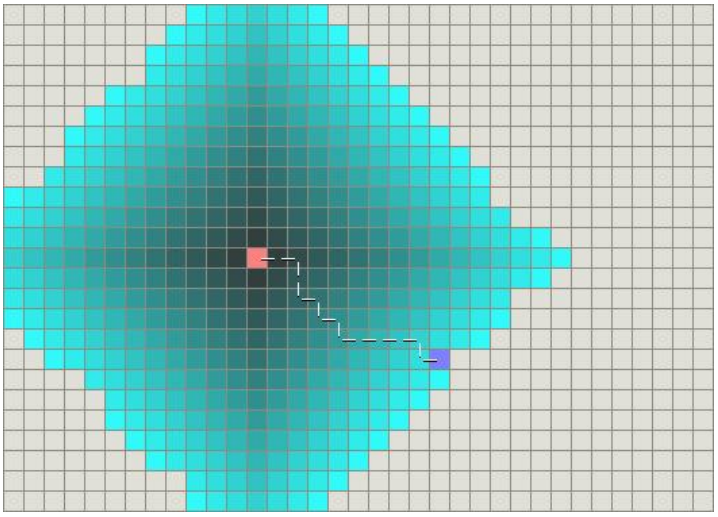


Рисунок 1.1 – Алгоритм Дейкстри на сітці

1.2 Алгоритм A*

A* схожий на алгоритм Дейкстри A та на жадібний пошук за першим найкращим збігом (Greedy Best-First Search) в тому, що він може використовувати евристику, щоб керувати собою. У простому випадку він працює так само швидко, як і жадібний найкращий пошук (рис.1.2). У прикладі з увігнутою перешкодою A* знаходить шлях так само добре, як і алгоритм Дейкстри (рис. 1.3).

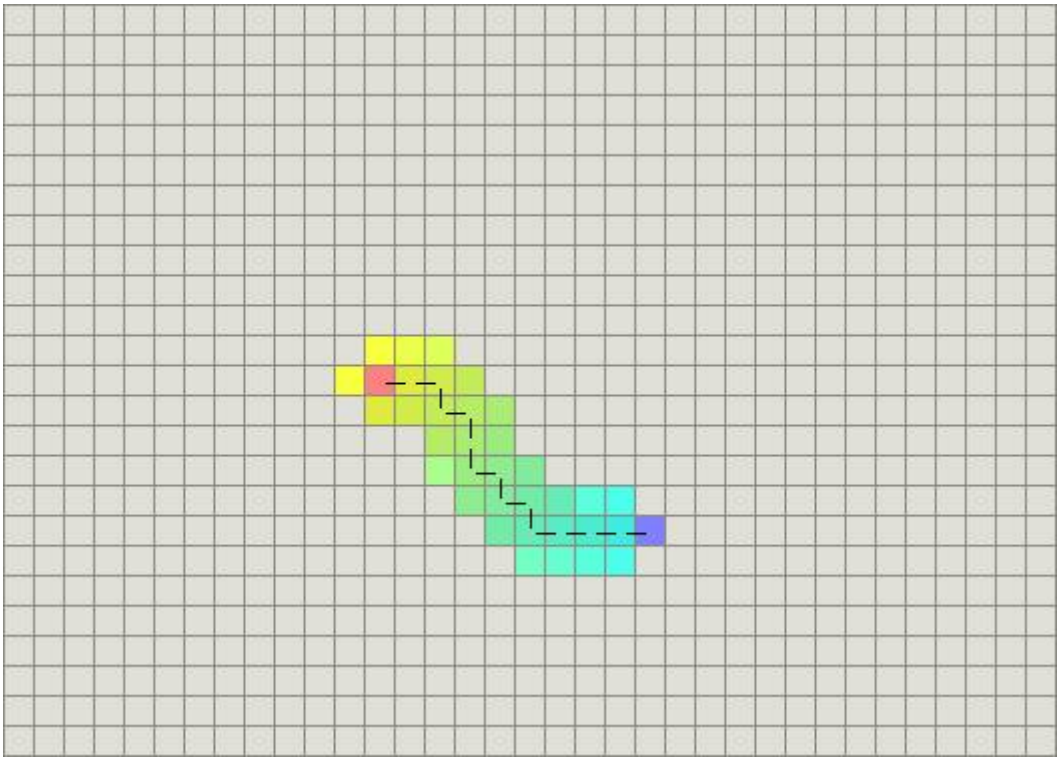


Рисунок 1.2 – Алгоритм A* без перешкод

Секрет його успіху полягає в тому, що він поєднує інформацію, яку використовує алгоритм Дейкстри (надаючи перевагу вершинам, близьким до початкової точки), та інформацію, яку використовує жадібний найкращий пошук (надаючи перевагу вершинам, близьким до мети) [2].

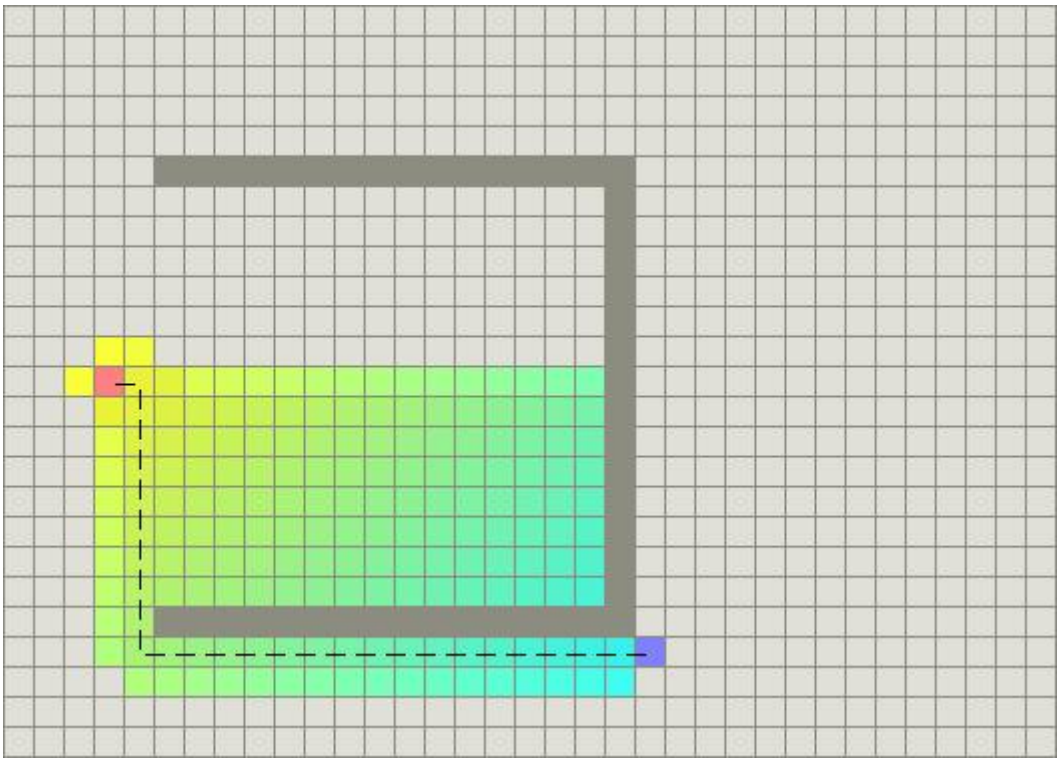


Рисунок 1.3 – Алгоритм A* із перешкодою

Секрет його успіху полягає в тому, що він поєднує інформацію, яку використовує алгоритм Дейкстри (надаючи перевагу вершинам, близьким до початкової точки), та інформацію, яку використовує жадібний найкращий пошук (надаючи перевагу вершинам, близьким до мети) [2].

1.3 Алгоритм CA*

Алгоритм CA* є розвитком алгоритму A* задля вирішення задачі мультиагентного пошуку шляху, запропонованим Девідом Сілвером. Алгоритм працює в тривимірному просторі-часі, враховуючи плановані маршрути інших агентів. Ключовою особливістю є можливість агента залишатися на місці, щоб уникнути колізій. Після обчислення маршруту для кожного агента, стани по маршруту відзначаються в таблиці резервування, що унеможливлює використання цих місць іншими агентами (рис 1.4).

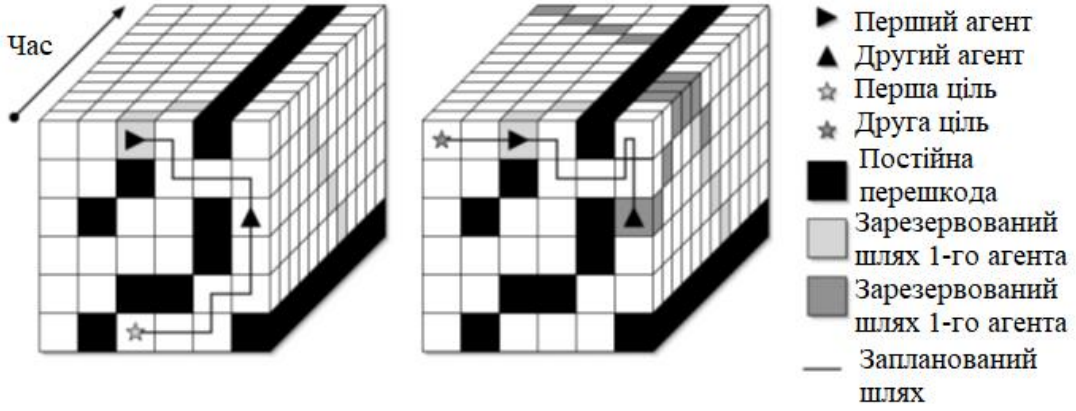


Рисунок 1.4 – Кооперативний пошук агентів зліва та справа. Агент справа будує свій шлях з урахуванням того, що деякі вузли резервовані попереднім агентом

Важливою перевагою CA* є маркування регіонів простору-часу, які зайняті іншими агентами. Це забезпечує спільне знання агентів про маршрути одне одного та допомагає уникнути зіткнень [3]. Слід зазначити, що рішення є субоптимальним, та його точність залежіть від заданої евристичної функції.

Ці алгоритми слугують фундаментом для нових алгоритмів та модифікацій наявних алгоритмів. Майже кожна модифікація претендує на перевагу в тій чи іншій ніші, такими як швидкодія чи уникання конфліктів.

1.4 Проблеми вирішення задачі MAPF

Однією з провідних причин створення нових алгоритмів пошуку шляху є спроби запропонувати оптимізовані рішення. Алгоритми кооперативного алгоритми пошуку шляху, як правило, є алгоритмічно складними. Наприклад, алгоритм Корнхаузера, який є найбільш загальним алгоритмом для вирішення задачі MAPF, має алгоритмічну складність O(V^3) [4] (рис. 1.5). Цей алгоритм є складним та заплутаним математично та структурно, не існує жодних доступних імплементацій цього алгоритму популярними мовами програмування [5].

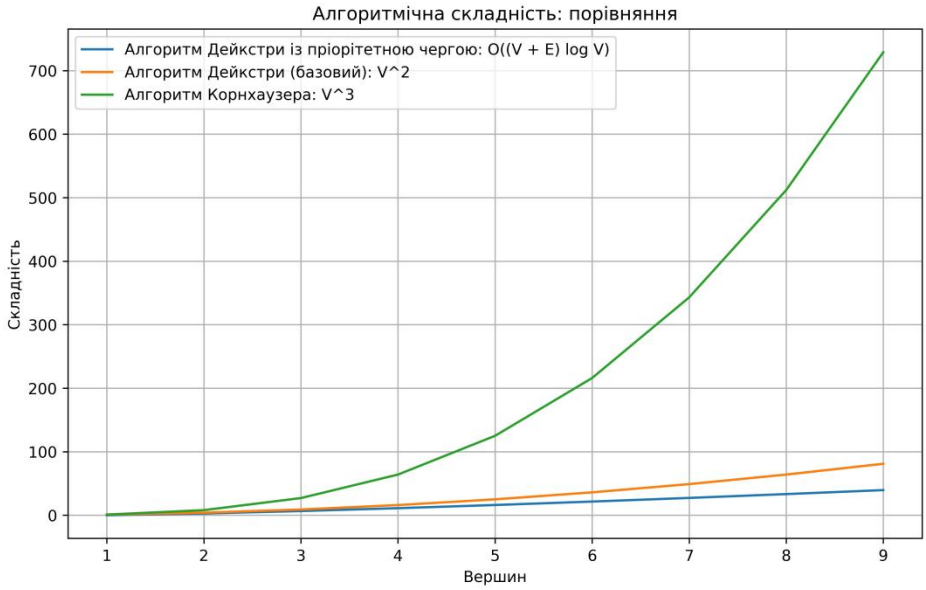


Рисунок 1.5 – Порівняння алгоритмічної складності деяких алгоритмів пошуку шляху

Алгоритм Корнхаузера не гарантує оптимальності рішення. Задача вирішення задачі MAPF є NP-складною, на даний момент не відомо універсального алгоритму, здатного вирішити її за поліноміальний час [6, 7]. Оптимальне вирішення за поліноміальний час може бути знайдено лише у деяких спеціальних випадках [4, 7, 8]. Субоптимальні рішення може бути знайдено за поліноміальний час у всіх випадках.

Таким чином, стрибок алгоритмічної складності між некооперативними та кооперативними алгоритмами пошуку, як правило, являється вкрай значним. Це дозволяє зробити висновок щодо необхідності створення напівкооперативного алгоритму, який намагається гнучко балансувати між цими двома підходами. Головним фокусом напівкооперативного алгоритму має бути швидкодія. Модифікації, створені на основі алгоритмів Дейкстри та A* найбільш підходять для цього, адже мають відносно низьку базову алгоритмічну складність.

2 ОГЛЯД НАПІВКООПЕРАТИВНОГО АЛГОРИТМУ ДЕЙКСТРИ

2.1 Опис алгоритму

На основі кооперативного A* Девіда Сілвера було створено напівкооперативний алгоритм [9], який можна умовно назвати «напівкооперативним алгоритмом Дейкстри. Поняття «перехідний», «гібридний» та «напівкооперативний» в данному контексті є тотожними.

Алгоритм можна описати таким чином:

а) припустимо, що певному агенту потрібно знайти найкоротший шлях від вершини A до вершини Z;

б) найкоротший шлях визначається за допомогою алгоритму Дейкстри.

в) ваги всіх вершин на шляху від A до Z (вкючено із вершинами A та Z) збільшуються на певну величину.

г) для подальших агентів маршрут від A до Z стає менш привабливим через збільшення ваги на шляху.

д) збільшені ваги можуть бути поступово зменшені до початкового (що симулює плин часу).

Основною відмінністю цього алгоритму від CA* є відсутність окремої структури даних для резервування клітин, замість цього відбувається зміна ваги ребер між відвіданими вершинами. Алгоритм CA* симулює додатковий вимір простору, час, шляхом резервування клітин за певним агентом у конкретний момент часу.

Напівкооперативний алгоритм Дейкстри не має окремого виміру часу. Замість цього він «згортає» третій вимір часу усередину вже наявних в алгоритмі ребер графу, але платить за це ціною зменшення точності рішення. Агенти збільшують значення ваг для кожного відвіданого ними ребра, що робить відповідні ребра менш привабними для подальших

агентів. Але це підвищення не має конкретного сроку, на який воно встановлено.

У рамках алгоритму CA* агенти ніби говорять: «мені потрібно це ребро у цей конкретний момент часу». У рамках напівкооперативного алгоритму Дейкстри агенти говорять: «мені потрібно це ребро, але я не знаю наскільки довго», оскільки час не є окремим виміром, і підвищує вагу цього ребра.

Слід зазначити, що цей алгоритм є напівкооперативним у тому сенсі, що він не повністю, а частково уникає конфліктів. Ця частковість обумовлена декількома факторами, серед яких одними з ключових є простота та швидкість алгоритму. Простота дозволяє потенційно адаптувати запропоновану модифікацію для багатьох варіантів пошуку шляху, не обмежуючись лише алгоритмом Дейкстри. У цьому відношенні алгоритм Дейкстри є зручним для розробки та демонстрації модифікації, але цей самий принцип може бути адаптований і для A* тощо.

Швидкість здатна забезпечити оперативність в умовах, де оперативність обчислень важлива, наприклад, у віртуальних середовищах або в галузі комп'ютерних ігор, де застосування CA* чи алгоритму Корнхаузера було б невиправданим із-за загальних обмежень наявних обчислювальних ресурсів.

Таким чином, алгоритм використовує симуляцію тривимірного простору для ефективного знаходження шляхів в графі. Як результат, зміна ваг дозволяє агентам побудувати неконфліктні один з одним маршрути. Конфліктом за ресурс вважається спроба двох та більше агентів одночасно скористатися одним і тим самим ресурсом – ребром.

Крім того, алгоритм є частковим у тому сенсі, що він лише частково симулює плин часу. В рамках перехідного алгоритму час не є окремим виміром, втім додаткові ваги можуть як збільшуватись у результаті прокладання все нових і нових маршрутів агентами, так і зменшуватись в результати симуляції плину часу.

Програмна реалізація алгоритму була виконана мовою програмування C++ без використання зовнішніх бібліотек.

2.2 Демонстрація принципу роботи

Наведемо демонстрацію роботи алгоритму.

На рис. 2.1 зображено приклад підвищення ваг по пройденному агентом шляху.

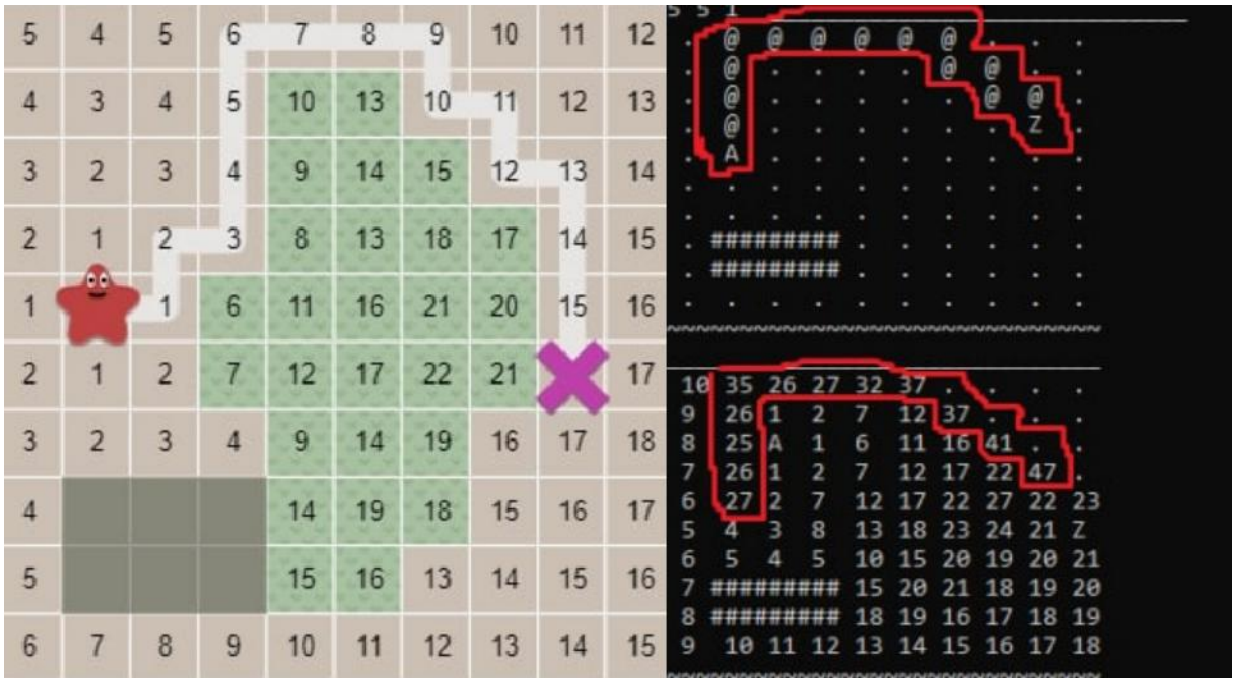


Рисунок 2.1 – Алгоритм Дейкстри на сітці. Ваги збільшено

В результаті побудова шляху наступним агентом відбувається за неконфліктним маршрутом, що принципово відрізняється від базового алгоритму (рис. 2.2).

Можна побачити, що агент обрав обхідний шлях замість повторення шляху першого агента.

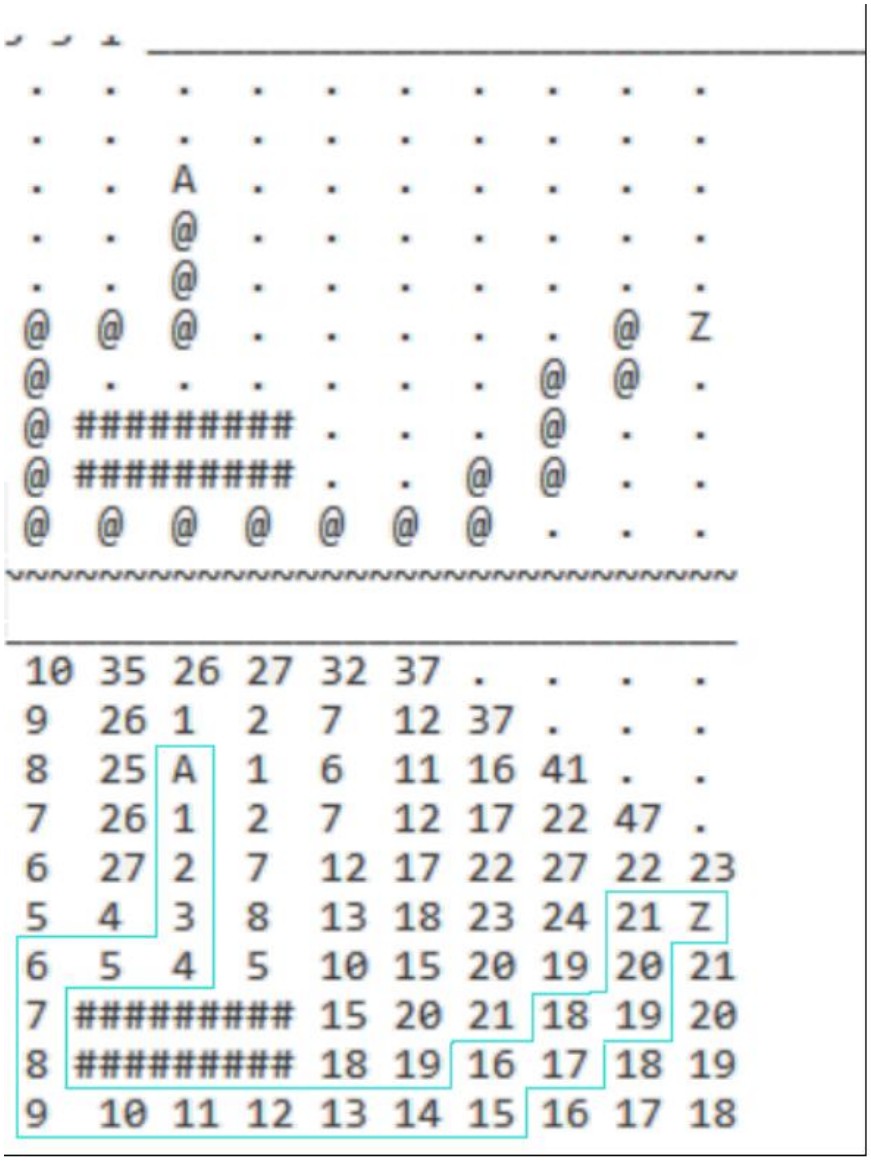


Рисунок 2.2 – Маршрут для другого агента на оновленій карті

3 ПРОМІЖНІ УДОСКОНАЛЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

3.1 Перехід до пріоритетної черги

Після завершення роботи [9] велись подальші дослідження у області застосування цього перспективного алгоритму. Було опубликовано статтю [10] із додатковими експериментами та оцінками перспектив алгоритму.

Одним із висновків, зазначених у статті, було те, що початкова модифікація алгоритму Дейкстри була розроблена для пошуку шляху на двовимірній сітці. Фактично це було суттєвим обмеженням, яке виключало можливість ефективного використання на загальних графах.

Топологія сітки із роботи [9] мала квадратичну складність:

$$O(V\hat{}2)$$

За низької кількості вузлів V складність цього алгоритму є прийнятною. Однак зі зростанням кількості вузлів, алгоритмічна складність значно зростає, що робить його менш ефективним для великих графів.

Замість цього за базовий алгоритм у поточній роботі було взято алгоритм Дейкстри із пріоритетною чергою [1]. Ця модифікація має значно меншу алгоритмічну складність, ніж сітка. Алгоритмічна складність цього алгоритму (E - кількість ребер):

$$O((V + E) * \log V)$$

Цей алгоритм ефективно працює навіть при великій кількості вузлів та ребер. Час, що витрачає алгоритм на виконання цього алгоритму, в середньому зростає логарифмічно. Такий підхід робить алгоритм більш придатним для великих та складних графів. На рис. 3.1 представлено порівняння цих двох алгоритмів шляху пошуку в графі. Дане порівняння наочно демонструє переваги обраного алгоритму Дейкстри із пріоритетною

чергою над початковою модифікацією алгоритму, орієнтованою на двовимірну сітку. Зокрема, під час росту розмірів графа, алгоритм із пріоритетною чергою виявляється значно ефективнішим і менш обтяженим обчислювальними витратами [1].

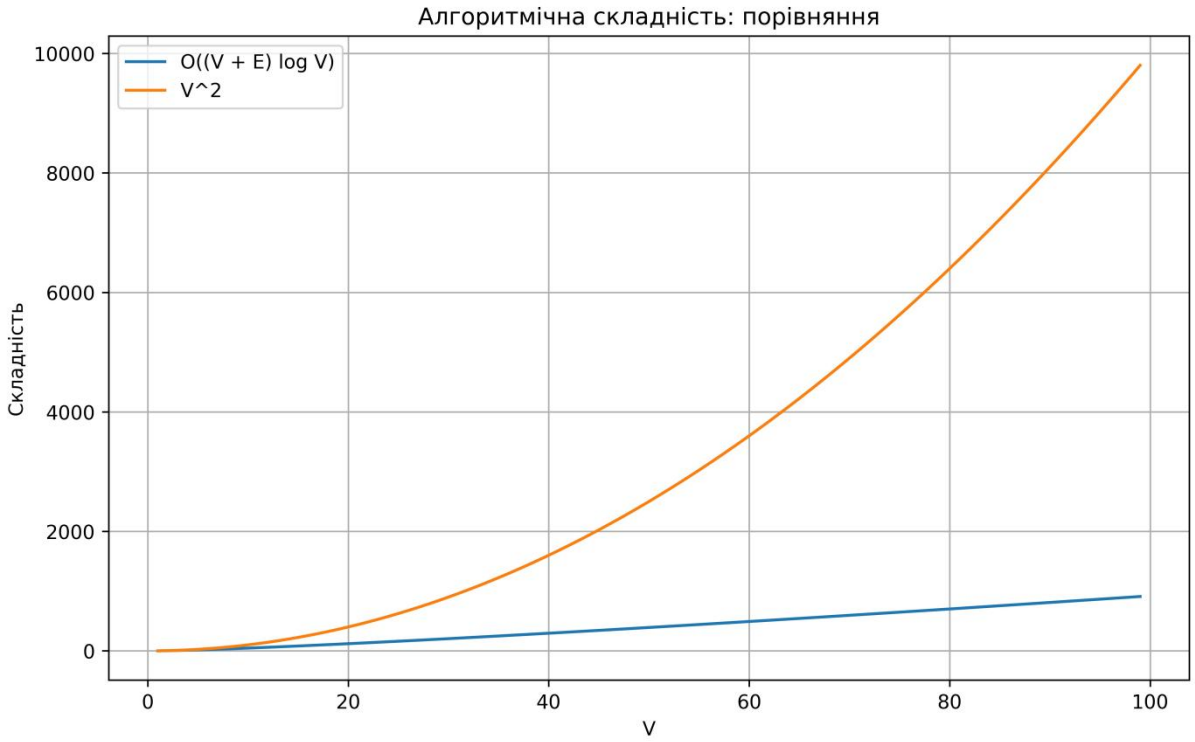


Рисунок 3.1 – Порівняння алгоритмічної складності

Це має велике значення, оскільки суттєво підвищує здатність базового алгоритма працювати на у тому числі на великих графах. В свою чергу, це відкриває можливість для подальшої оптимізації, таких як оптимізація безпосередньо тих модифікацій, які були додані до базового алгоритму [11].

3.2 Зміна топології графу

Крім того, перехід до алгоритму із пріоритетною чергою дозволяє змінити топологію, що репрезентує граф. Таким чином, у поточному дослідженні було здійснено значний крок уперед шляхом розробки

універсальної модифікації, здатної працювати з будь-яким графом, що може бути представлений у вигляді матриці суміжності. На рис. 3.2 міститься схематичне зображення топології сітки та суміжності.

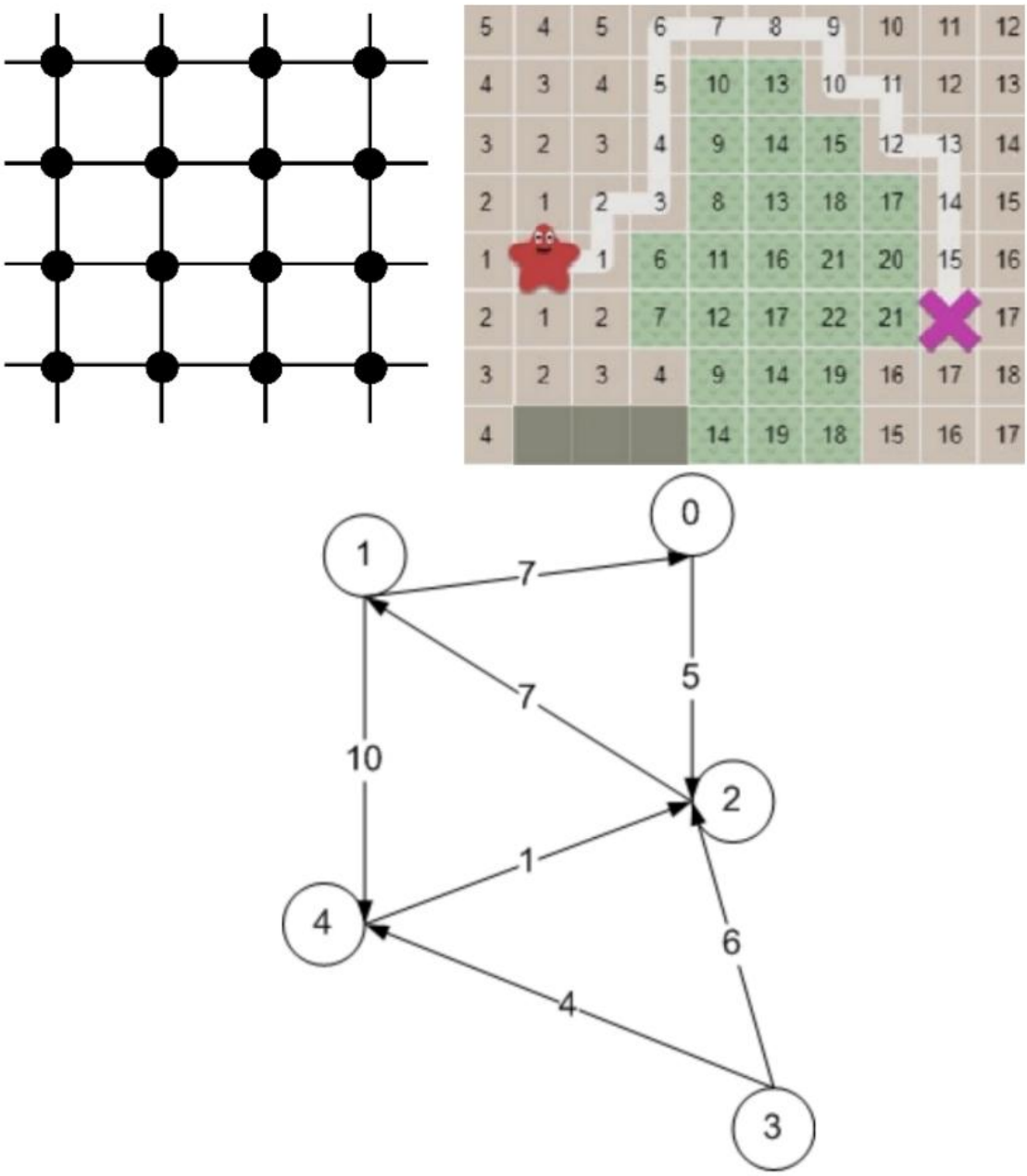


Рисунок 3.2 – Топологія сітки та суміжності

Різниці між цими топологіями відповідає різниця на програмному рівні. Для репрезентації матриці суміжності було створено новий класс Graph, у якому зберігається масив вершин. Для кожної вершині зберігається масив

ребер (лістинг 3.1). На рис. 3.3 наведено порівняння того, які типи матриць це дає.

```
struct MyEdge
{
    int to;
    int weight;
    int weight2 = {}; // dynamically added weight
    MyEdge() {}
    MyEdge(int to, int weight) : to(to), weight(weight) {}
    void read() {
        cin >> to >> weight;
    }
};

struct MyQueueVertex
{
    int number;
    int dist;
    MyQueueVertex(int number, int dist) : number(number),
dist(dist) {}
};

bool operator<(const MyQueueVertex& v1, const MyQueueVertex& v2)
{
    return v1.dist > v2.dist;
}

class MyGraph
{
         vector<vector<MyEdge>> link;
};
```

Лістинг 3.1 – Програмна репрезентація матриці суміжності

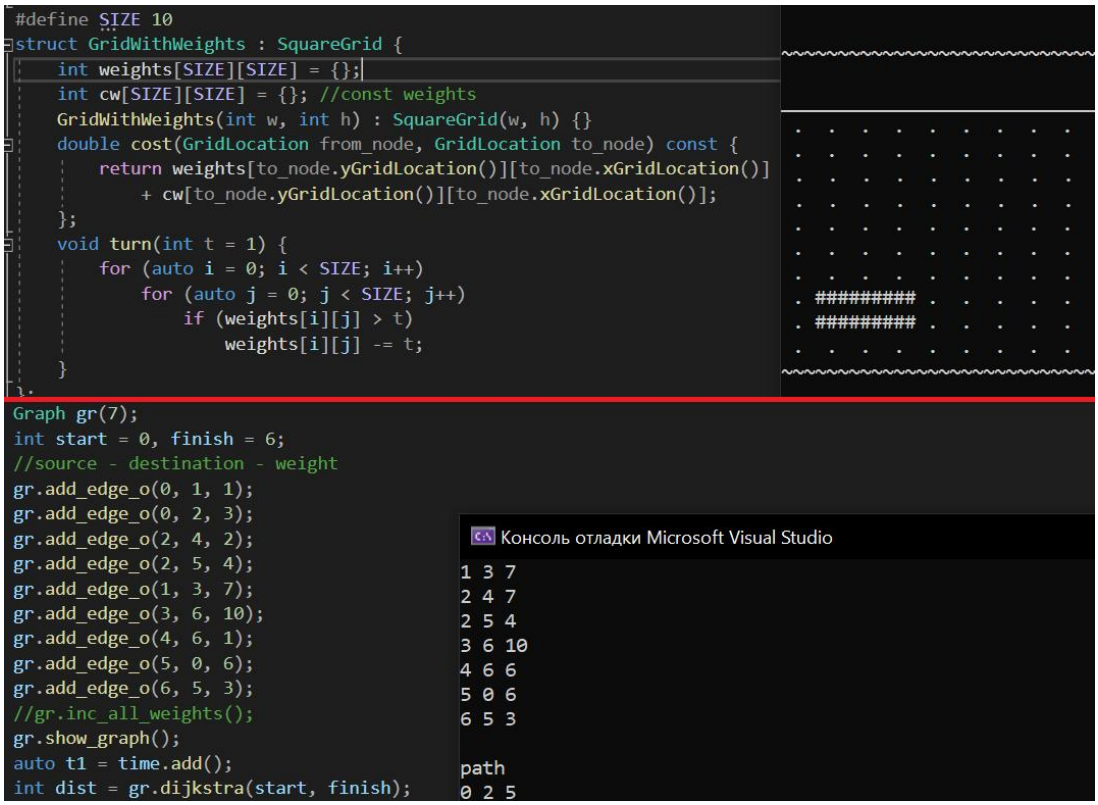


Рисунок 3.3 – Топологія сітки та матриці суміжності на програмному рівні; минула та оновлена версія розділені червоною лінією

Обмеження, які залишилися, є невід'ємними для алгоритму Дейкстри в цілому. До них відносяться вимоги щодо відсутності у графі циклів із від'ємною сумою. В таких ситуаціях алгоритм може застрягти в нескінченному циклі, оскільки завжди існує можливість зменшення ваги шляху, проходячи через цикл [12]. Також алгоритм передбачає повну інформацію про граф, тобто що відомі всі відстані між вузлами – всі ребра мають певне відоме значення.

3.3 Демонстрація роботи алгоритму на матриці суміжності

Створимо граф за рис 3.4. Нехай існують три агенті A, B. C, що бажають потрапити із точок A1, B1 , C2, розташованих на деяких вузлах, у

відповідні точки A2, B2, C2. Назва точок є умовною для зв’язку із агентами та не являється окремим вузлом чи сутністю.

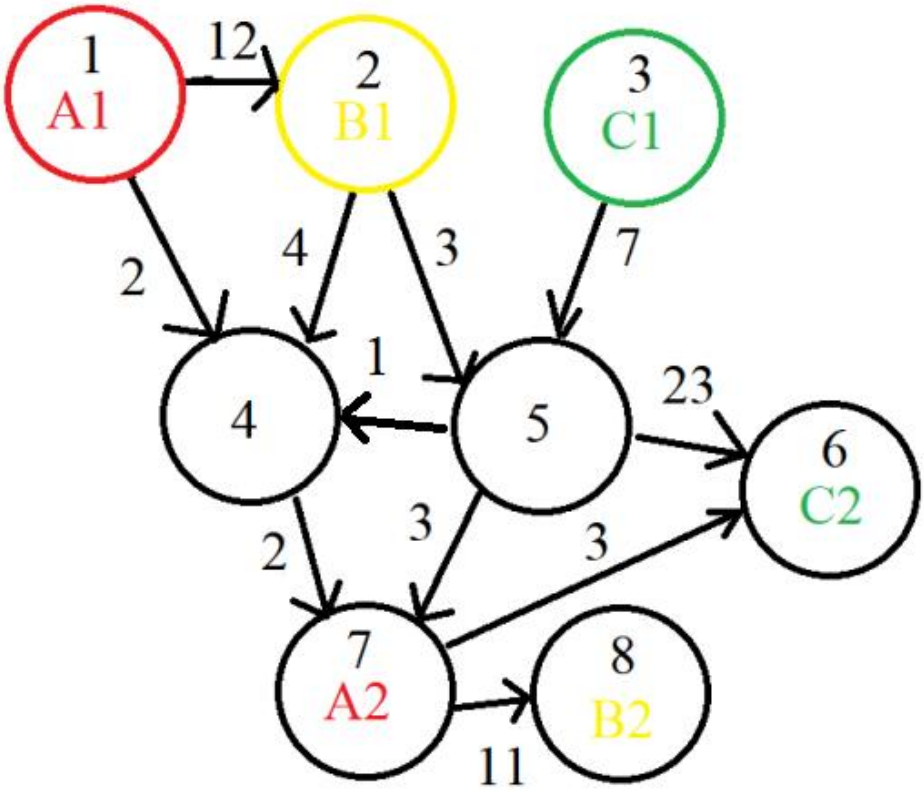


Рисунок 3.4 – Граф, що можна задати матрицею суміжності.

На рис. 3.4 видно, що найкоротший шлях від C1 до C2 пролягає через вузли 5 та 7. На рис. 3.5 наведено програмну репрезентацію рис.3.4. Для кожного з трьох агентів будується шлях згідно напівкооперативному алгоритму Дейкстри. Вага кожного використаного ребра збільшується на 25.

Оновимо схематичне зображення графу. Воно наведено на рис. 3.6. Слід окремо підкреслити, що підвищення ваг є умовністю, а не реальною зміною характеристик графу. Підвищення значення ребра з 3 до 28 не означає, що шлях через це ребро став фізично більш складним. У реальності це буде та ж сама дорога чи щось інше. Підвищення ваги ребра лише показує бажання агентів уникнути конфлікту за використання цього ребра.

```
Graph gr(10);
gr.add_edge_o(1, 2, 12);
gr.add_edge_o(1, 4, 2);
gr.add_edge_o(2, 5, 3);
gr.add_edge_o(2, 4, 4);
gr.add_edge_o(3, 5, 7);
gr.add_edge_o(4, 7, 2);
gr.add_edge_o(5, 1, 1);
gr.add_edge_o(5, 6, 23);
gr.add_edge_o(5, 7, 3);
gr.add_edge_o(7, 8, 11);
gr.add_edge_o(7, 6, 3);
gr.show_graph();
auto r = 0;
int start = 1, finish = 7;
r = gr.dijkstra(start, finish);
cout << "Price: " << r << '\n';
start = 2, finish = 8;
r = gr.dijkstra(start, finish);
cout << "Price: " << r << '\n';
start = 3, finish = 6;
r = gr.dijkstra(start, finish);
cout << "Price: " << r << '\n';
cout << '\n';
gr.show_graph();
cout << '\n';
```

```
Консоль отладк
1 2 12
1 4 2
2 5 3
2 4 4
3 5 7
4 7 2
5 1 1
5 6 23
5 7 3
7 8 11
7 6 3

path
1 4 7
Price: 4
path
2 5 7 8
Price: 17
path
3 5 6
Price: 30
```

Рис. 3.5. – Побудова графу за допомогою матриці суміжності

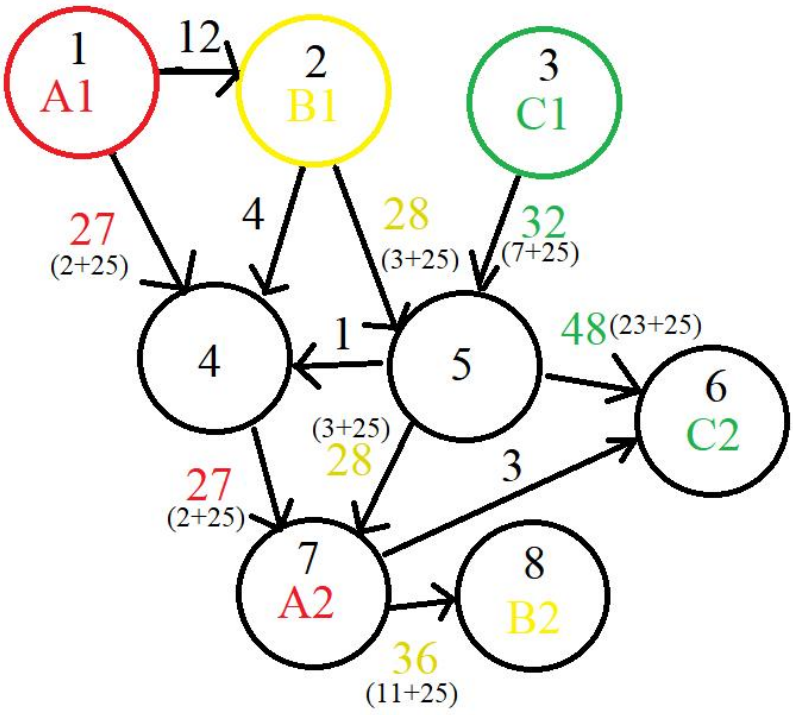


Рис. 3.6 – Оновлена схема графу. Кольоровими числами позначено загальну вагу, у дужках наведено спочатку базову вагу, потім додану. Загальна вага є сумою базових 2 із доданими 25 тощо

Видно, що шлях від C1 до C2 побудовано із врахуванням оновлених значень ребер між вузлами 5 і 7, що дозволило уникнути конфліктів між агентами. Шлях від C1 до C2 має вигляд 3->5->6 та вартість 7+23=30. Без підвищення ваг шлях 3->5->7->6 мав би меншу вартість 7+3+3=13. Але із урахуванням додаткових ваг він складає вже 7+(3+25)+3=38. Таким чином, хоча реальна (базова) вартість є оптимальною, третій агент C уникає конфлікту із агентом B, вибираючи субоптимальний шлях.

Продемонструємо функцію turn() на наведеному вище графі із матрицею суміжності. Викликаємо функцію turn() 7 разів поспіль, після чого знову надрукуємо граф. На рис. 3.7 видно, що значення ребер, які використали для побудови шляху, частково повернулися до попередніх (до початкових). Значення ваг, що не відвідувалися, залишилися сталими.

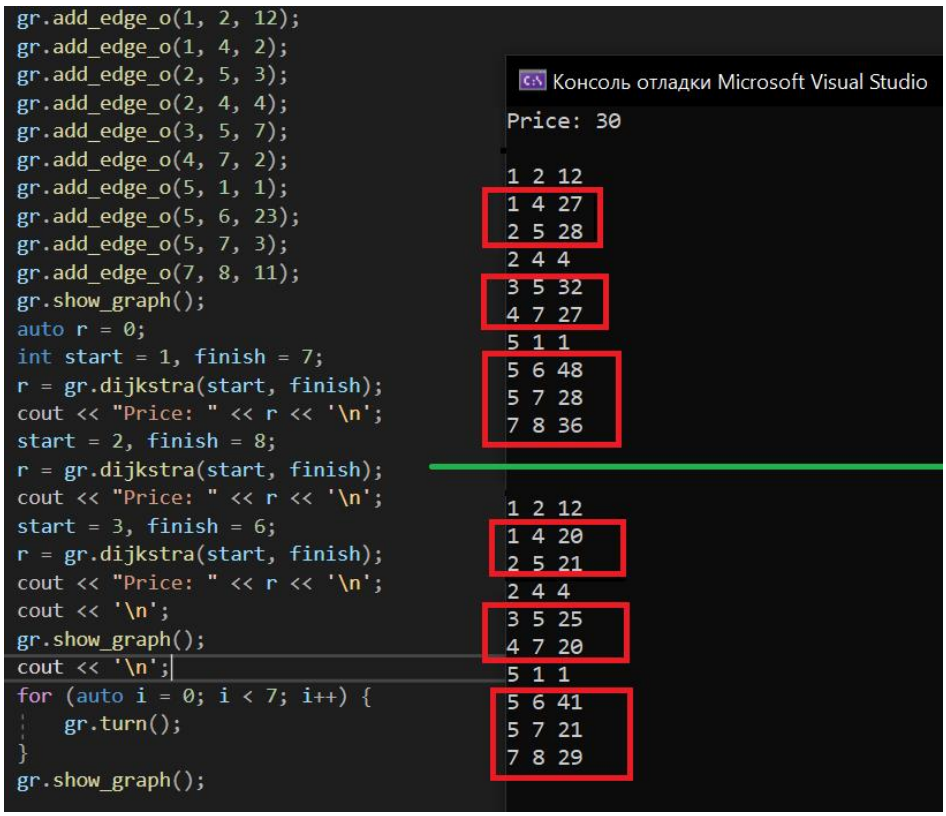


Рис. 3.7 – Результат функції turn(). Червоним відмічено відвідані ребра. Зеленою рискою відмічено граф до та після функції turn()

Тепер додамо в граф четвертого агента D, який намагається пройти шлях від точки D1 у вузлі 9 до точки D2 у вузлі 6.

Зобразимо це на рисунку 3.8.

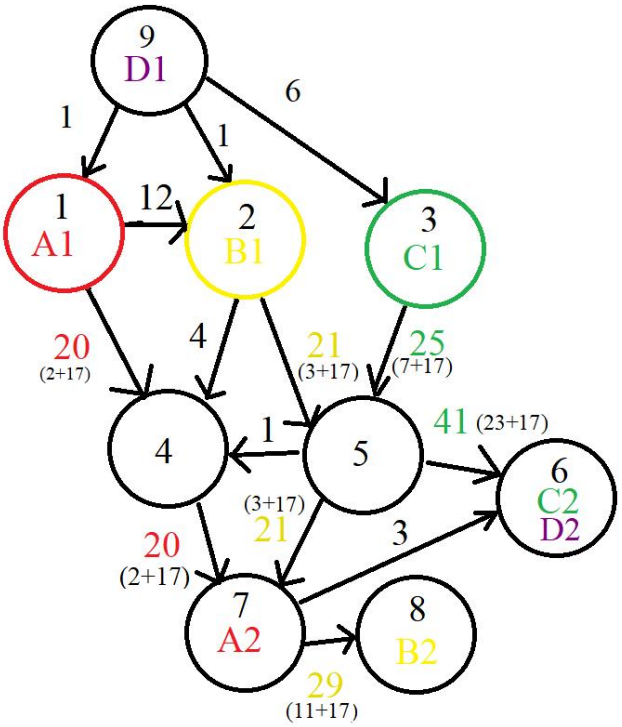


Рисунок 3.8 – Четвертий агент D

Побудуємо шлях з D1 до D2 (рис. 3.9).

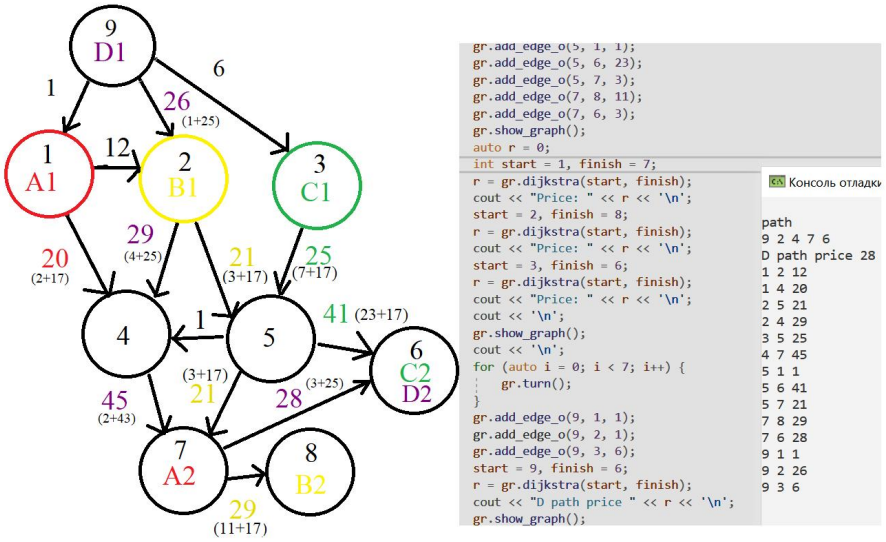


Рисунок 3.9 – Шлях для агенту D та відповідний програмний код

На рисунку видно, що шлях неможливо побудувати без конфліктів. Втім побудований графік містить лише 1 конфлікт, а саме ребро 4->7, яке вже було використано агентом A.

Порівняємо це зі сценарієм, у якому агент D побудує оптимальний, але більш конфліктний шлях. Для цього збільшено кількість ітерацій turn() з 7 до 30 перед побудовою шляху для агенту D. Це нівелює всі додані ваги, граф повернеться у початковий стан. Як наслідок, шлях для агента D буде фактично побудовано за звичайним алгоритмом Дейкстри, він не буде намагатися уникнути конфліктів із іншими агентами (рис. 3.10). Підрахуємо кількість конфліктів у цьому сценарії.

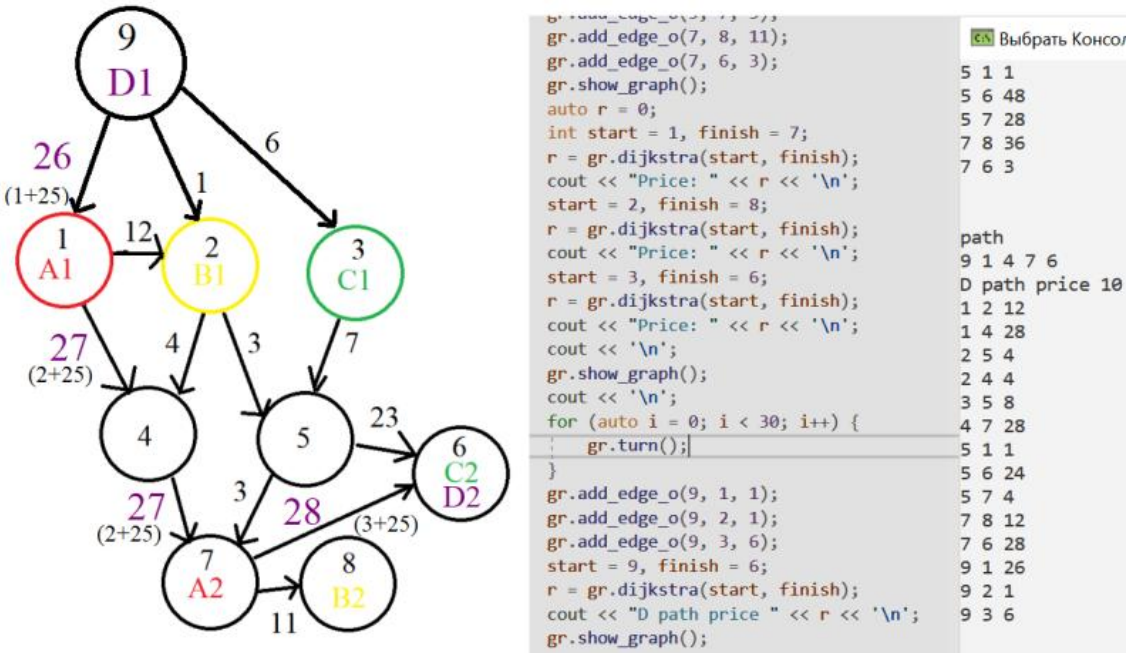


Рисунок 3.10 – Шлях агенту D за де-факто звичайному алгоритмі Дейкстри (без уникання конфліктів)

Видно, що тепер кількість конфліктів із агентом A вдвічі більша, оскільки тепер обидва агенти використовують ребра 1->4 та 4->7, у той час як у попередньому сценарії (рис.3.9) конфліктним було лише 4->7.

Також порівняємо базову ціну між цими шляхами. Перший (рис. 3.9) шлях виглядав як 9->2->4->7->6, тобто мав базову ціну (без урахування додаткових ваг) 1+4+2+3=10.

Другий шлях (рис. 3.10) виглядає як 9->1->4->7->6, відповідно його ціна 1+1+2+2+3=10. У даному випадку уникання конфлікту не спричинило отримання субоптимального результату.

Підведемо підсумки (рис. 3.9, рис. 3.10, рис. 3.6) у таблиці 3.1.

Таблиця 3.1 – Порівняння шляхів та кількості конфліктів

Маршрут	Вартість шляху (із уникання конфліктів)	Кількість конфліктів	Оптимальна вартість	Кількість конфліктів	Вартість шляху при turn(12)	Кількість конфліктів при turn(12)
C1->C2	30	0	13	1	13	Невідомо
D1->D2	10	10	1	2	10	Невідомо

Таким чином, функція turn() дозволяє гнучко балансувати між мірою уникання конфліктів та оптимальністю результатів. Додаткові ваги зменшують кількість конфліктів ціною отримання субоптимальних маршрутів. Втім, погіршення не є гарантованим – альтернативний шлях може мати дуже схожу або навіть ідентичну вартість. Загалом можна сказати, що функція turn() продемонструвала велику важливість для симуляції плину часу у рамках наявного алгоритму.

Можна зробити висновок, що наявний алгоритм коректно працює із використанням більш ефективного способу представлення графу. Це слугує фундаментом для подальшої оптимізації та більш масштабних тестів.

4 ВЕКТОРИЗАЦІЯ ОКРЕМИХ ЕТАПІВ АЛГОРИТМУ

4.1 Аналіз функції імітації плину часу

Плин часу симулюється за допомогою функції turn(), приведеної на лістингу 4.1. Функція turn() забезпечує покрокове повернення системи до початкового стану. В цьому контексті початковим станом графу розуміється сукупність значень ваги графу до впровадження напівкооперативного алгоритму, тобто до збільшення значень ваги у відвіданих вершинах. Таким чином, вагою вважається сума ваг вершин з однаковими координатами з основного (базового) та додаткового масивів. Додатковий масив, необхідний для коректної роботи функції turn(), не потребує значної кількості пам'яті, оскільки його розмір збігається з кількістю вершин у графі, уникаючи можливого експоненційного збільшення використовуваної оперативної пам'яті у деяких інших кооперативних алгоритмах.

Подібні функції часто зустрічаються у ігрових рушіях та симуляції у реальному часі [13]. Зазвичай у подібних ігрових або симуляційних системах процес оновлення стану відбувається з певною фіксованою частотою, відомою як «тік» (tick), яка визначає кількість оновлень стану за одиницю часу.

```
void turn(int weight = 1) {
  for (auto i = 0; i < link.size(); i++)
      for (auto j = 0; j < link[i].size(); j++) {
          if (link[i][j].weight2 - weight > 0) {
              link[i][j].weight2 -= weight;
          }
          else {
              link[i][j].weight2 = 0;
          }
      }
```

Лістинг 2.1 – функція turn()

У зазначеній функції реалізовано зміну ваги ребра weight2 для кожного ребра у графі. Використовуючи умову if, вага weight2 зменшується на вказане значення weight, але ніколи не стає від'ємною, а стає рівною 0 у випадку, якщо після зменшення вага стає від'ємною. Оцінимо поточну швидкодію цієї функції порівняно із безпосередньо функцією пошуку шляху dijkstra().

Для цього заміряємо відсоток часу, який витрачається на один окремий виклик turn() порівняно із dijkstra(). Для втілення цієї мети створимо графи з різною кількістю вершин та ребер. Поділимо час, що витрачається на виклик функції turn() на час, що витрачається на виклик функції dijkstra(), та помножимо на 100, щоб отримати відсотки. Результати наведено у таблиці 4.1.

Таблиця 4.1 – функція turn() займає близько 10% від dijkstra()

Кількість вершин	Кількість ребер	Відсоток
100	500	11.30
100	1500	17.19
1000	5000	7.15
1000	15000	8.88
10000	50000	8.37
10000	150000	10.67

Ці ж сами дані можна зобразити на рисунку 4.1. Зробимо шкалу логарифмічною, щоб коректно відражати масштаб.

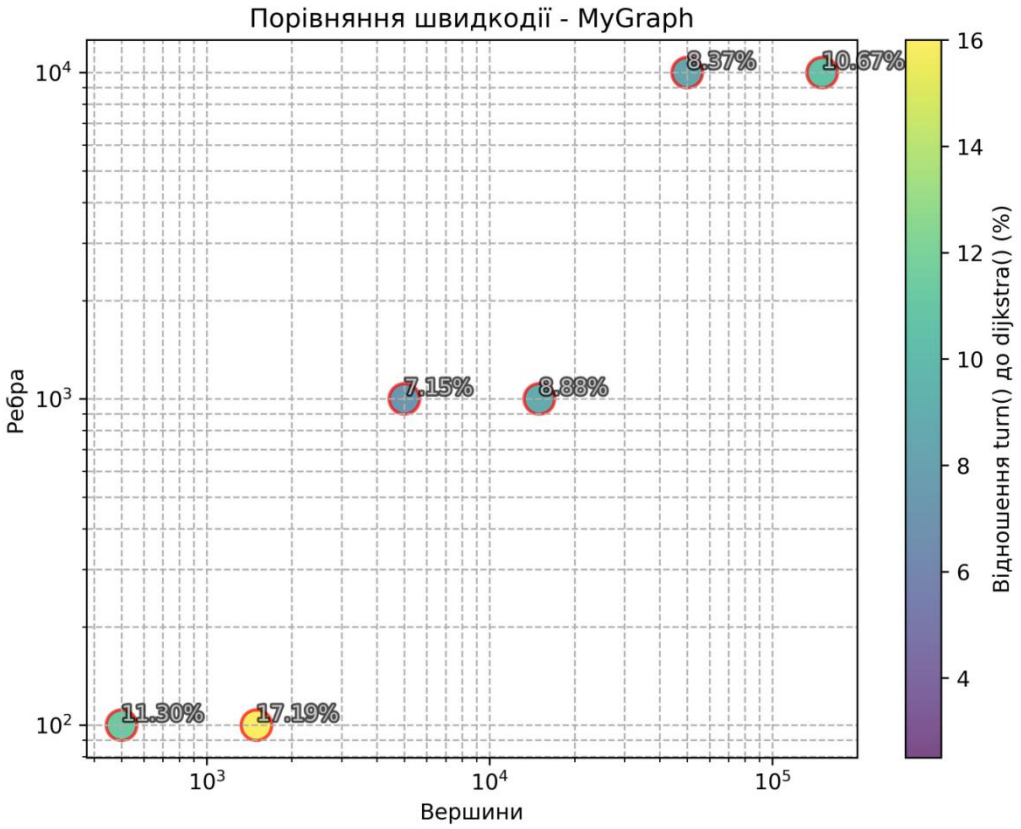


Рисунок 4.1 – функція turn() займає близько 10% від dijkstra()

Важливо відзначити, що функція turn() за вказаними параметрами може викликатися 20-60 разів на секунду (відповідно до кількості тіків у секунду у грі чи симуляції). Оскільки вона займає близько 10% часу від функції dijkstra(), яка, відповідно, є ключовою для пошуку шляху в графі, то існує багато сценаріїв, у яких вплив цієї функції на швидкодії буде значним. З цього випливає необхідність оптимізації цієї функції або алгоритму в цілому, якщо це можливо.

4.2 Залежності даних у алгоритмі Дейкстри

Алгоритм Дейкстри призначений для знаходження найкоротших шляхів в графі, і в класичній формі його важко векторизувати через декілька причин.

Алгоритм Дейкстри має сильні залежності даних між ітераціями [14] Кожна вершина обробляється окремо, і відомості про найдешевший шлях до інших вершин не можуть бути використані паралельно. Відвідування наступної вершини вимагає інформації щодо відвідання попередньої. Причому вимагає як інформації щодо вдалого, так і невдалого відвідування вершини. У класичному алгоритмі Дейкстри один найкоротший шлях. Якщо їх декілька, то вибирається перший, що був оброблений алгоритмом [15]. Це унеможливлює можливість ефективної векторизації. Крім того, це унеможливлено використанням пріоритетної черги, оскільки в ній пріоритети можуть бути переоцінені на кожній ітерації алгоритму.

Проте окремі кроки із розробленої напівкооперативної модифікації алгоритму Дейкстри можуть бути ефективно векторизовані. До таких задач відноситься симуляція плину часу turn() – її можна ефективно векторизувати. Прискорення цієї функції, як одного із кроків алгоиртму, що може витрачати значну долю часу, може поліпшити швидкодію алгоритму в цілому.

4.3 Можливості пришвидшення алгоритму

Векторизація є ефективним методом для обробки як великих, так і відносно невеликих масивів даних на відміну від багатопоточності, яка підходить лише для великих масивів даних [16].

AVX (Advanced Vector Extensions) є розширенням набору інструкцій x86 процесорів, що підтримують векторні операції з довжиною в 256 або 512 бітів [17]. Це дозволяє використовувати одну інструкцію для роботи з більшою кількістю даних одночасно, тобто являється різновидом SIMD (single instruction multiple data) інструкцій. Векторизація за допомогою AVX є ефективним методом для оптимізації обробки невеликих масивів.

SIMD-інструкції дозволяють зменшити кількість інструкцій, необхідних для виконання того ж самого обсягу обчислювальної роботи

(рис. 4.2) [18]. Використання AVX-інструкцій дозволяє ефективно використовувати апаратні можливості сучасних процесорів та суттєво підвищити продуктивність [19], у тому числі у випадку задач на графах [20].

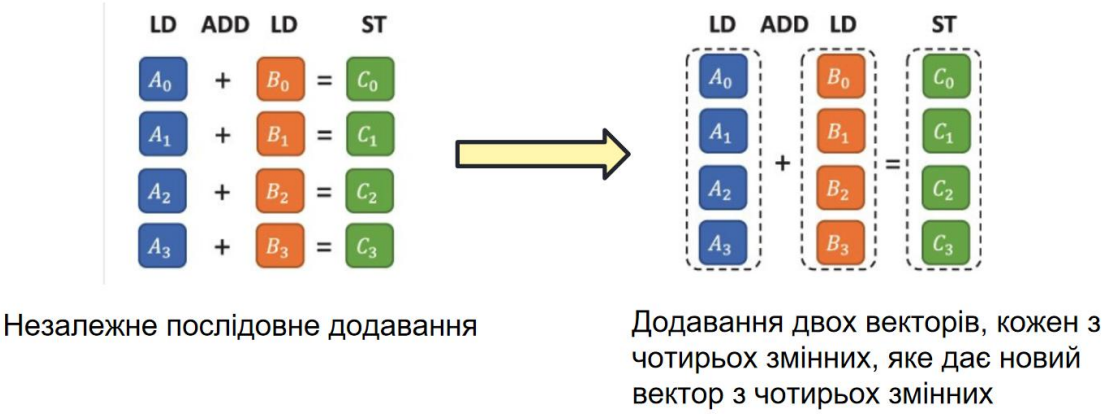


Рисунок 4.2 – Одна AVX-інструкція здатна замінити 4 поштучні додавання чотирьох послідовних елементів

Для графів характерно багато невеликих масивів. Прикладом може слугувати окремий вузол графу, який має 10, 20, але не тисячу ребер. Відповідно використання «тонкозернистої» паралелізації є оптимальним шляхом прискорення функції turn() у наявному алгоритмі. Наприклад, якщо у окремого вузла є 8 ребер із змінними типу int, то ці значення можуть бути записані у один 256-бітний вектор. Таким чином, векторизація може прискорити функцію turn().

Розглянемо, який спосіб векторизації є оптимальним. Можна нарахувати декілька способів векторизації на x86 процесорах [16]:

1) асемблерний код;

2) ручна векторізація за допомогою інтриксиків (intrinsics);

3) автоматична векторизація компіляторами;

4) ручна векторизація за допомогою зовнішніх бібліотек.

Асемблерний код – це найкращий спосіб оптимізації коду. В асемблері можливо майже все. Але це є і проблемою. Існують тисячі різних інструкцій, і досить важко запам'ятати, яка інструкція належить до якого розширення

набору інструкцій. Код на асемблері важко документувати, важко налагоджувати і важко підтримувати.

Інтринсики лише дещо краще асемблерного коду. Вони підтримуються більш високорівневими мовами програмування, такими як C++, Java та C#. Крім того, на відміну від асемблерного, код що використовує інтринсики може бути додатково оптимізований компілятором. Проте інтринсики зберігають близький до асемблера синтаксис та мають дуже важкі для читання імена функцій.

Автоматична векторизація компіляторами є найбільш поширеним на сьогодні способом векторизації лінійного коду. Хороший компілятор здатен перетворити лінійний код на векторний. Проте автоматична векторизація не завжди спрацьовує та не завжди спрацьовує ефективно. Вона здатна створити неоптимальний код, коли алгоритм занадто складний, структури даних недостатньо упорядковані, коли компілятору невідома деяка інформація, яка відома програмісту, коли алгоритм має багато розгалужень та так далі.

Існує чотири популярних C++ компілятора: MSVC, GNU, Clang та Intel Compiler. Їх здатність векторизувати код сильно відрізняється один від одного, що додає ще одну складність для автоматичної векторизації.

Ручна векторизація за допомогою зовнішніх бібліотек поєднує переваги ассемблерного коду та інтринсиків, уникаючи їх недоліків. Для цих цілей було обрано VCL (Vector Class Library), створену Агнером Фогом [21].

Вона використовує всі наявні переваги C++: низькорівневе програмування, таке як біт-маніпуляції та інтринсики, функції програмування високого рівня, такі як класи та шаблони, перевантаження операторів, метапрограмування, компіляція в машинний код без будь-якого проміжного байт-коду. C++ є одним із найкраших виборів для написання сучасного та високоефективного коду [22].

Бібліотека VCL підтримується для всіх популярних компілятів, таких як GNU, Clang, MSVC та Intel, що дозволить коректно порівняти ефективність їх роботи.

4.4 Параметри компіляції

За оптимізацію швидкодії в усіх популярні C++ компілятори відповідає всього декілька з них [23]. До них відносяться:

1) -On (зазвичай -O2, рідше -O3). Загальний рівень оптимізації, включає в себе багато оптимізацій щодо використання регістрів та операций читання/запису у регістрах;

2) -msse, -mavx. Відповідають за включення генерації відповідних SSSE та AVX-інструкцій. Відрізняються версіями (є SSE, SSE2 і т.д.);

3) -Oi. Включити підстановку функцій (inline). Нівелює штраф за виклик невеликих функцій.

Всі подальші експерименти буде виконано із використанням цих параметрів.

4.5 Прискорення функції імітації плину часу

Розглянемо ассемблерний код, сгенерованний компілятором Clang для коду із оригінальної роботи (лістинг 4.1). Повний код сгенерованного assembler коду наведено у додатку А.

```
turn(int) [clone .constprop.0]:
        sub     rsp, 8
        mov     r10, QWORD PTR [rdi]
        mov     rsi, QWORD PTR [rdi+8]
        xor     r8d, r8d
        mov     rdi, r10
        mov     r9, rax
        mov     rsi, rdx
        vpxor   xmm1, xmm1, xmm1
        add     rdx, 1
        sub     DWORD PTR [rax+8], 1
        vmovd   xmm0, DWORD PTR [rax+8]
        vpmaxsd xmm0, xmm0, xmm1
        vmovd   DWORD PTR [rax+8], xmm0
        cmp     rcx, rdx
```

Лістинг 4.1 – Автоматична векторизація оригінального коду

Видно, що код фактично не векторизований – використовуються лише окремі скалярні (не векторні) версії SSE-інструкцій, які менш продвинуті порівняно із AVX, хоча і більш поширені.

Загалом цей результат є цілком очікуваним. У цьому прикладі дизайн зберігання даних графа є неоптимальним. Значення ваг є розпиленними у якості окремих значень, а не зібрані у суцільний масив, що зберігається у пам’яті послідовно. Спробуємо усунути цей недолік. Для цього ми змінимо дизайн зберігання даних – тепер для кожного вузла дані про основні та додані ваги будут зберігатися суцільним масивом (лістинг 4.2).

```
struct Edge
{
    vector<int> to;
    vector<int> weight;
    vector<int> weight2;
}
class Graph {
    vector<Edge> link;
struct MyEdge {
    int to;
    int weight;
    int weight2 = {}; // dynamically added weight

class MyGraph {
    vector<vector<MyEdge>> link;}
```

Лістинг 4.2 – Виправлений дизайн структур даних

Компілятори, такі як Clang та GCC, володіють автовекторизаційними здатностями для ефективної роботи з операціями, які можуть бути векторизовані. Компілятори автоматично аналізують програмний код, щоб виявити ланки, які можна векторизувати. Теоретично, ці зміни у лістингу 4.2 мають дозволити компіляторам автоматично генерувати векторний код.

Оновлені результати генерації компілятором Clang наведено у лістингу 4.3.

```
void turn2(int weight = 1) { //vectorizable turn
            for (auto& edge: link) {
                for (auto i = 0; i < edge.weight2.size();i++) {
                    if (edge.weight2[i] > weight) {
                        edge.weight2[i] -= weight;
                    }
                    else {
                        edge.weight2[i] = 0;
                    }
                }
            }
        }

    Graph::turn2(int) [clone .constprop.0]:
    .L364:
            vpcmpeqd        xmm3, xmm3, xmm3
            vmovd   xmm1, eax
            vpcmpeqd        ymm2, ymm2, ymm2
            vpshufd xmm4, xmm1, 0
            vpbroadcastd    ymm1, xmm1
            vpmaxsd ymm0, ymm1, YMMWORD PTR [rax]
            add     rax, 32
            vpaddd  ymm0, ymm0, ymm2
            vmovdqu YMMWORD PTR [rax-32], ymm0
```

Лістинг 4.3 – Результати роботи компілятора для оновленого дизайну

Якість сгенерованного компілятором ассемблерного коду значно покращилась. Деякі із інструкцій є SSE-інструкціями, деякі навіть є AVX-інструкціями. Втім цілком можливо, що ручна векторизація буде ще краще.

Із допомогою VCL було переписано ту ж саму функцію turn(). Таким чином, наведена у лістингу 4.4 функція напряму компілюється у векторний

код. Vec8i відповідає 256-бітним AVX-векторам, які зберігають по 8 змінних типу integer.

```
   void turn2(int weight = 1) {
        const Vec8i w(weight);
        for (auto& edge : link) {
            std::size_t size = edge.weight2.size();
            auto i = 0;
            Vec8i d, t, r, m;
            for (; i + 8 <= size; i += 8) {
                d.load(&edge.weight2[i]);
                m = d > w;
                r = if_sub(m, d, w);
                r = r & m;
                r.store(&edge.weight2[i]);
            }
.L147:
        mov     rcx, QWORD PTR [rsi+48]
        lea     rcx, [rcx+rdx*4]
        vmovdqu ymm3, YMMWORD PTR [rcx]
        vpcmpgtd        ymm0, ymm3, ymm2
        vpand   ymm1, ymm0, YMMWORD PTR [rsp+64]
        vmovdqu ymm5, YMMWORD PTR [rcx]
        vpsubw  ymm1, ymm5, ymm1
        vpand   ymm0, ymm0, ymm1
        vmovdqu YMMWORD PTR [rcx], ymm0
```

Лістинг 4.4 – Результат компіляції переписаної із використанням інструкцій функції

У наведеному в лістингу 4.4 ассемблерному коді переважають інструкції із латинським префіксом v (vector), що оперують на ymm-регістрах, які відповідають AVX-інструкціям. Це повністю відповідає очікуваному результату. Загалом число інструкцій зменшилось, а кількість оброблюваних за одну операцію даних збільшилась – це має суттєво прискорити виконання функції.

5 ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕЗУЛЬТАТІВ ВЕКТОРИЗАЦІЇ

5.1 Порівняння швидкодії

Повторимо експеримент, зображений на рисунку 4.1, але на цей раз використовуючи оновлену та оптимізовану функцію, з метою отримати відносні показники покращення швидкодії функції. На рис. 5.1. можна побачити, що оптимізована функція стабільно займає 3-5.5% часу замість 8-17% на рис. 4.1. Це є значним прискоренням відносних показників.

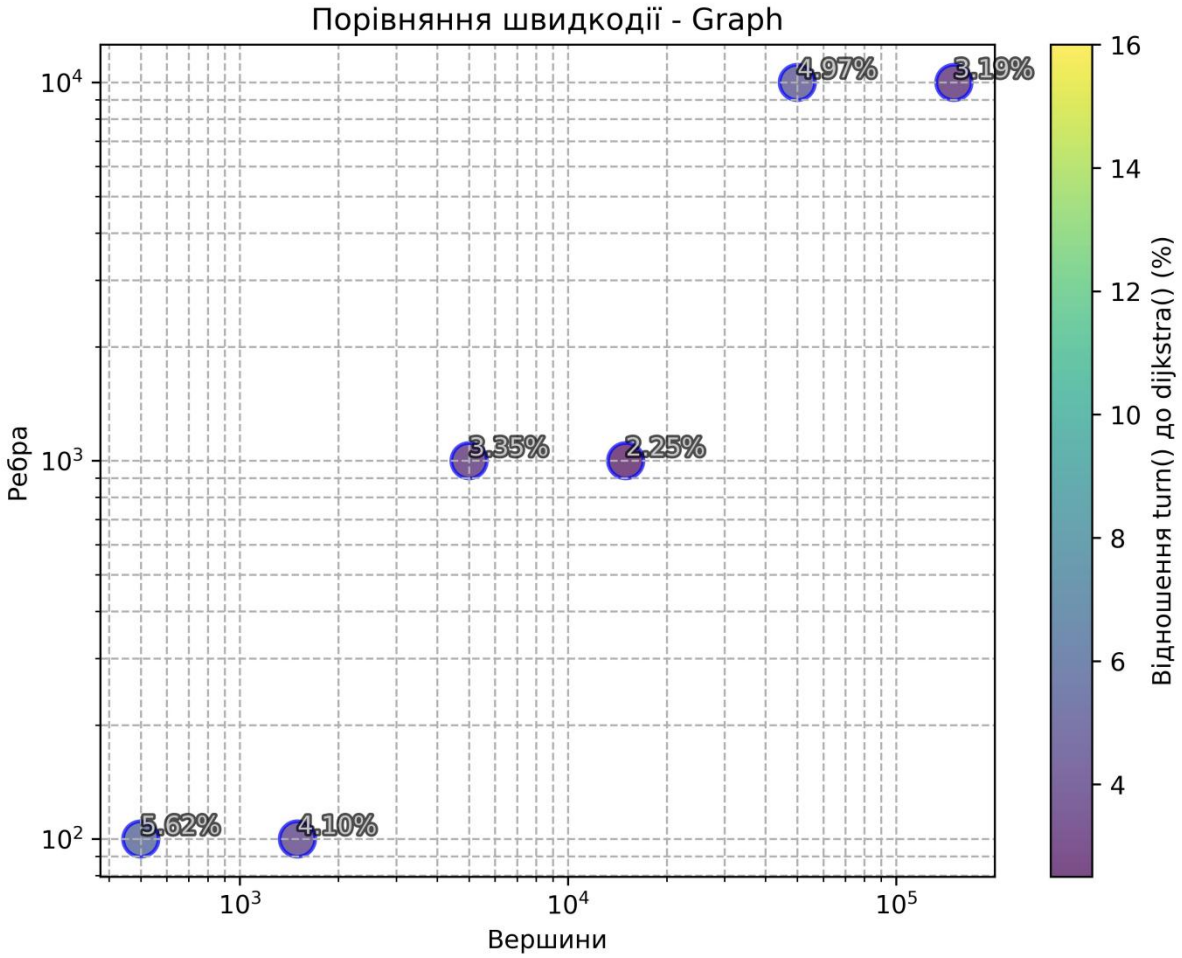


Рисунок 5.1 – Оптимізована функція turn2() займає близько 3.5% від dijkstra() замість 10%

Поглянемо на те, як показники змінились в абсолютних значеннях. Тестування здійснювалося із використанням усіх популярних компіляторів. У таблиці 5.1 наведено результат автовекторизації старої версії алгоритму та вручну векторизованного коду. Значення наведені у наносекундах. Флаги оптимізації були виставлені максимальними (-O2 -Oi), кількість вузлів – 100, кількість ребер – 1500. Тест здійснювався шляхом виклику функції turn2() у 1000 незалежних тестів.

Таблиця 5.1 – Порівняння ручної та автоматичної векторизації

	MSVC	GNU gcc/g++	LLVM Clang	Intel Compiler Classic
Ручна векторизація, AVX2, час (нс)	9735	8677	8809	9949
Автоматична векторизація, AVX2, час (нс)	33453	15543	17483	33730
Прискорення, разів	3.5x	1.8x	2x	3.4x
Автоматична векторизація за замовчанням (SSE2), час (нс)	34071	18184	25402	34275
Прискорення, разів	3.5x	2.2x	2.5x	4.3x

Автоматична векторизація протестована для двох типів: автоматична векторизація за замовчанням та із мануально виставленим флагом AVX2. SSE2 наявний в усіх 64-бітних x86-процесорах, тому є базовим флагом за

замовчанням для всіх компіляторів. AVX2 є найбільш новітнім із доступних на використаному обладнанні набором інструкцій. Із наведеної таблиці видно, що ручна векторизація дозволяє досягти значно кратно кращих результатів порівняно із початковою автовекторизацією. Це пояснюється тим, що ручна векторизація здатна повністю використати всі 256 біт, що доступні у FPU-регістрах.

Проведемо додаткове тестування на більш великому графі, кількість вузлів в якому 3000, а ребер – 15000. Результати наведено у таблиці 5.2. Проведене тестування на більшому графі з підвищеним обсягом даних також підтверджує ефективність ручної векторизації навіть при збільшенні розміру вхідних даних. У всіх тестованих випадках ручна векторизація демонструє кращі результати порівняно з автоматичною векторизацією, що робить її привабливим варіантом для оптимізації напівкооперативного мультиагентного алгоритму пошуку шляху.

Таблиця 5.2 – Порівняння ручної та автоматичної векторизації на більш великому графі

	MSVC	GNU gcc/g++	LLVM Clang	Intel Compiler Classic
Ручна векторизація, час (нс)	259857	244423	239743	301957
Автоматична векторизація, AVX2, час (нс)	906453	464403	472420	885121
Прискорення, разів	3.5x	1.9x	2x	2.7x

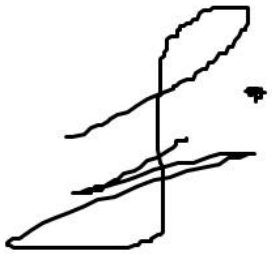
GNU GCC/g++ та clang демонструють помітно кращі результаті, аніж компілятори MSVC та Intel. Це пояснюється тим, що на відміну від них вони сгенерували немало SSE та деякі AVX-інструкції. Тим не менш вони все ще суттєво поступаються результатам ручної векторизації [17].

Загалом ручне застосування AVX інструкцій дозволило досягти прискорення функції turn() порівняно із компіляторами GCC та Clang, а також прискорення у 3.5 рази порівняно із компіляторами MSVC та Intel. Повний код використаної у роботі програми та тестів наведено у додатку Б.

Теоретичний максимум прискорення 32-бітних змінних типу integer за допомогою 256-бітних регістрів становить 8 разів[16, 22]. Можна зробити висновок, що компілятори обмежено ефективні у векторизації коду, а ручна векторизація дозволяє наблизитися до оптимального результату настільки, наскільки це можливо. Крім того, експериментально підтверджено, що окремі елементи запропонованої модифікації можуть бути ефективно векторизовані на сучасних x86 процесорах. Оскільки симуляція плину часу за допомогою функції turn() залежно від сценарію може займати від 10% до 100-200% часу та більше порівняно із безпосереднім обчисленням пошуку шляху, то прискорення у 2-3.5 разів є гарним результатом, який дає можливість більш гнучкого та широкого застосування напівкооперативного алгоритму Дейкстри.

ВИСНОВОК

За результатами аналізу у кваліфікаційній роботі реалізовано векторизацію за допомогою AVX-інструкцій окремих етапів наявного напівкооперативного MAPF-алгоритму. Шляхом порівняльного аналізу з’ясовано, що популярні компілятори розкривають лише частину потенціалу векторизації. Проведена власноруч векторизація демонструє кратно кращі показники порівняно із автоматичною. З’ясовано, що не весь алгоритм може бути ефективно векторизовано. Разом з тим експериментально продемонстровано помітне підвищення загальної швидкодії алгоритму завдяки прискоренню окремих кроків алгоритму.



/Коган В.В./

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Andreiana A.D. An Experimental Comparison of Implementations of Dijkstra's Single Source Shortest Path Algorithm Using Different Priority Queues Data Structures. / A.D. Andreina, C. Bădică, E. Ganea.. // 24th International Conference on System Theory, Control and Computing (ICSTCC). Sinaia, Romania, 2020. – PP. 120-124
2. Foead D. A Systematic Literature Review of A* Pathfinding [Електронний ресурс] – Режим доступу: https://www.sciencedirect.com/science/article/pii/S1877050921000399 – 2023.11.18
3. Silver D. Cooperative Pathfinding // Proceedings of the First Conference on Artificial Intelligence and Interactive Digital Entertainment, 2005
4. Roni S. Multi-Agent Path Finding – An Overview. [Електронний ресурс] – Режим доступу: https://www.researchgate.net/publication/336611576_Multi-Agent_Path_Finding_-_An_Overview – 2023.12.01
5. Roger G.. Non-optimal Multi-agent Pathfinding is Solved. / G. Roger, M. Helmert // Workshop on Multi-agent Pathfinding AAAI, 2012. – PP. 9-15.
6. Yu J.. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. / J. Yu, S. LaValle // Artificial Intelligence. – 2013. – № 27(1). – PP. 1443-1449
7. G. Sharon. Conflict-based search for optimal multi-agent pathfinding / G. Sharon, R. Stern, A. Felner, N.R. Sturtevant // Artificial Intelligence. – 2015. – № 219. – PP. 40-66
8. Sharon G. The increasing cost tree search for optimal multi-agent pathfinding. / G. Sharon, R. Stern, M. Goldenberg, A. Felner // Artificial Intelligence. – 2015. – № 219. – PP. 470-495

9. Коган В.В. Методи пошуку оптимальної стратегії поведінки у системі дорожнього трафіку / В.В. Коган, В.Г. Пенко // – Одесса: ОНУ, 2022
10. Пенко В.Г. Підхід до усунення конфліктів у мультиагентних системах на основі алгоритму Дейкстри / В.Г. Пенко, О.В. Пенко, В.В. Коган. // Інформатика та математичні методи в моделюванні. – 2023. –Том 13, № 1-2 - С. 287-295
11. Коган В.В. Алгоритм Дейкстри. Методи удосконалення алгоритму півкооперативного пошуку шляху / В.В. Коган, В.Г. Пенко // Інформатика, інформаційні системи та технології: тези доповідей XX Всеукр. конференції студентів і молодих науковців. Одеса, 28 квітня 2023 р. – Одеса, 2023. – С. 148-152
12. LaValle S. Planning Algorithms // Cambridge: Cambridge University Press, 2006. – PP.36-37
13. Change G. On Determinism of Game Engines Used for Simulation-Based Autonomous Vehicle Verification. / G. Change, A. Ghobrial, R. McAreavey and others. // Transactions on Intelligent Transportation Systems. – 2022. – № 23(11). – PP. 1-15.
14. Stefano G.D. On the Implementation of Parallel Shortest Path Algorithms on a Supercomputer. / G.D. Stefano, A. Petricola, C. Zaroliagis // Parallel and Distributed Processing and Applications, 4th International Symposium, ISPA. December 6, 2006, Sorrento, Italy.
15. Cormen T.H. Introduction to Algorithms / T.H. Cormen, C.E. Leiserson, R. L. Rivest, C. Stein // Cambridge: The MIT Press, 2009. – PP. 620-636
16. Fog A. The microarchitecture of Intel, AMD, and VIA CPUs. [Електронний ресурс] – Режим доступу: https://agner.org/optimize/microarchitecture.pdf. – 2023.11.26.
17. Kusswurn D. Modern X86 Assembly Language Programming // Berkleley. Appres , 2014. – PP. 557-562

18. Gueron S. Fast Quicksort Implementation Using AVX Instructions / S. Gueron, V. Krasnov // The Computer Journal. – 2015. –№ 59(1). – PP. 83-90
19. Hwancheol J. Performance of SSE and AVX Instruction Sets / J. Hwancheol, K. Sunghoon, L. Weonjong, M. Seok-Ho // Contribution to proceedings of the 30th International Symposium on Lattice Field Theory, June 29, 2012, Liverpool, England
20. Hossain M. Impact of AVX-512 Instructions on Graph Partitioning Problems / M. Hossain, E. Saule // 50th International Conference on Parallel Processing Workshop. – 2021. – № 33. – PP. 1-9
21. Fog A. C++ vector class library manual [Електронний ресурс] – Режим доступу: https://www.agner.org/optimize/vcl_manual.pdf – 2023.11.26
22. Коган В.В. Методи пришвидшення розрахунку мел-кепстральних коефіцієнтів. Матеріали XXI Міжнародної науково-практичної конференції «Шевченківська весна» / В.В. Коган, В.Г. Пенко // Матеріали XXI Міжнародної науково-практичної конференції «Шевченківська весна – 2023», 14 квітня 2023 р., Київ, Україна, Київський національний університет імені Тараса Шевченка – С.88
23. Mingxuan Z. Compiler Auto-tuning through Multiple Phase Learning / Z. Mingxuan, H. Dan, C. Junjie. // ACM Trans. Soft. Eng. Methodol. – 2023. – № 1. – PP.1-5

ДОДАТОК А. Код програми напівкооперативного алгоритму Дейкстри із векторізацією

```
#include <stdio.h>
#include <iostream>
#include <chrono>
#include <array>
#include <valarray>
#include <future>
#include <numeric>
#include <thread>
#include <vector>
#include <math.h>
#include <algorithm>
#include <functional>
#include <queue>
#include <climits>
#include <random>
#include "vcl/vectorclass.h"
#define INF INT_MAX
#define WEIGHT 225
#define TURNS 25
using namespace std;


typedef std::chrono::steady_clock::time_point tp;
using namespace std;
class Time {
public:
    static void show(tp t1, tp t2) { //time passed since t1
        std::cout <<
std::chrono::duration_cast<std::chrono::nanoseconds>(t2 -
t1).count() << '\t';
```

```
        printf("nanoseconds\n");
    }
    static void showMils(tp t1, tp t2) { //time passed since t1
        std::cout <<
std::chrono::duration_cast<std::chrono::milliseconds>(t2 -
t1).count() << '\t';
        printf("milliseconds\n");
    }
    static auto count(tp t1, tp t2) { //time passed since t1
        return
std::chrono::duration_cast<std::chrono::nanoseconds>(t2 -
t1).count();
    }
    tp add() {
        tp p = std::chrono::steady_clock::now();
        return p;
    }
};
template<typename F, typename... Ts>
inline auto
reallyAsync(F&& f, Ts&&... params)
{ //C++14 //This function receives a callable object f and zero
or more param
    return std::async(std::launch::async,
        std::forward<F>(f),
        std::forward<Ts>(params)...);
}

struct Edge //new design
{
    vector<int> to;
    vector<int> weight;
    vector<int> weight2;
```

```
    Edge() {}

    Edge(int to, int weight)
    {
        add(to, weight);
    }

    void add(int to, int weight)
    {
        this->to.push_back(to);
        this->weight.push_back(weight);
        this->weight2.push_back(0); // По умолчанию добавляем 0
для weight2
    }

    void read()
    {
        int numEdges;
        cin >> numEdges;
        to.resize(numEdges);
        weight.resize(numEdges);
        weight2.resize(numEdges);
        for (int i = 0; i < numEdges; i++)
        {
            cin >> to[i] >> weight[i];
        }
    }
};

struct QueueVertex
{
```

```
    int number;
    int dist;
    QueueVertex(int number, int dist) : number(number),
dist(dist) {}
};

bool operator<(const QueueVertex& v1, const QueueVertex& v2)
{
    return v1.dist > v2.dist;
}

class Graph
{
    vector<Edge> link;

public:
    Graph(int vertex_count) : link(vertex_count) {}

    void add_edge_u(int from, int to, int weight) { //
unoriented
        link[from].add(to, weight);
        link[to].add(from, weight);
    }

    void add_edge_o(int from, int to, int weight) { // oriented
        link[from].add(to, weight);
    }

    int vertex_count() const
    {
        return link.size();
    }
```

```
    inline void turn2(int weight = 1)
    {
        const Vec8i w(weight);
        for (auto& edge : link)
        {
            std::size_t size = edge.weight2.size();
            auto i = 0;
            Vec8i d, t, r, m;
            for (; i + 8 <= size; i += 8) { //main AVX2
vectorized loop
                d.load(&edge.weight2[i]);
                m = d > w;
                r = if_sub(m, d, w);
                r = r & m;
                r.store(&edge.weight2[i]);
            }

            for (; i < size; i++) { //tail loop
                if (edge.weight2[i] > weight) {
                    edge.weight2[i] -= weight;
                }
                else {
                    edge.weight2[i] = 0;
                }
            }
        }
    }

    int dijkstra(int start, int finish)
    {
        vector<int> dist;
```

```
        vector<int> parent;
        dist.resize(vertex_count(), INF);
        dist[start] = 0;
        parent.resize(vertex_count(), -1);
        priority_queue<QueueVertex> q;
        q.push(QueueVertex(start, 0));

        while (!q.empty())
        {
            int current = q.top().number;
            int current_dist = q.top().dist;
            q.pop();

            if (current_dist > dist[current])
            {
                continue;
            }

            for (size_t i = 0; i < link[current].to.size(); i++)
            {
                int to = link[current].to[i];
                int weight = link[current].weight[i];
                int totalWeight = weight +
link[current].weight2[i];

                if (dist[current] + totalWeight < dist[to])
                {
                    dist[to] = dist[current] + totalWeight;
                    parent[to] = current;
                    q.push(QueueVertex(to, dist[to]));
                }
            }
```

```
        }
        //path(parent, start, finish);
        return dist[finish];
    }

    void path(const vector<int>& parent, int start, int finish)
    {
        deque<int> path;

        while (finish != -1)
        {
            path.push_front(finish);
            finish = (finish == start) ? -1 : parent[finish];
        }

        cout << "path" << endl;
        for (int node : path)
        {
            cout << node << ' ';
        }
        cout << endl;
    }

    void inc_wpath(const vector<int>& parent, int start, int
finish)
    {
        deque<int> path;

        while (finish != -1)
        {
            path.push_front(finish);
            finish = (finish == start) ? -1 : parent[finish];
```

```
        }

        for (auto i = 0; i < link.size(); i++)
        {
            for (auto& edge : link[i].to)
            {
                for (auto k = 0; k < path.size() - 1; k++)
                {
                    if (path[k + 1] == edge && i == path[k])
                    {
                        size_t idx = find(link[i].to.begin(),
link[i].to.end(), edge) - link[i].to.begin();
                        link[i].weight2[idx] += WEIGHT;
                    }
                }
            }
        }
    }

    void show_graph()
    {
        for (size_t i = 0; i < link.size(); i++)
        {
            for (size_t j = 0; j < link[i].to.size(); j++)
            {
                cout << i << ' ' << link[i].to[j] << ' ' <<
link[i].weight[j] + link[i].weight2[j] << endl;
            }
        }
        cout << endl;
    }
```

```
    void incRandom() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<int> distribution(0, 1);
// Генератор випадкових значень 0 або 1

        for (auto& edge : link) {
            for (auto i = 0; i < edge.weight2.size(); i++) {
                if (distribution(gen) == 1) {  // Випадковим
чином вибираємо, чи збільшувати дане значення
                    edge.weight2[i] += TURNS^2;
                }
            }
        }
    }

    struct LongestPathInfo
    {
        int start;
        int finish;
        int length;

        LongestPathInfo(int start, int finish, int length) :
start(start), finish(finish), length(length) {}
    };

    vector<LongestPathInfo> top_three_longest_paths()
    {
        vector<LongestPathInfo> all_paths;

        for (int start = 0; start < vertex_count(); ++start)
        {
```

```
            for (int finish = 0; finish < vertex_count();
++finish)
            {
                if (start != finish)
                {
                    int length = dijkstra(start, finish);
                    all_paths.emplace_back(start, finish,
length);
                }
            }
        }
        sort(all_paths.begin(), all_paths.end(), [](const
LongestPathInfo& a, const LongestPathInfo& b) {
            return a.length > b.length;
            });
        vector<LongestPathInfo> top_three;
        for (size_t i = 0; i < min(all_paths.size(),
static_cast<size_t>(3)); i++)
        {
            top_three.push_back(all_paths[i]);
        }

        return top_three;
    }
};



//OLD UNOPTIMIZED VERSION
struct MyEdge
{
    int to;
    int weight;
    int weight2 = {}; // dynamically added weight
```

```
    MyEdge() {}
    MyEdge(int to, int weight) : to(to), weight(weight) {}
    void read() {
        cin >> to >> weight;
    }
};

struct MyQueueVertex
{
    int number;
    int dist;
    MyQueueVertex(int number, int dist) : number(number),
dist(dist) {}
};

bool operator<(const MyQueueVertex& v1, const MyQueueVertex& v2)
{
    return v1.dist > v2.dist;
}

class MyGraph
{
    vector<vector<MyEdge>> link;

public:
    MyGraph(int vertex_count) :
        link(vertex_count) {}

    void add_edge_u(int from, int to, int weight) { //
unoriented
        link[from].push_back(MyEdge(to, weight));
        link[to].push_back(MyEdge(from, weight));
    }
```

```
void add_edge_o(int from, int to, int weight) { // oriented
    link[from].push_back(MyEdge(to, weight));
}

int vertex_count() const {
    return link.size();
}

void turn(int weight = 1) {
    for (auto i = 0; i < link.size(); i++)
        for (auto j = 0; j < link[i].size(); j++) {
            if (link[i][j].weight2 - weight > 0) {
                link[i][j].weight2 -= weight;
            }
            else {
                link[i][j].weight2 = 0;
            }
        }
}
int dijkstra(int start, int finish) {
    vector<int> dist;
    vector<int> parent = {};
    dist.resize(vertex_count(), INF);
    dist[start] = 0;
    parent.resize(vertex_count(), -1);
    priority_queue<MyQueueVertex> q;
    q.push(MyQueueVertex(start, 0));

    while (!q.empty()) {
        int current = q.top().number;
        int current_dist = q.top().dist;
```

```
            q.pop();

            if (current_dist > dist[current]) {
                continue;
            }

            for (auto& edge : link[current]) {
                if (dist[current] + (edge.weight + edge.weight2)
< dist[edge.to]) {
                    dist[edge.to] = dist[current] + (edge.weight
+ edge.weight2);
                    parent[edge.to] = current;
                    q.push(MyQueueVertex(edge.to,
dist[edge.to]));
                }
            }
        }
        //path(parent, start, finish);
        return dist[finish];
    }

    void path(vector<int> parent, int start, int finish) {
        deque<int> path;

        while (finish != -1) {
            path.push_front(finish);
            finish = (finish == start) ? -1 : parent[finish];
        }

        cout << "path" << endl;
        for (int node : path) cout << (node) << ' ';
        cout << endl;
    }
```

```
    void inc_wpath(const vector<int> parent, int start, int
finish) {
        deque<int> path;

        while (finish != -1) {
            path.push_front(finish);
            finish = (finish == start) ? -1 : parent[finish];
        }

        for (auto i = 0; i < link.size(); i++) {
            for (auto& edge : link[i]) {
                for (auto k = 0; k < path.size() - 1; k++) {
                    if (path[k + 1] == edge.to && i == path[k])
{
                        edge.weight2 += WEIGHT;
                    }
                }
            }
        }
    }

    void show_graph() {
        for (auto i = 0; i < link.size(); i++)
            for (auto& edge : link[i]) {
                cout << i << ' ' << edge.to << ' ' <<
edge.weight + edge.weight2 << endl;
            }
        cout << endl;
    }



    void myIncRand() {
```

```
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<int> distribution(0, 1);

        for (auto i = 0; i < link.size(); i++) {
            for (auto& edge : link[i]) {
                if (distribution(gen) == 1) {
                    edge.weight2 += TURNS^2;
                }
            }
        }
    }
};

#define TESTNUM 1000
void operateOldGraph(MyGraph& gr, int start, int finish, bool
print = true) {
    long long total1 = 0, total2 = 0, total3 = 0;
    Time time;
    for (auto k = 0; k < TESTNUM; k++) {
        auto t1 = time.add();
        int dist = gr.dijkstra(start, finish);
        auto t2 = time.add();
        auto t3 = time.add();
        for (auto i = 0; i < TURNS; i++) {  //simulate multiple
separate turns
            gr.turn();
        }
        auto t4 = time.add();
        auto t5 = time.add();
        gr.turn();
        auto t6 = time.add();
```

```
        total1 += time.count(t1, t2);
        total2 += time.count(t3, t4);
        total3 += time.count(t5, t6);
    }
    if (print) {
        cout << total1 / TESTNUM << " average nanoseconds for
dijkstra\n";
        cout << total2 / TESTNUM << " average nanoseconds for "
<< TURNS << " OLD turning\n";
        cout << total3 / TESTNUM << " average nanoseconds for
OLD one additional turn\n";
    }
}


void operateGraph(Graph gr, int start, int finish, bool print =
true) {
    long long total1 = 0, total2 = 0, total3 = 0;
    Time time;
    for (auto k = 0; k < TESTNUM; k++) {
        auto t1 = time.add();
        int dist = gr.dijkstra(start, finish);
        auto t2 = time.add();
        auto t3 = time.add();
        for (auto i = 0; i < TURNS; i++) { //simulate multiple
separate turns
            gr.turn2();
        }
        auto t4 = time.add();
        auto t5 = time.add();
        gr.turn2();
        auto t6 = time.add();
        total1 += time.count(t1, t2);
        total2 += time.count(t3, t4);
```

```
        total3 += time.count(t5, t6);
    }
    if (print) {
        cout << total1 / TESTNUM << " average nanoseconds for
dijkstra\n";
        cout << total2 / TESTNUM << " average nanoseconds for "
<< TURNS << " NEW turning\n";
        cout << total3 / TESTNUM << " average nanoseconds for
NEW one additional turn\n";
    }
}


pair<Graph, MyGraph>generateRandomGraph(int numNodes, int
numEdges) {
    Graph gr(numNodes);
    MyGraph mygr(numNodes);
    for (int i = 0; i < numEdges; ++i) {
        int from = rand() % numNodes;
        int to = rand() % numNodes;
        int weight = rand() % TURNS; // Случайный вес от 0 до
100
        gr.add_edge_o(from, to, weight);
        mygr.add_edge_o(from, to, weight);
    }
    gr.incRandom();
    mygr.myIncRand();
    pair<Graph, MyGraph> result = { gr,mygr };
    return result;
}


void testAllPairs(int numNodes, int numEdges)
{
    auto graphs = generateRandomGraph(numNodes, numEdges);
```

```
    Graph& gr = graphs.first;
    MyGraph& mygr = graphs.second;

    const int numTests = 1000;
    std::vector<long long> elapsedTimesGraphDijkstra;
    std::vector<long long> elapsedTimesGraphTurn;
    std::vector<long long> elapsedTimesMyGraphDijkstra;
    std::vector<long long> elapsedTimesMyGraphTurn;

    elapsedTimesGraphDijkstra.reserve(numTests);
    elapsedTimesGraphTurn.reserve(numTests);
    elapsedTimesMyGraphDijkstra.reserve(numTests);
    elapsedTimesMyGraphTurn.reserve(numTests);

    Time time;
    for (auto it = 0; it < 1; it++) {
        for (int start = 0; start < numNodes; ++start)
        {
            for (int finish = 0; finish < numNodes; ++finish)
            {
                if (start != finish)
                {
                    // Dijkstra for Graph
                    auto t1 = time.add();
                    int distGraph = gr.dijkstra(start, finish);
                    auto t2 = time.add();

elapsedTimesGraphDijkstra.push_back(time.count(t1, t2));

                    // Turn for Graph
                    auto t3 = time.add();
                    for (auto i = 0; i < TURNS; i++) {
```

```
                    gr.turn2();
                }
                auto t4 = time.add();

elapsedTimesGraphTurn.push_back(time.count(t3, t4));

                // Dijkstra for MyGraph
                auto t5 = time.add();
                int distMyGraph;
                auto t6 = time.add();
                distMyGraph = mygr.dijkstra(start, finish);
                auto t7 = time.add();

elapsedTimesMyGraphDijkstra.push_back(time.count(t6, t7));

                // Turn for MyGraph
                auto t8 = time.add();
                for (auto i = 0; i < TURNS; i++) {
                    mygr.turn();
                }
                auto t9 = time.add();

elapsedTimesMyGraphTurn.push_back(time.count(t8, t9));

                // Перевірка на відповідність відстаней
                if (distGraph != distMyGraph)
                {
                    std::cerr << "Error: Distances do not
match!" << std::endl;
                    return;
                }
            }
        }
```

```
        }
    }

    // Знаходимо мінімальний та максимальний час для Graph
Dijkstra
    long long minTimeGraphDijkstra =
*min_element(elapsedTimesGraphDijkstra.begin(),
elapsedTimesGraphDijkstra.end());
    long long maxTimeGraphDijkstra =
*max_element(elapsedTimesGraphDijkstra.begin(),
elapsedTimesGraphDijkstra.end());

    // Знаходимо середній час для Graph Dijkstra
    long long averageTimeGraphDijkstra =
accumulate(elapsedTimesGraphDijkstra.begin(),
elapsedTimesGraphDijkstra.end(), 0LL) / numTests;

    // Знаходимо мінімальний та максимальний час для Graph Turn
    long long minTimeGraphTurn =
*min_element(elapsedTimesGraphTurn.begin(),
elapsedTimesGraphTurn.end());
    long long maxTimeGraphTurn =
*max_element(elapsedTimesGraphTurn.begin(),
elapsedTimesGraphTurn.end());

    // Знаходимо середній час для Graph Turn
    long long averageTimeGraphTurn =
accumulate(elapsedTimesGraphTurn.begin(),
elapsedTimesGraphTurn.end(), 0LL) / numTests;

    // Знаходимо мінімальний та максимальний час для MyGraph
Dijkstra
    long long minTimeMyGraphDijkstra =
*min_element(elapsedTimesMyGraphDijkstra.begin(),
elapsedTimesMyGraphDijkstra.end());
    long long maxTimeMyGraphDijkstra =
*max_element(elapsedTimesMyGraphDijkstra.begin(),
elapsedTimesMyGraphDijkstra.end());
```

```
    // Знаходимо середній час для MyGraph Dijkstra
    long long averageTimeMyGraphDijkstra =
accumulate(elapsedTimesMyGraphDijkstra.begin(),
elapsedTimesMyGraphDijkstra.end(), 0LL) / numTests;


    // Знаходимо мінімальний та максимальний час для MyGraph
Turn
    long long minTimeMyGraphTurn =
*min_element(elapsedTimesMyGraphTurn.begin(),
elapsedTimesMyGraphTurn.end());
    long long maxTimeMyGraphTurn =
*max_element(elapsedTimesMyGraphTurn.begin(),
elapsedTimesMyGraphTurn.end());


    // Знаходимо середній час для MyGraph Turn
    long long averageTimeMyGraphTurn =
accumulate(elapsedTimesMyGraphTurn.begin(),
elapsedTimesMyGraphTurn.end(), 0LL) / numTests;


    // Виводимо результати
    //std::cout << "With " << numNodes << " nodes and " <<
numEdges << " edges\n";
    std::cout << "Graph Dijkstra Results:" << '\n';
    std::cout << "Minimum time: " << minTimeGraphDijkstra << "
nanoseconds\n";
    std::cout << "Maximum time: " << maxTimeGraphDijkstra << "
nanoseconds\n";
    std::cout << "Average time: " << averageTimeGraphDijkstra <<
" nanoseconds\n";


    std::cout << "\nGraph Turn Results:" << '\n';
    std::cout << "Minimum time: " << minTimeGraphTurn << "
nanoseconds\n";
    std::cout << "Maximum time: " << maxTimeGraphTurn << "
nanoseconds\n";
```

```
    std::cout << "Average time: " << averageTimeGraphTurn << "
nanoseconds\n";


    std::cout << "\nMyGraph Dijkstra Results:" << '\n';
    std::cout << "Minimum time: " << minTimeMyGraphDijkstra << "
nanoseconds\n";
    std::cout << "Maximum time: " << maxTimeMyGraphDijkstra << "
nanoseconds\n";
    std::cout << "Average time: " << averageTimeMyGraphDijkstra
<< " nanoseconds\n";


    std::cout << "\nMyGraph Turn Results:" << '\n';
    std::cout << "Minimum time: " << minTimeMyGraphTurn << "
nanoseconds\n";
    std::cout << "Maximum time: " << maxTimeMyGraphTurn << "
nanoseconds\n";
    std::cout << "Average time: " << averageTimeMyGraphTurn << "
nanoseconds\n";
}


void testSomePairs(int numNodes, int numEdges) {
    auto graphs = generateRandomGraph(numNodes, numEdges);
    auto gr = graphs.first;
    auto mygr = graphs.second;
    vector<int> starts;
    vector<int> finishes;


    for (int i = 0; i < TESTNUM; ++i) {
        starts.emplace_back(rand() % numNodes);
        finishes.emplace_back(rand() % numNodes);
    }


    cout << "Average times for Graph:\n";
    Time time;
```

```
    // Measure the time for operating on Graph
    auto t1 = time.add();
    for (auto i = 0; i < 1; i++) {
        operateGraph(gr, starts[i], finishes[i], false);
    }
    auto t2 = time.add();

    // Measure the time for the turn operation
    for (auto i = 0; i < TURNS; i++) {
        gr.turn2();
    }
    auto t3 = time.add();

    // Calculate and display the average times for Graph
    auto totalOperateTimeGraph = time.count(t1, t2);
    auto averageOperateTimeGraph = totalOperateTimeGraph /
TESTNUM;

    auto totalTurnTimeGraph = time.count(t2, t3);
    auto averageTurnTimeGraph = totalTurnTimeGraph / TURNS;

    cout << "Average time for operateGraph: " <<
averageOperateTimeGraph << " nanoseconds\n";
    cout << "Average time for gr.turn(): " <<
averageTurnTimeGraph << " nanoseconds\n";

    cout << "\nAverage times for MyGraph:\n";

    // Measure the time for operating on MyGraph
    auto t5 = time.add();
    for (auto i = 0; i < 1; i++) {
        operateOldGraph(mygr, starts[i], finishes[i], false);
```

```
    }
    auto t6 = time.add();

    // Measure the time for the turn operation in MyGraph
    for (auto i = 0; i < TURNS; i++) {
        mygr.turn();
    }
    auto t7 = time.add();

    // Calculate and display the average times for MyGraph
    auto totalOperateTimeMyGraph = time.count(t5, t6);
    auto averageOperateTimeMyGraph = totalOperateTimeMyGraph /
TESTNUM;

    auto totalTurnTimeMyGraph = time.count(t6, t7);
    auto averageTurnTimeMyGraph = totalTurnTimeMyGraph / TURNS;

    cout << "Average time for operateOldGraph: " <<
averageOperateTimeMyGraph << " nanoseconds\n";
    cout << "Average time for mygr.turn(): " <<
averageTurnTimeMyGraph << " nanoseconds\n";

}



int main()
{
    Time time;
    auto graphs = generateRandomGraph(3000, 15000);
    auto mygr = graphs.second;
    operateOldGraph(mygr, 10, 94);
    cout << '\n';
```

```
    cout << '\n';
    auto gr = graphs.first;
    operateGraph(gr, 10, 94);
    cout << '\n';
    //testAllPairs(100, 1500);

    vector<pair<int, int>> testCases = {
        {100, 500},
        {100, 1500},
        {1000, 5000},
        {1000, 15000},
        {10000, 50000},
        {10000, 150000},
    };

    for (auto i = 1; i <= 10; i+=5) {
        for (auto j = 1; j <= 10; j+=5) {
            //testCases.emplace_back(i * 100, j * 500);
            //testCases.emplace_back(i * 100, j * 1500);
        }
    }

    constexpr const auto testAll = false;
    for (const auto& testCase : testCases) {
        int numNodes = testCase.first;
        int numEdges = testCase.second;

        cout << "Testing for " << numNodes << " nodes and " <<
numEdges << " edges:\n";
        if constexpr (testAll) {
            testAllPairs(numNodes, numEdges);
        }
```

```
        else {
            testSomePairs(numNodes, numEdges);
        }
        auto t2 = time.add();
        cout << "------------------------\n";
    }
    return 0;
}
```