

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

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))

Дипломна робота

на здобуття ступеня вищої освіти «бакалавр»

(освітньо-кваліфікаційний рівень)

на тему

Методи пошуку оптимальної стратегії поведінки у системі дорожнього
трафіку

Methods of finding the optimal strategy of behavior in the road traffic system

Виконав: студент денної форми навчання

спеціальності 123 – Комп'ютерна інженерія.

(шифр і назва напрямку 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)

Рекомендовано до захисту:

Протокол засідання кафедри

№ від « » 2022 р.

Завідувач кафедри

Євгеній МАЛАХОВ

(підпис)

(ім'я, прізвище)

Захищено на засіданні ЕК №

протокол № від « » 2022

р.

Оцінка /

(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

Надія КАЗАКОВА

(підпис)

(ім'я, прізвище)

Одеса – 2022

АНОТАЦІЯ

Пошук шляху залишається актуальною темою досліджень, тісно пов'язаною з ефективним розподіленням ресурсів агентами. Метою пошуку шляху як правило є пошук найкоротшого шляху від одної точки графу до іншої. Чимало прикладних задач з реального світу можуть розглядатися як проблеми пошуку шляху, такі як планування руху, логістика та прийняття рішень.

Для рішення пов'язаних з пошуком шляху проблем запропоновано чимало алгоритмів. Рішення задачі планування пошуку шляху зазвичай є відносно простим у разі наявності лише одного агента, але значно ускладнюється можливістю конфліктів у разі наявності багатьох агентів одночасно. Крім того, швидкість рішення задач відчутно залежить від масштабів та наявних обчислювальних ресурсів.

Метою дипломної роботи є реалізація на основі існуючих методів рішення задач планування маршруту методу раціонального планування маршруту для декількох агентів, що уникають взаємних конфліктів. У дипломній роботі проведено аналіз існуючих алгоритмів рішень задачі, аналіз та оцінка існуючих інструментів для їх втілення, реалізовано деякі з алгоритмів. В результаті дипломної роботи запропоновано та реалізовано перехідний алгоритм пошуку шляху для декількох агентів, що уникають взаємних конфліктів.

Ключові слова: пошук шляху, алгоритміка, теорія графів

ABSTRACT

Pathfinding remains to be an interesting research topic closely related to the efficient allocation of agents' resources. The goal of pathfinding is usually to find the shortest path from one graph point to another. Many real-world applications can be viewed as pathfinding problems, such as motion planning, logistics, and decision making.

Many algorithms have been proposed to solve pathfinding-related problems. Solving a pathfinding planning problem is usually relatively straightforward with one agent, but is greatly complicated by the possibility of conflicts when many agents are present at the same time. Moreover, the speed of solving problems depends significantly on the scale and available processor resources.

The aim of this thesis is to implement, based on existing methods of solving route planning problems, a method for rational route planning for several agents, who trying to avoid mutual conflicts. The thesis analyzes existing algorithms for solving the problem, analyzes and evaluates existing tools for their implementation, and implements some of the algorithms. As a result of the thesis work, a transitive pathfinding algorithm for multiple agents, who trying to avoid mutual conflicts, is designed and implemented.

Keywords: pathfinding, algorithms, graph theory

ЗМІСТ

ВСТУП.....	5
1 ІСНУЮЧІ АЛГОРИТМИ ПОШУКУ ШЛЯХУ	7
2 БІБЛІОТЕКА BOOST GRAPH LIBRARY	12
3 АЛГОРИТМИ ФЛОЙДА-УОРШЕЛА ТА ДЕЙКСТРИ З ЗАСТОСУВАННЯМ БІБЛІОТЕКИ BOOST	16
4 НАПІВКООПЕРАТИВНИЙ АЛГОРИТМ ДЕЙКСТРИ	24
5 ДИНАМІЧНІ ВАГИ	28
6 ОЦІНКА ЕФЕКТИВНОСТІ МОДІФІКОВАНОГО АЛГОРИТМУ	33
ВИСНОВОК.....	35
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	36
ДОДАТОК А. Код програми за допомогою бібліотеки Boost.....	37
ДОДАТОК Б. Код напівкооперативного алгоритму Дейкстри.....	41

ВСТУП

Пошук шляху – це математична та комп’ютерна задача, що складається з пошуку оптимального шляху у графі. Вона зустрічається у багатьох сферах, таких як комп’ютерні ігри, роботехніка, логістика, навігація тощо. Пошук оптимального шляху залишається актуальною темою досліджень, тісно пов’язаною с ефективним розподіленням ресурсів агентами. Існує багато алгоритмів пошуку шляху відповідно специфіці конкретних задач.

Пошук шляху у системі дорожнього трафіку є окремим випадком більш загальної задачі пошуку оптимального шляху. Шлях може плануватися не тільки у фізичному просторі. У вигляді графу чи матриці, задачі на яких будуть вирішуватися алгоритмом пошуку шляху, можуть бути представлені і інші класи задач, наприклад карта дій (рішень, подій) тощо. Таким чином, одні й ті самі алгоритми рішень оптимального шляху можуть бути застосовані не лише для вирішення задач пошуку шляху у системі дорожнього руху та руху взагалі, але й у більш широкій області. Проте задля простоти термінології та демонстрації у подальшому в роботі будуть використовуватися терміни, що описують пошук оптимального шляху у фізичному просторі.

Двома головними стратегіями є інформований та неінформований пошук шляху. Відмінність інформованого пошуку шляху від неінформованого полягає у знанні щодо необхідного кінцевого стану, що забезпечується деякою евристичною функцією. Обидві стратегії мають переваги та недоліки, хоча інформований пошук помітно популярніше за неінформований.

Конкретна реалізація однієї зі стратегій може різнитися залежно від кількості та пропорції наявних ресурсів у вигляді обчислювальних потужностей, часу та пам’яті. Наприклад, багато пристроїв мають обмежені потужності, що схиляє їх до економних реалізацій алгоритмів пошуку шляху. Ще більш це ускладнюється необхідністю знаходити оптимальний шлях автономно. І навпаки, у випадку рішення задачі найефективнішого

переміщення крупної техніки, що оснащена телекомунікаційними технологіями має сенс обчислення настільки ефективного шляху, наскільки це можливо. Навіть якщо у відносному вимірі щодо економних засобів обчислення це буде здаватися занадто витратним, у цілому буде зекономлено у багато разів більше ресурсів – ймовірно й час теж.

Іншим викликом, що існує у цій сфері, є проблема масштабування алгоритмів. Найпопулярніший на сьогоднішній день алгоритм пошуку A^* дає швидкі та економні результати, але не найефективніші, а також погано масштабується. Багато сучасних дослідників пропонують або суттєві модифікації алгоритму, або його альтернативи. Існуючі рішення включають в себе обрізку графів, препроцесінг та модифікації евристичної функції. Всі вони так чи інакше пропонують компроміс між швидкістю та оптимальністю кінцевого результату. Особливою проблемою масштабування є кооперативність – тобто урахування більше ніж одного агента. Це фактично додає ще один вимір проблеми, що розглядається.

Найвідомішими алгоритмами пошуку шляху є алгоритм Флойда-Уоршела, алгоритм Дейкстри, A^* та його модифікації, направлені на виправлення існуючих недоліків та більшу ефективність алгоритму для деяких типів задач. Реалізації найвідоміших алгоритмів пошуку шляху можна писати з нуля, проте вони в тому чи іншому вигляді пропонуються різними бібліотеками, такими як Boost Graph Library (BGL) і Simple and Fast Multimedia Library (SFML). На жаль, просунуті модифікації або інші варіанти алгоритмів пошуку шляху не підтримуються жодними з широко відомих бібліотек.

Актуальності задачі надає у тому числі поширеність пов'язаних з нею задач у повсякденному житті. Це веде до пропонування рішень для кінцевих користувачів від багатьох компаній. Наприклад, для планування маршруту по населеним пунктам існують GPS-навігатори та такі додатки, як Google Maps, 2GIS та інші. Однак не можна вважати, що рішення, які вони пропонують,

цілком вичерпують питання. По-перше, публічно не відомі подробиці щодо того, як саме та на основі яких алгоритмів вони функціонують. По-друге, відомо, що при будівництві маршруту користувача вони мають за мету врахувати лише його інтереси, а не інтереси всіх своїх користувачів чи інтереси всіх учасників дорожнього руху. На противагу цьому існують кооперативні алгоритми пошуку шляху, що мають за мету уникання конфліктів між декількома агентами руху та самі по собі варті становлять окремий об'єкт дослідження.

Можна зробити висновок, що актуальність та можливості дослідження даної теми є достатніми для кваліфікаційної роботи. Метою дипломної роботи є проектування та реалізація методу раціонального планування маршрутів для декількох агентів, що уникають взаємних конфліктів. Для досягнення цієї мети було заплановано вирішити наступні задачі: аналіз предметної області, огляд існуючих алгоритмів, проектування перспективної модифікації алгоритмів та оцінку їх ефективності.

1 ІСНУЮЧІ АЛГОРИТМИ ПОШУКУ ШЛЯХУ

Існує багато алгоритмів рішення задач пошуку оптимального шляху. В рамках даного розділу буде розглянуто основні з них, такі як:

1. Алгоритм Флойда-Уоршела.
2. Алгоритм Дейкстри.
3. Алгоритм A*
4. Алгоритм Cooperative A* (CA*).

Алгоритм Флойда-Уоршела - це алгоритм пошуку найкоротших шляхів у графі з позитивною чи негативною вагою ребер (але без негативних циклів) у орієнтованому графі [1]. Алгоритм знаходить найкоротші відстані між всіма парами вершин графу, вирішуючи задачу пошуку для кожної упорядкованої пари вершини v, w найменшого з усіх знайдених шляхів з вершини v до вершини w . За замовченням він повертає відстань, а не шлях, але за необхідності шлях неважко реконструювати за допомогою відносно простих модифікацій алгоритму. Приклад задачі та її рішення за допомогою алгоритма Флойда-Уоршелла наведено на рис. 1.1 та таблиці 1.1.

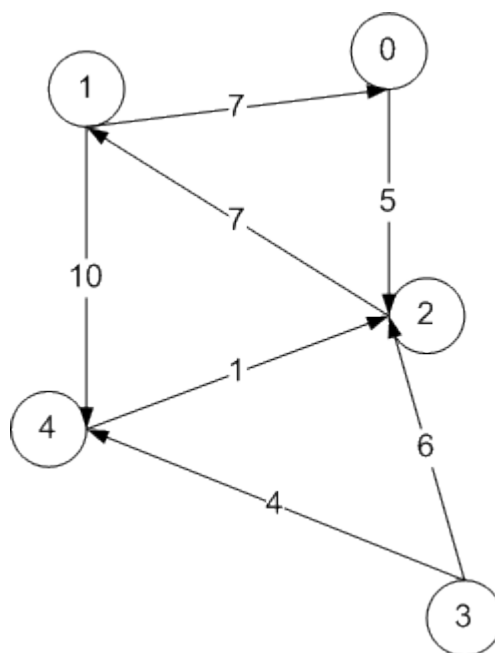


Рисунок 1.1 – Приклад направленного графу

Таблиця 1.1 – Таблиця відстаней для рис. 1.1, отриманих за допомогою алгоритма Флойда-Уоршелла

Номера вершин					
	0	1	2	3	4
0	0	12	5	-	22
1	7	0	11	-	10
2	14	7	0	-	17
3	19	12	5	0	4
4	15	8	1	-	0

Слід зазначити, що алгоритм Флойда-Уоршела не намагається перебрати всі шляхи у графі, які в принципі є можливими. Алгоритм Флойда-Уоршела є прикладом динамічного програмування. Динамічне програмування – це підхід до вирішення складних задач шляхом їх розшарування на декілька взаємопов’язаних підзадач, сукупність яких дає рішення початкової задачі шляхом менших ресурсних задач, ніж при спробі вирішення початкової задачі повністю (без розбиття на підзадачі). Це дозволяє зменшити складність алгоритму до $O(n^3)$ [2].

Алгоритм Дейкстри

Алгоритм Дейкстри — алгоритм на графах, відкритий Едсгером Дейкстрою. Знаходить найкоротший шлях від однієї вершини графа до всіх інших вершин. Класичний алгоритм Дейкстри працює тільки для графів без циклів від’ємної довжини. Складність алгоритму складає $O(n^2)$. На рис. 1.2 наведено приклад використання алгоритму Дейкстри на сітці – двомірному просторі. Синіми клітинами відзначені вершини, які алгоритм обчислив спочатку, блакитними – в кінці, білими – вершини, що не відвідувалися.

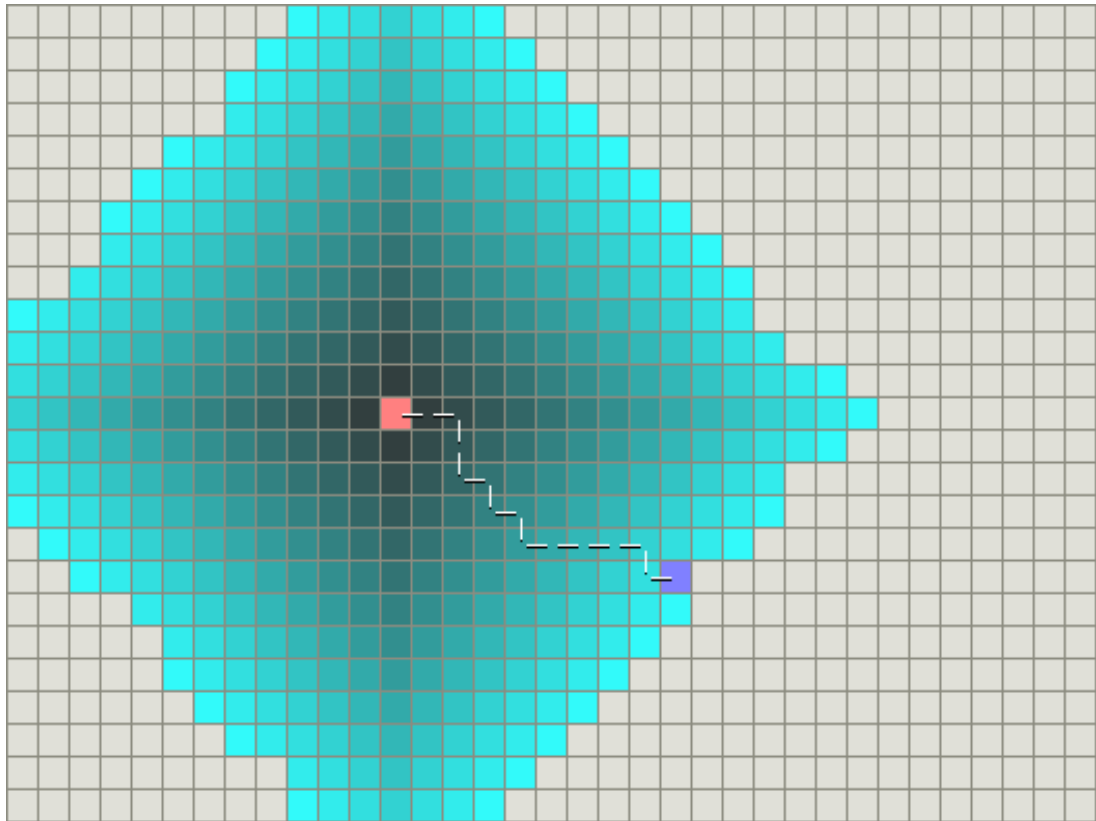


Рисунок 1.2 – Алгоритм Дейкстри на сітці

Алгоритм A^*

На відміну від алгоритма Дейкстри, алгоритм A^* враховує не тільки те, як далеко розташована поточна вершина від початкової, але й те, наскільки близько вона розташована до мети. Однак ця вартість є точно невідомою, адже для того, щоб її точно знати, необхідно знати найкоротші шляхи між усіма вершинами та метою. Тому дається не точне значення, а приблизна оцінка. Ця оцінка називається евристичною функцією. Правильне значення евристичної функції здатно суттєво вплинути на ефективність алгоритму A^* [3].

Наявність евристичної функції робить A^* алгоритмом інформованого пошуку, у той час як Дейкстра тощо є неінформованим, бо оцінка відстані до мети в ньому відсутня. Перевага від цього додаткового знання залежно від значення евристичної функції може як прискорити пошук ціною зменшення його оптимальності, так і дати близькі або ідентичні з Дейкстрою

результати. Хоча у більшості випадків використання евристичної функції дає перевагу алгоритму A^* над алгоритмом Дейкстри, у разі неможливості задання евристичної функції використання алгоритму Дейкстри є більш доцільним [4].

Алгоритм SA^*

A^* пошук наявність інших агентів або, можливо, розглядає їх як нерухомі перешкоди. Якщо зіткнення неминуче, то одиниці, що беруть участь у ньому, виконують повторний пошук і вибирають новий шлях. Чим більше щільність агентів у середі, тим більше буде подібних конфліктів і тим менш ефективним буде A^* . Для вирішення цієї проблеми Девід Сілвер запропонував істотну модифікацію A^* , яку він назвав Cooperative A^* [5].

Кооперативний пошук шляху дає агентам можливість знати наміри один одного. A^* пошук все ще використовується, але таким чином, що враховує інші рухи інших агентів. Це зробить пошук набагато повільнішим, хоча Сілвер наполягає на тому, що шляхом деяких трюків та модифікацій початкового алгоритму можна суттєво збільшити його швидкість без втрат у оптимальності.

Щоб вирішити проблему спільного пошуку шляху, алгоритм пошуку повинен мати повне знання як про перешкоди, так і про агентів. Однак, коли агенти переміщуються, не існує задовільного способу уявити їх маршрути на нерухомій карті. Щоб вирішити цю проблему, Девід Сілвер пропонує розширити карту, додаючи до неї третій вимір: час.

Після того, як агент вибрав шлях, йому необхідно переконатись, що інші блоки знають, що треба уникати клітин на його шляху. Це досягається шляхом позначення кожної клітини у таблиці резервування. Це структура даних, що містить запис для кожної комірки карти простору-часу.

Кожен запис визначає, чи є доступною або заблокованою відповідна вершина у конкретний момент часу. Як тільки вершина зарезервована, будь-

який інший блок не має права переміщатися в неї. Резервування діє як перешкода, блокуючи місце розташування на один тимчасовий крок у майбутньому.

Використовуючи таблицю резервування та карту простору-часу, Cooperative A* дозволяє вирішити проблему кооперативного пошуку шляхів (рис. 3.1).

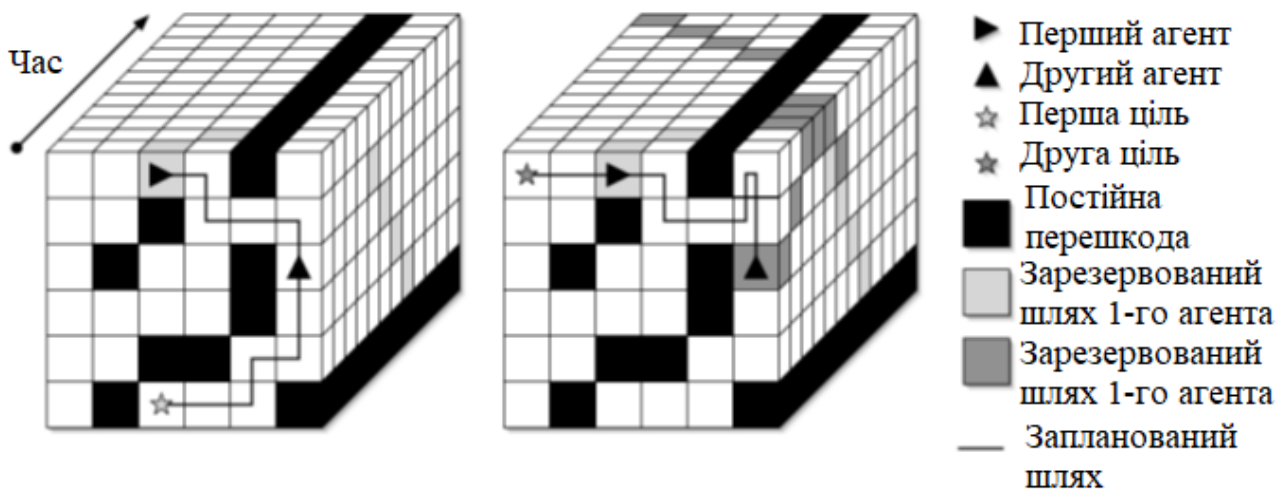


Рисунок 1.3 - Два агенти спільно шукають шлях.

Ліворуч перший агент шукає шлях та зазначає його у таблиці резервування. Праворуч другий пристрій шукає шлях, беручи до уваги існуючі резервування, а також зазначає власні переміщення у таблиці резервування.

Прикладом реалізації кооперативного A* є робота GitHub користувача Co3us з відкритим кодом, створена на двигуні Unity [6]. Порівняння A* та Cooperative A* наведено на рис. 1.4. Позначений синім кольором агент повинен досягнути позначеного зеленим кола знизу. Він не враховує шлях переміщення білого прямокутника, що веде до зіткнення. На відміну від нього на ілюстрації праворуч агент враховує напрямок руху білого прямокутника,

що дозволяє йому обійти його прямо під час переміщення білого прямокутника, уникаючи зіткнення з ним

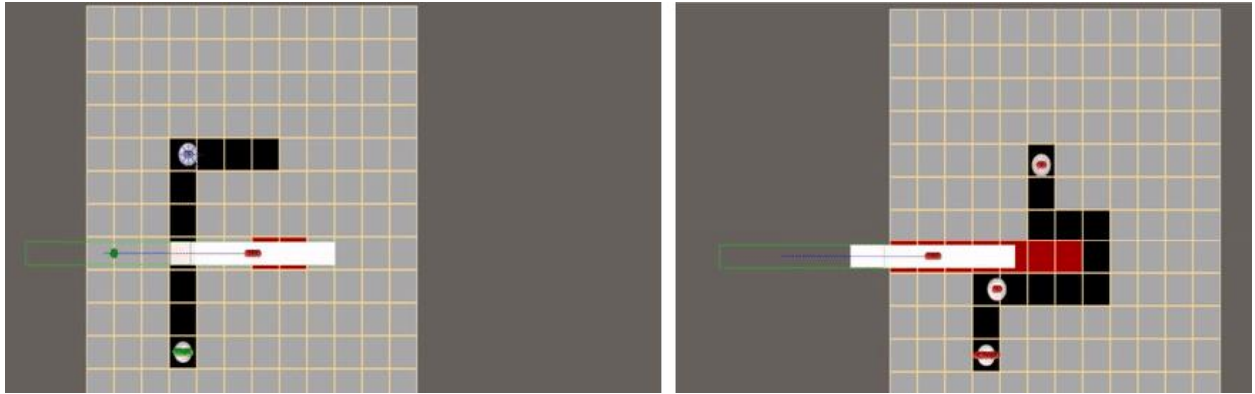


Рисунок 1.4 – Ліворуч звичайний алгоритм пошуку A*. Праворуч кооперативн агент звичайний алгоритм пошуку A*.

2 БІБЛІОТЕКА BOOST GRAPH LIBRARY

Графи є одними з найкращих видів структур даних. Реалізація з нуля пов'язаних з графами алгоритмів пов'язана з чималою кількістю труднощів та рутини. У зв'язку з цим була проведена спроба застосувати в ході роботи спеціальні бібліотеки, що призначені для роботи з графами. Аналіз показав, що існує зовсім небагато бібліотек, що пропонують такі можливості.

Бібліотеки Boost C++ Libraries - це колекція сучасних бібліотек з відкритим вихідним кодом на основі стандарту C++. Початковий код випущений під ліцензією Boost Software License, яка дозволяє будь-кому використовувати, змінювати та розповсюджувати бібліотеки безкоштовно. Бібліотеки не залежать від платформи і підтримуються абсолютною більшістю популярних компіляторів, а також менш відомими. Популярні дистрибутиви Linux та UNIX, такі як Fedora, Debian та NetBSD містять готові пакети Boost. Boost також може бути завантажено з офіційного веб-сайту бібліотеки.

Співтовариство Boost виникло приблизно 1998 року, коли було випущено першу версію стандарту. З того часу воно постійно зростало і зараз грає велику роль у стандартизації C++. Незважаючи на те, що між спільнотою Boost та комітетом зі стандартизації немає формальних відносин, деякі розробники беруть активну участь в обох групах. Поточна версія стандарту C++, яка була затверджена в 2011 році, включає бібліотеки, коріння яких сягає спільноти Boost.

Спільнота Boost відповідає за розробку та публікацію бібліотек Boost. Спільнота складається з відносно великої групи розробників C++ з усього світу, які координуються через веб-сайт www.boost.org. У якості репозиторія коду використовується GitHub. Місія спільноти полягає у розробці та збиранні високоякісних бібліотек, що доповнюють стандартну бібліотеку. Бібліотеки, які доводять свою цінність і стають важливими для розробки додатків на C++, можуть бути включеними до стандартної бібліотеки C++.

Деякі бібліотеки Boost пропонують досить низькорівневі розширення стандартних бібліотек. З роками частина з цих елементів потрапила до стандартної бібліотеки C++, проте більша їх частина продовжує існувати як бібліотека до C++, а не безпосередньо у вигляді частини стандарту.

Як неважко зрозуміти з того факту, що деякі елементи Boost були включені до стандарту C++, Boost є однією з найпопулярніших бібліотек, що використовуються для C++. Бібліотеки Boost позиціонуються як хороший вибір для підвищення продуктивності проєктів на C++ [7], коли вимоги проєкту виходять за рамки інструментів, що пропонуються стандартною бібліотекою мови C++.

Таким чином, ключовими перевагами Boost є:

- 1) відкритий код та рецензуємість;
- 2) широкий спектр функціональних можливостей;
- 3) крос-платформеність;
- 4) наявність детальної документації;
- 5) вільне ліцензування;
- 6) продуктивність;

Шаблонна бібліотека Boost Graph Library з'явилась у складі Boost 1.18.1 у 2000 році. Було написано мануал до бібліотеки, складений Джеремі Сіком, Лай-Кваном Лі та Ендрю Ламсдейном. Бібліотека інтенсивно оновлювалася та розвивалася практично до кінця 2013 року (Boost 1.55.0). Зокрема, у 2005 році з'явився анонс її розподіленої версії (PBGL), яка увійшла до складу Boost з версії 1.40 у 2009 році та по поточний час є найпоширенішим рішенням для графових обчислень на високопродуктивних кластерах.

Таким чином, бібліотеці в цілому присвячена хороша і різноманітна література, але власна її документація, також, взагалі кажучи, досить велика, дещо страждає на ряд проблем. Наприклад, її зміст та кореневі розділи, які

претендують на те, щоб давати вичерпний перелік ключових сутностей, не змінювалися з самого 2001 року.

Необхідно відзначити, що в даний час основні розробники, як видно, втратили інтерес до подальшої роботи над бібліотекою. Останні десять років вона знаходиться в підвішеному стані, аж ніяк не вичерпавши потенціал свого розвитку, але не звільнившись цілком від внутрішніх неузгодженостей і прямих помилок. Озвучені у 2011 році плани щодо суттєвого розширення набору методів та покриття нових областей теорії графів (у тому числі за рахунок додавання внутрішньої підтримки `graph partitioning` до можливості читати формат METIS) залишилися нереалізованими. Звісно ж, що бібліотека могла б багато виграти (як мінімум, щодо читаності) від широкого використання новинок, які увійшли до стандарту після 2011 року.

Тим не менш, на даний момент тільки ця бібліотека принципово дозволяє без втрати продуктивності, нав'язування жорстких угод за поданням даних та втрати контролю над внутрішніми механізмами самої бібліотеки, повністю розвести графові алгоритми та графові уявлення, додатково зробивши останні цілком незалежними від уявлення метаданих, асоційованих з ребрами і вершинами (що у принципі є, очевидно, найбільш правильним способом ведення справ).

Boost Graph Library декларує повну реалізацію пошуку в глибину, пошуку завширшки та методу рівних цін з реалізацією на їх основі таких алгоритмів, як алгоритм Дейкстри та алгоритм Флойда-Уоршелла [8]. Однак у ході детального ознайомлення з бібліотекою було з'ясовано, що насправді звернення до функцій класів, які реалізують ці алгоритми, необхідна істотна робота з боку програміста. По-перше, для звернення до функцій класів необхідно створити та ретельно перевірити структури даних на відповідність тим конструкторам класів, які реалізовані бібліотекою. По-друге, це може бути проведено лише на основі типів та структур даних, пропонованих бібліотекою, що пов'язано зі специфікою реалізації інтерфейсу для взаємодії з такими

структурами даних. По-третє, навіть після реалізації сумісних з бібліотекою структур даних на основі запропонованих бібліотекою типів даних безпосередньо звернення до функцій, що реалізують алгоритми пошуку, неминуче виходить дуже громіздким як з точки зору обсягу, так і з точки зору логіки коду, що ускладнює його налагодження.

Цим недолікам досить легко знайти пояснення. Орієнтація на крос-платформність часто веде до меншої гнучкості, мабуть, Boost не є винятком. Крім того, як було зазначено вище, деякі частини документації не оновлювалися близько десятка або навіть більше років. Очевидно, це позначилося як на дружелюбності деяких фрагментів документації, і на увазі зручності використання запропонованих бібліотекою структур даних у сучасних версіях мови C++. Нарешті – і це, швидше за все, головна причина – високий ступінь абстракції пропонованих бібліотек Boost вже сама по собі диктує необхідність значного обсягу самотійної роботи для можливості повноцінної взаємодії з нею.

Слід додатково зазначити, що кожна з перелічених вище причин, напевно, додатково посилена тим фактом, що розробка та підтримка бібліотеки Boost здійснюється спільнотою на некомерційних засадах. Безумовно, багато компаній зацікавлені у розвитку засобів розробки. Однак є підстави припускати, що в той час, як удосконалення стандарту C++ фінансово підтримується низкою комерційних компаній, бібліотеки такої уваги не отримують. Безумовно, абсурдно було б припускати підтримку безлічі бібліотек. Однак бібліотеки C++ фінансово підтримується низкою комерційних компаній, бібліотеки такої уваги не отримують. Безумовно, абсурдно було б припускати підтримку безлічі бібліотек. Однак бібліотеки C++ Проте бібліотеки C++ Standard Library і Boost загальновизнано вважаються невід'ємними супутниками самої мови, що дає підстави розглядати відсутність помітного фінансування як помилку з боку провідних компаній, які активно користуються можливостями C++ та бібліотек до неї.

Для ілюстрації розглянемо приклад алгоритму Флойда Уоршелла. Запропонований бібліотекою заголовний файл `bellman_ford_shortest_paths.hpp` містить чотири функції, озаглавлені як `bool bellman_ford_shortest_paths`, з чотирма різними переліками аргументів, необхідних функціями, чим неважко збентежити будь-якого нового користувача бібліотеки. На щастя, документація бібліотеки чітко специфікує потрібні при виклику функції аргументи та формат вихідних даних, проте жодних інших деталей – у вигляді коментарів усередині самого заголовного файлу, наприклад, щоб різниця між чотирма `bool bellman_ford_shortest_paths` була більш очевидною – бібліотека не представляє, як і додаткових. рекомендацій щодо їх використання.

3 АЛГОРИТМИ ФЛОЙДА-УОРШЕЛА ТА ДЕЙКСТРИ З ЗАСТОСУВАННЯМ БІБЛІОТЕКИ BOOST

В рамках роботи була проведена бібліотека Boost на Windows була проведена за допомогою завантаження пакету з офіційного сайту та вказівки шляху до набору бібліотек всередині інтерфейсу IDE Visual Studio 2019.

Насамперед необхідно підключити бібліотеку Boost. У рамках проекту бібліотеку Boost було підключено до IDE Visual Studio 2019 (рис 3.1). Повний код програми наведено у додатку А.

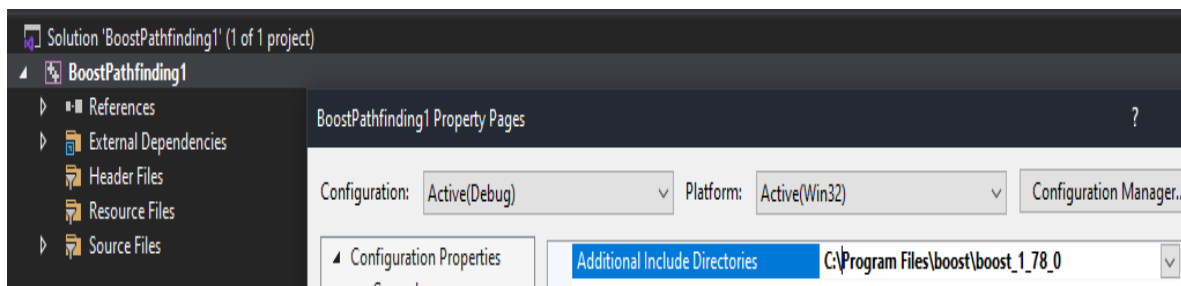


Рисунок 3.1 – Підключення бібліотеки Boost до Visual Studio 2019

Підключення заголовних файлів проводиться наступним чином (ліс. 3.1).

```
#include <boost/graph/floyd_warshall_shortest.hpp>
#include "boost/graph/random.hpp"
#include <stdlib.h>
#include <iostream>
#include <iterator>
#include <chrono>
```

Лістинг 3.1 – Підключення Boost

Бібліотеки, що починаються з boost, є елементами Boost Graph Library. Слід зазначити, що компілятор вимагає підключення boost/graph/random. Це пов'язано з особливостями роботи інших частин бібліотеки.

Stdlib.h і iosstream є поширеними у всіх C++ проектах бібліотеками, і це не є винятком. В рамках даної реалізації stdlib.h необхідний, оскільки додає змінну size_t, що повертається ключовим словом sizeof, необхідну для роботи з векторами. Iostream у свою чергу необхідний виведення результатів роботи програми в консоль. Iterator також потрібен для зручної роботи з векторами. Chrono використано для фіксації різниці між різними моментами часу всередині програми. Set необхідний для створення одного із шаблонних класів.

Після цього слід оголошення коротких звернень до змінних і структур даних, які будуть використані в проекті. Більшість їх належить до Boost. Визначено та оголошено кастомний тип даних DirectedGraph, а також необхідні для роботи з ним списки суміжності, ітератори для роботи з ними, а також необхідні властивості бібліотеки графа. Крім цього, один з typedef відноситься до chrono (він буде використаний в класі Time, який буде розглянутий далі), а також оголошено одну пару змінних типу int і вектор таких пар, а також вектор таких пар. Це необхідне роботи алгоритму Дейкстри.

DirectedGraph являється направленим графом с весами. Для его наполнения созданы функции insertEdge и fillGraph (ліс. 3.2).

```
void insertEdge(unsigned long s, unsigned long d, int w,
                DirectedGraph& a) {
    add_edge(s, d, w, a);
}
void fillGraph(int n, int g1, int g2, int w1, int w2,
               DirectedGraph& a, VVPII& d) {
//add n edges in graphs between g1 and g2 with weights
//between w1 and w2
```

Лістинг 3.2 – Функції InsertEdge та fillGraph

Функція insertEdge реалізує можливість роботи з матрицями суміжності, що надається бібліотекою. Вона приймає чотири аргументи – вихідну вершину, кінцеву вершину, вагу ребра та граф, до якого додається нове ребро.

Функція fillGraph дозволяє автоматично створювати графи довільних розмірів на основі заданих параметрів. FillGraph приймає такі аргументи: n –

кількість вершин, які потрібно додати; $g1, g2$ – діапазон значень вершин, між якими будуть додані нові ребра; $w1, w2$ – діапазон значень ваг у новостворених ребрах; все це додається до переданих функцій графів.

Алгоритм Флойда-Уоршелла за допомогою бібліотеки Boost Graph Library реалізується в такий спосіб (лістинг 3.3).

```
#define INF SHRT_MAX
vector<vector<int>> floydWarshall(DirectedGraph g) {
    vector<vector<int>> distances;
    <...>
    return distances; }
```

Лістинг 3.3 – алгоритм Флойда-Уоршела

Створений у рамках Boost Graph Library алгоритм Флойда-Уоршела викликається у наступному рядку (ліс. 3.4).

```
distances[boost::source(*it, g)]
[boost::target(*it, g)] =
boost::get(boost::edge_weight_t(), g, *it);
```

Лістинг 3.4 – виклик алгоритму Флойда-Уоршела

Функція повертає вектор векторів, що містять змінні типу `int` з обчисленими найкоротшими відстанями між усіма парами вершин. На прикладі функції `floydWarshall()` можна побачити, що незважаючи на декларовану Boost Graph Library підтримку алгоритму Флойда-Уоршелла, взаємодія з бібліотекою є нетривіальною і вимагає написання досить складної логіки.

Використання стандартного пропонованого Boost Graph Library алгоритму Дейкстри неможливе, оскільки передбачає використання `color map`. Це веде до двох проблем: по-перше, `color map` неможливо відобразити у консольному додатку. По-друге, багато IDE вкрай неохоче відображають `color`

map у принципі, і Visual Studio 2019 не стала винятком. У зв'язку з цим алгоритм Дейкстри взято from scratch.

Зазначимо, що він вирішує відмінну від алгоритму Флойда-Уоршелла задачу. Алгоритм Флойда-Уоршелла вирішує завдання пошуку найкоротших відстаней між будь-якими двома вершинами. Алгоритм Дейкстри вирішує завдання пошуку найкоротшої відстані між однією вершиною та рештою. У рамках проекту граф, на якому надалі розраховуватиметься алгоритм Дейкстри, належить тому ж типу, що й DirectedGraph для Флойда-Уоршелла, а саме – спрямований граф зі зваженими ребрами.

Для вимірювання часу, витраченого на розрахунки пошуку шляху у реалізованих у межах проекту алгоритмах створено клас Time (ліс. 3.5).

```
typedef std::chrono::steady_clock::time_point tp;
class Time {
public:
    static void show(tp t1, tp t2){ //time passed since t1
        cout << chrono::duration_cast<chrono::milliseconds>(t2 -
t1).count() << " milliseconds" << endl;
    }
    tp addPoint() {
        tp p = chrono::steady_clock::now();
        return p;
    }
};
```

Лістинг 3.5 – клас Time

Клас Time реалізує можливості, що надаються стандартною бібліотекою, за точним підрахунком часу в зручних одиницях часу. Ними можуть бути як секунди, так і мілісекунди, мікросекунди та наносекунди. Функція addPoint() фіксує поточний час з точністю до зазначених одиниць змінної типу timepoint, для зручності скороченої до tp за допомогою typedef. Вищезазначені функції використовуються в main() у наступному вигляді (ліс. 3.6).

```

int main()
{
    VVPII dgraph = {};
    Time time;
    auto t1 = time.addPoint();
    int nodes1 = 0;
    int nodes2 = 36;
    fillGraph(180, nodes1, nodes2, 2, 15, g, dgraph);
    auto t2 = time.addPoint();
    time.show(t1, t2);
    vector<vector<int>> FloydWarhsall = floydWarshall(g); //Here
    we get the minimum distances from all nodes to other ones using
    Floyd-Warshall's algorithm
    auto t3 = chrono::steady_clock::now();
    time.show(t2, t3);
    for (auto i = 0; i < FloydWarhsall.size(); i++) {
        for (auto j = 0; j < FloydWarhsall.size(); j++) {
            cout << FloydWarhsall[i][j]<< "\t";
        }
        cout << endl;
    }
    auto t4 = chrono::steady_clock::now();
    auto dijkstra = dijkstrasShortestPath(17, nodes2, dgraph);
    for (auto i = 0; i < dijkstra.size(); i++)
        cout << i << "\t" << dijkstra[i] << endl;
    cout << chrono::duration_cast<chrono::milliseconds>(t2 -
t1).count() << " milliseconds" << endl;
    auto t5 = chrono::steady_clock::now();
    time.show(t4, t5);
    return 0;
}

```

Лістинг 3.6 – Main() програми

Графи *g* і *dgraph* відповідно до заданих параметрів наповнюються випадковими ребрами. Після цього відбувається виклик алгоритмів спочатку Флойда-Уоршелла, потім Дейкстри.

У випадку Дейкстри додатково задається номер вершини, найкоротшу відстань від якої до всіх інших вершин необхідно знайти. На кожному етапі функція `time.show (t1, t2)` показує витрачений час на розрахунки.

Результат розрахунків для алгоритму Флойда-Уоршелла має вигляд. Видно, що через загальну кількість ребер відстані між майже будь-якими вершинами неминуче перебувають. У той же час існують деякі ізольовані вершини – тому відстань до них `SHRT INT MAX`, тобто 32767 (рис. 3.1).

Microsoft Visual Studio Debug Console																				-	📄
0	10	21	11	7	3	14	15	6	8	6	9	11	8	9	11	12	3	6	13		
14	11	9	6	17	11	11	7	7	4	13	13	12	4	7	32767	32767					
9	0	16	5	4	12	13	12	6	2	6	4	9	7	5	16	2	8	11	12		
4	6	3	6	7	8	6	6	12	4	13	9	2	4	3	32767	32767					
13	8	0	12	12	16	17	7	10	10	14	4	17	6	9	24	10	16	15	12		
12	12	10	9	13	11	6	14	20	12	17	9	10	11	11	32767	32767					
9	10	24	0	13	12	19	11	6	12	15	14	7	8	11	20	12	12	15	16		
14	9	12	15	17	17	14	12	16	13	18	13	10	12	9	32767	32767					
10	10	15	4	0	13	20	8	6	6	11	3	11	8	12	21	12	13	16	11		
14	9	2	16	17	4	5	16	17	14	16	8	12	13	13	32767	32767					
11	10	20	15	13	0	11	19	14	12	4	14	19	8	6	22	12	14	17	11		
14	16	13	13	17	15	10	16	18	14	21	13	12	13	13	32767	32767					
12	12	17	6	2	15	0	10	8	5	12	5	13	10	14	23	12	15	15	13		
16	11	4	2	19	6	7	18	19	14	18	10	10	12	15	32767	32767					
17	16	22	11	7	20	16	0	13	13	18	10	18	14	19	28	18	20	8	18		
20	9	9	18	23	11	12	22	24	20	23	15	18	19	19	32767	32767					
9	4	20	9	8	12	17	15	0	6	10	8	13	2	9	20	6	12	15	10		
8	10	7	10	11	11	10	10	16	8	15	7	6	7	7	32767	32767					
22	19	21	16	12	18	28	20	15	0	11	9	23	17	24	33	7	25	20	17		
23	17	14	11	26	16	11	25	29	9	22	14	19	21	22	32767	32767					
12	14	30	19	16	15	23	24	13	16	0	18	18	12	19	23	16	15	18	7		
14	20	17	15	17	20	20	19	19	13	25	20	16	11	17	32767	32767					
13	10	12	13	14	16	23	11	6	9	14	0	19	8	15	24	12	16	19	8		
14	8	13	16	17	7	2	16	20	14	13	5	12	13	13	32767	32767					
5	15	26	7	12	8	19	18	11	13	11	14	0	13	14	16	17	8	11	18		
19	9	5	11	12	16	16	12	12	9	11	16	17	9	12	32767	32767					
7	2	18	7	6	10	15	14	7	4	8	6	11	0	7	18	4	10	13	14		
6	8	5	8	9	10	8	8	14	6	15	11	4	5	5	32767	32767					
9	4	14	9	8	12	8	13	8	6	10	8	13	2	0	20	6	12	15	10		
8	10	7	10	11	9	4	10	16	8	15	7	6	7	7	32767	32767					
17	10	24	13	9	15	18	17	8	7	8	12	13	10	15	012	18	17	15	14		
8	11	5	17	13	14	16	22	6	15	15	4	6	13	32767	32767						
15	12	14	9	5	11	21	13	8	7	4	2	16	10	17	26	0	18	13	10		
16	10	7	4	19	9	4	18	22	2	15	7	12	14	15	32767	32767					
16	15	25	13	13	10	11	18	16	15	14	13	20	13	15	817	0	3	11	18		
8	15	13	21	17	15	11	4	14	10	15	12	14	13	32767	32767						
20	15	25	14	10	23	8	18	16	13	20	13	21	13	20	31	17	23	0	21		
19	19	12	10	22	14	15	21	27	19	26	18	17	18	18	32767	32767					
5	7	23	12	9	8	19	17	11	9	8	11	12	5	12	16	9	8	11	0		
7	13	10	8	10	13	13	12	12	6	18	14	9	9	10	32767	32767					
5	15	26	12	12	8	19	20	11	13	11	14	5	13	14	16	17	8	11	10		
0	14	10	11	3	16	16	12	12	9	15	7	17	9	12	32767	32767					
12	7	23	5	11	15	20	16	11	9	13	11	12	5	12	23	9	15	18	13		
11	0	10	13	14	15	13	13	19	11	20	16	9	10	10	32767	32767					
8	12	26	2	15	11	21	13	8	14	14	16	9	10	13	19	14	11	14	18		
16	7	0	14	19	19	16	14	15	12	20	15	12	12	11	32767	32767					

Рисунок 3.1 – Результат роботи програми

Результат розрахунків алгоритму Дейкстри для графу з 13 вершинами. Початкова вершина 11, тому шлях від неї дорівнює нулю. Шлях до інших вершин відрізняється, а до вершин 0 та 1 не існує взагалі (рис. 3.2).

0	32767
1	32767
2	26
3	48
4	37
5	54
6	45
7	50
8	68
9	40
10	80
11	0
12	55
13	50

Рисунок 3.2 – Результат роботи програми

Для алгоритму Флойда-Уоршелла зроблено експериментальні виміри часу виконання на графах великого розміру з наступними результатами. Тести були виготовлені на процесорі i5005u 2 GHz в одному потоці (таб. 3.1).

Таблиця 3.1 – Результат тестів алгоритму Флойда-Уоршелла за допомогою Boost Graph Library

Кількість вершин	Кількість ребер	Час виконання (секунд)
1000	300	5.6
2000	600	38.5
3000	900	129.3
6000	1800	948.2
10000	3000	4399.1
12000	3600	9781.1
12000	5400	24349.3

Для алгоритму Дейкстри на графах аналогічних розмірів отримано значення не більше 10 секунд.

Офіційно в Одесі нвлічується близько 1650 вулиць. Як правило, кожна вулиця межує з 2-10 вулицями. Якщо взяти середнє значення як 7, то розрахунок відстаней між усіма вершинами алгоритму Флойда-Уоршелла

займе близько 853 секунд. З цього випливає, що алгоритми Флойда-Уоршелла та Дейкстри застосовуються для розрахунку шляху в межах міст середніх розмірів, таких як Одеса.

В рамках даного розділу роботи були досліджені можливості широко застосовуваної в ІТ-індустрії колекції бібліотек Boost, а саме її підбібліотеки Boost Graph Library для вирішення завдань, пов'язаних із пошуком оптимальних маршрутів. За підсумками роботи можна сказати, що Boost є у край суперечливою бібліотекою. З одного боку, Boost є універсальною та потужною бібліотекою, що широко застосовується в індустрії. Boost Graph Library декларує реалізацію та підтримку множини алгоритмів пошуку шляху. Усі бібліотеки Boost опубліковані з відкритим вихідним кодом та є крос-платформними.

Помітимо, що кросплатформність обертається витратами як занадто абстрактних класів, пряма взаємодія з якими дуже утруднено. Добровільний принцип підтримки бібліотек обертається провалами в документації, яка подекуди застаріла на півтора десятка років, не кажучи вже про адаптацію класів та їх інтерфейсів до сучасних версій C++.

Дані проблеми не є непереборними, і хоча у випадку Boost Graph Library вони є суттєвими, немає причин думати, що інші підбібліотеки колекції бібліотек Boost мають ті ж недоліки, що й Boost Graph Library. З цього можна зробити висновок, що Boost все ще здатний бути певним стандартом для індустрії, проте це вимагає доопрацювання існуючих бібліотек.

Незважаючи на цілий ряд недоліків і труднощів, на основі Boost можна реалізувати алгоритми пошуку шляху, а також оцінити можливості їх масштабування. Можна підвести підсумок, що хоча для дуже великих графів необхідні прогресивніші, ніж Флойд-Уоршелл і Дейкстра алгоритми, їх застосування в рамках окремих міст, порівнянних з розмірами міста Одеса, цілком ефективно і дає результати за реальний час.

4 НАПІВКООПЕРАТИВНИЙ АЛГОРИТМ ДЕЙКСТРИ

На основі кооперативного A^* Девіда Сілвера було створено перехідний алгоритм, який буде умовно називатися напівкооперативним алгоритмом Дейкстри. Повна реалізація алгоритму Девіда Сілвера потребує симуляції трьохвимірного простору з координатами x , y , z та двох окремих структур даних. Програмна реалізація даного алгоритму є відносно складною.

Девід Сілвер протиставляє свій трьохвимірний алгоритм двомірному. В рамках даної роботи розроблено і запропоновано перехідний між некооперативним двовимірним та кооперативним трьохвимірним алгоритм пошуку.

Оскільки без окремої структури даних резервування клітин (cells) у конкретний момент часу неможливо, воно може бути проведено лише шляхом змінення значень ваги у ребер між двома вершинами. Таким чином ми отримуємо наступний алгоритм:

1. Нехай деякому агенту необхідно знайти найкоротший шлях від вершини A до вершини Z .
2. Найкоротший шлях знаходиться згідно алгоритму Дейкстри.
3. Ваги всіх вершини на шляху від A до Z (з вершинами A та Z) збільшуються на довільну величину.
4. Для подальших агентів маршрут шляху від A до Z стане менш привабливим.

Це не дозволяє повністю уникнути всіх можливих конфліктів, але дозволяє зменшити їх кількість.

Відсутність окремої структури даних з резервуванням кожної клітини окремо здатно прискорити виконання алгоритму порівнянно с CA^* і в той самий час частково вирішити проблеми, що виникають у A^* . Рух проходить в декартовому просторі (вгору, вниз, вліво та вправо). Рух під кутом може бути просимульований за допомогою поєднання цих засобів руху. Наприклад, рух

під кутом 60 градусів можна просимулювати як два рухи вліво та один вгору. За необхідності для цього можна додатково збільшити розмір карти.

Програму було реалізовано за допомогою мови C++ без застосування жодних зовнішніх бібліотек. Повний код програми наведено у додатку Б. В даному розділі буде наведено результат роботи головної частини програми (лістинг 4.1).

```
int main() {
    GridWithWeights grid = make_diagram4();
    GridLocation start{ 1, 4 }, goal{ 8, 3 };
    std::unordered_map<GridLocation, GridLocation> came_from;
    std::unordered_map<GridLocation, double> cost_so_far;
    dijkstra_search(grid, start, goal, came_from, cost_so_far);
    draw_grid(grid, nullptr, &came_from, nullptr, &start,
    &goal);
    std::cout << '\n';
    std::vector<GridLocation> path = reconstruct_path(start,
    goal, came_from);
    draw_grid(grid, nullptr, nullptr, &path, &start, &goal);
    std::cout << '\n';
    draw_grid(grid, &cost_so_far, nullptr, nullptr, &start,
    &goal);

    grid = edit_diagram4(grid, path); //new
    draw_grid(grid, &cost_so_far, nullptr, nullptr, &start,
    &goal);

    GridLocation start2{ 2,2 }, goal2{ 9,5 };
    std::unordered_map<GridLocation, GridLocation> came_from2;
    std::unordered_map<GridLocation, double> cost_so_far2;
    dijkstra_search(grid, start2, goal2, came_from2,
    cost_so_far2);
    std::vector<GridLocation> path2 = reconstruct_path(start2,
    goal2, came_from2);
    draw_grid(grid, nullptr, nullptr, &path, &start, &goal);
    draw_grid(grid, &cost_so_far2, nullptr, nullptr, &start2,
    &goal2);
    return 0;}
```

Лістинг 4.1 – головна частина програми

У лістингу 4.1 створюється діаграма, у якій є 4 основні види кліток: звичайні, ліс (пересування у 5 разів більш важке) та вершини, по яким було проведено шлях (в 25 разів більш важкі). Початкову задачу наведено на рис. 4.1).



Рисунок 4. 1 – Початкова задача

Після проходження шляху з рис. 4.1 значення важелів на шляху першого агента зміняться та для наступного агента будуть більшими на 25 (відповідно до заданої константи, яка може відрізнятися). Минулий шлях обведено червоним. Вершини, які не були відвідані, складаються з крапок (рис. 4.2). А – початкова вершина, Z – ціль.

В результаті будування шляху відбувається за неконфліктним маршрутом, який міг би бути неможливим у звичайному алгоритмі Дейкстри (рис. 4.3).

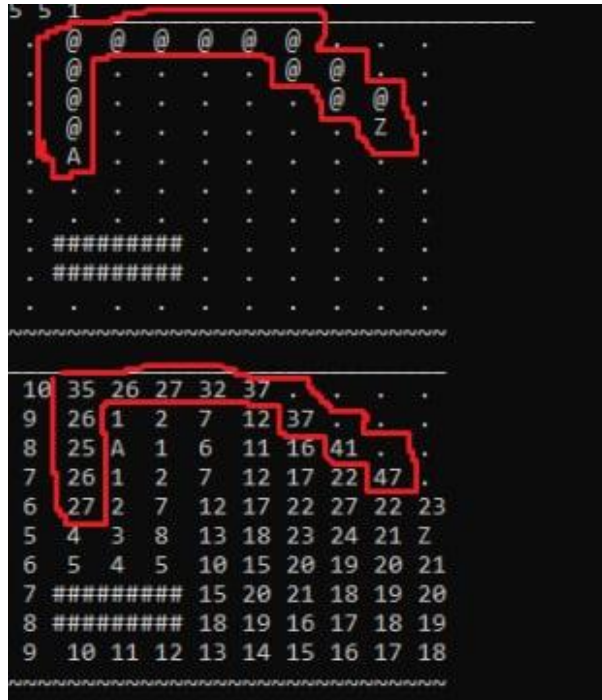


Рисунок 4.2 – Нові ваги на карті

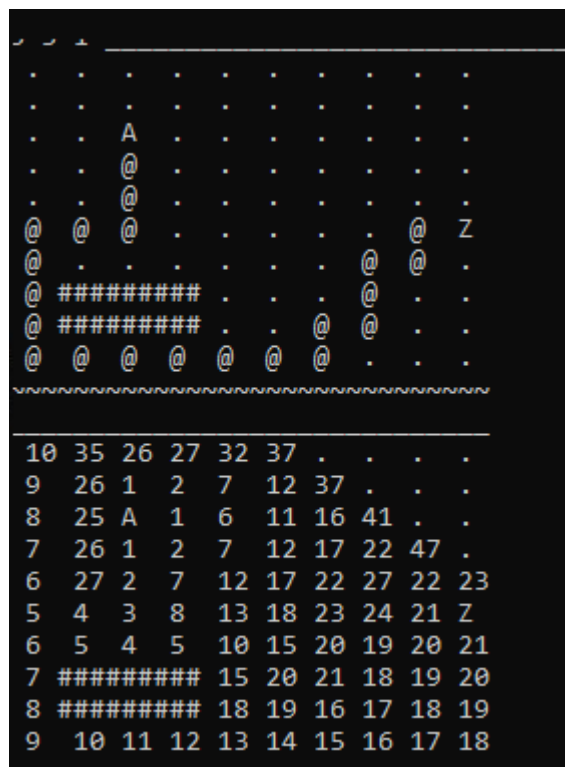


Рисунок 4.3 – Маршрут для другого агента на оновленій карті

Таким чином, зміна ваг вершин дозволяє агентам побудувати неконфліктуючі один з одним маршрути. Значення евристичної функції у запропонованому маршрути було взято за 25. На практиці таке значення може бути як занадто великим, так і занадто малим в залежності від конкретної задачі. Оптимальність важко визначити наперед. Для оптимізації цього значення доцільним є застосування систем штучного інтелекту.

5 ДИНАМІЧНІ ВАГИ

Напівкооперативний алгоритм Дейкстри має подальший потенціал розвитку. Одним з варіантів його удосконалення є утілення поступового повернення системи до початкового стану.

Під початковим станом графу мається на увазі сукупність значень ваги графу до того, як у відповідності до напівкооперативного алгоритму значення ваги у відвіданих вершинах (рис. 5.1) були збільшені. Для цього достатньо на кожному кроці зменшувати будь-яку вагу, поки вона не досягне одиниці.

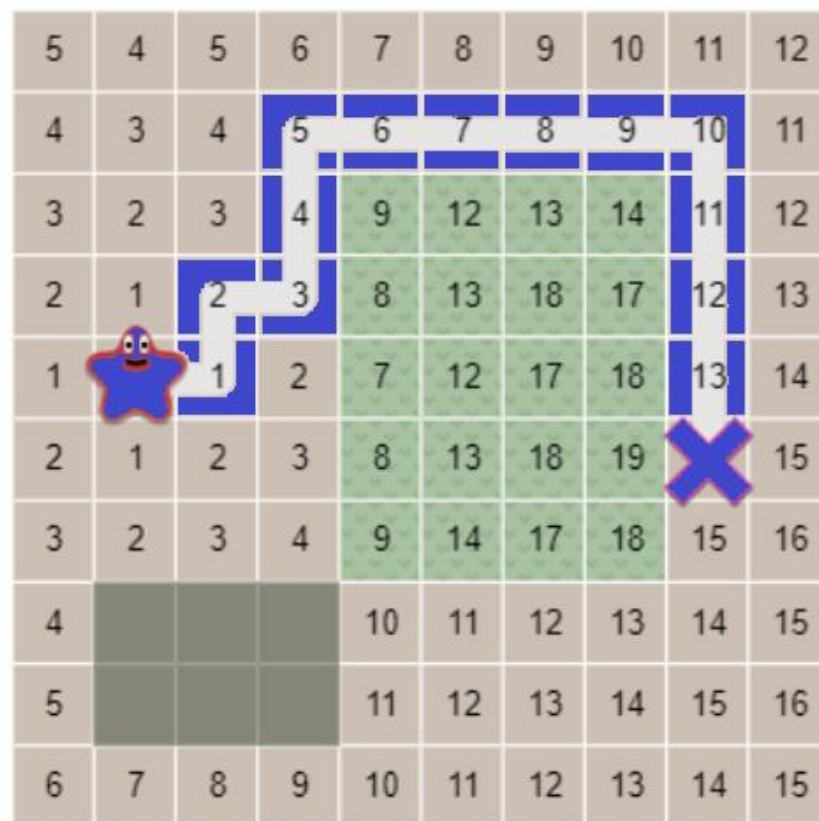


Рисунок 5.1 – Мапа графу, відвідані вершини обведено синім – їх ваги будуть збільшені

Важливо відмітити, що початкові значення можуть бути відмінними від одиниці за допомогою зберігання додаткових ваг вершин в окремому масиві. Таким чином, вагою буде вважатися сума ваг вершин з однаковими

координатами з основного та додаткового масивів. Додатковий масив не буде займати багато місця, адже в ньому буде стільки ж елементів, скільки є вершин в графі, що не створить можливої у деяких кооперативних алгоритмах перспективи експоненційного зростання використовуємої оперативної пам'яті [9]. В рамках цього розділу реалізовано саме таку модифікацію алгоритму.

Для цього необхідно створити два масиви та функцію `turn` (лістинг 5.1). За допомогою функції `turn()` першого масиву `weights` будуть поступово повертатися до одиниці. Для того, щоб відобразити на графі можливість вершин, початкова вага яких більша за одиницю, створено додатковий масив `sw`. Ціною руху до вершини буде їх сума, що дозволяє задати будь-яку початкову вагу вершини.

```
struct GridWithWeights : SquareGrid {
    int weights[SIZE][SIZE] = {};
    int cw[SIZE][SIZE] = {}; //const weights
    GridWithWeights(int w, int h) : SquareGrid(w, h) {}
    double cost(GridLocation from_node, GridLocation to_node)
const {
    /*std::cout << (std::max((forests.find(to_node) !=
forests.end() ? 5 : 1),
    (visited.find(to_node) != visited.end() ? 25 : 1)))
<< ' ';*/
    return
weights[to_node.yGridLocation()][to_node.xGridLocation()]
+cw[to_node.yGridLocation()][to_node.xGridLocation()];
};
void turn(int t = 1) {
    for (auto i = 0; i < SIZE; i++)
        for (auto j = 0; j < SIZE; j++)
            if (weights[i][j] > t)
                weights[i][j] -= t;
}
};
```

Лістинг 5.1 – динамічні ваги

Приклад використання функції `turn` (рис 5.2). З лівого боку наведено значення ваг вершин після реалізації `turn` с аргументом 9 (всі відвідувані вершини зменшуються на 9), з правого боку – немодифікований варіант програми з розділу 4.

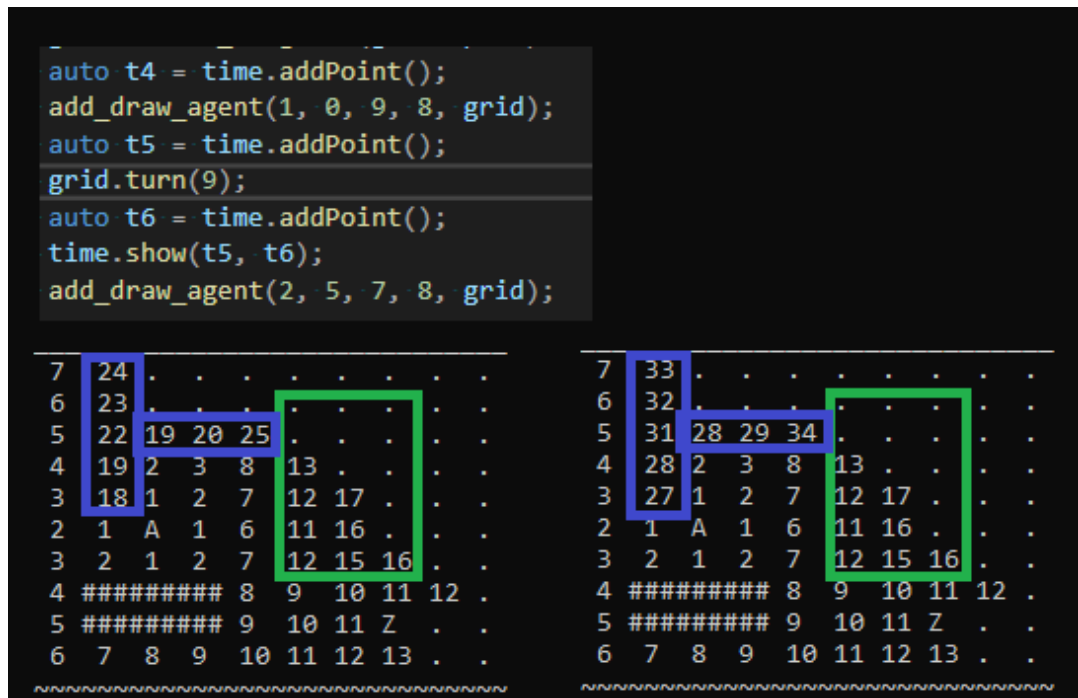


Рисунок 5.2 – Результат роботи функції `turn()`

Наведемо приклад використання данної модифікації алгоритму для декількох агентів із використанням зміни ваг вершин. Нехай існує три агенти, яким необхідно добратися в одну й ту ж саму точку. Можна йти прямо, обійти з півночі або з півдня (рис. 5.3).

4	5	6	7	8	9	10	11	12	13
3	4	5	6	7				13	14
2	3	4	5	6				14	15
1	2	3	4	5	10	15	20	15	16
			3	4	9	14	19		15
3	4	5	6	7	12	17	20	15	16
4	5	6	7	8	13	16	17	14	15
5					9	10	11	12	13
6					10	11	12	13	14
7	8	9	10	11	12	13	14	15	16

Рисунок 5.3 – Початкова задача для трьох агентів

Нехай спочатку ходить синій агент, після нього – червоний, а після нього – помаранчевий (рис. 5.4). При значенні NEUR 25 та без використання функції turn() агенти оберуть три шляхи, які не конфліктують один з одним. Вартість шляху для кожного з трьох агентів буде відповідно 15, 16 та 20.

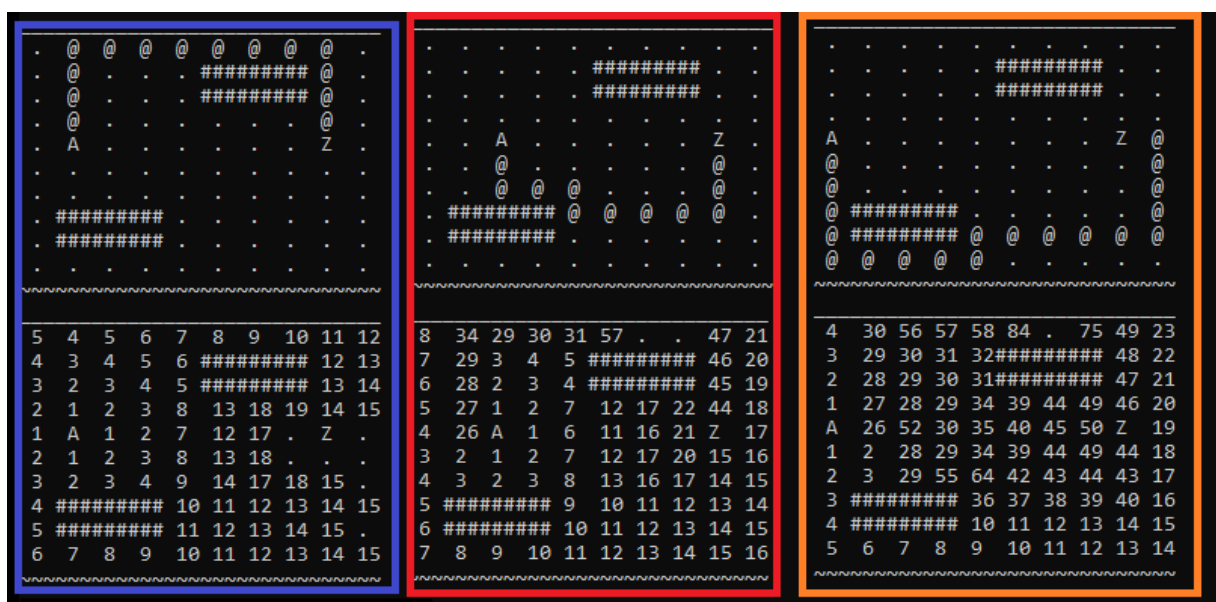


Рисунок 5.4 – Результат роботи напівкооперативного алгоритму. Шляхи обведено відповідно до кольору агентів з рис. 5.3

Продемонструємо приклад роботи функції `turn()`. Припустимо, що помаранчевий агент вибрав свій шлях на 25 кроків пізніше, ніж синій та червоний. Тоді результат буде наступним (рис. 5.5). В результаті роботи `turn()` для помаранчевого агента відстані відповідали початковим значенням графа, і на відміну від ситуації на рис. 5.4 він не вважав північний шлях зарезервованим за іншими агентами. Як наслідок, він вибрав той самий шлях до цілі, що і синій агент до нього. Вартості шляху агентів у такому разі становитимуть 15, 16 та 16 одиниць – вартість шляху для помаранчевого агента скоротилася на 4 одиниці за рахунок можливості використання того ж самого північного шляху, що був обраний синім агентом.

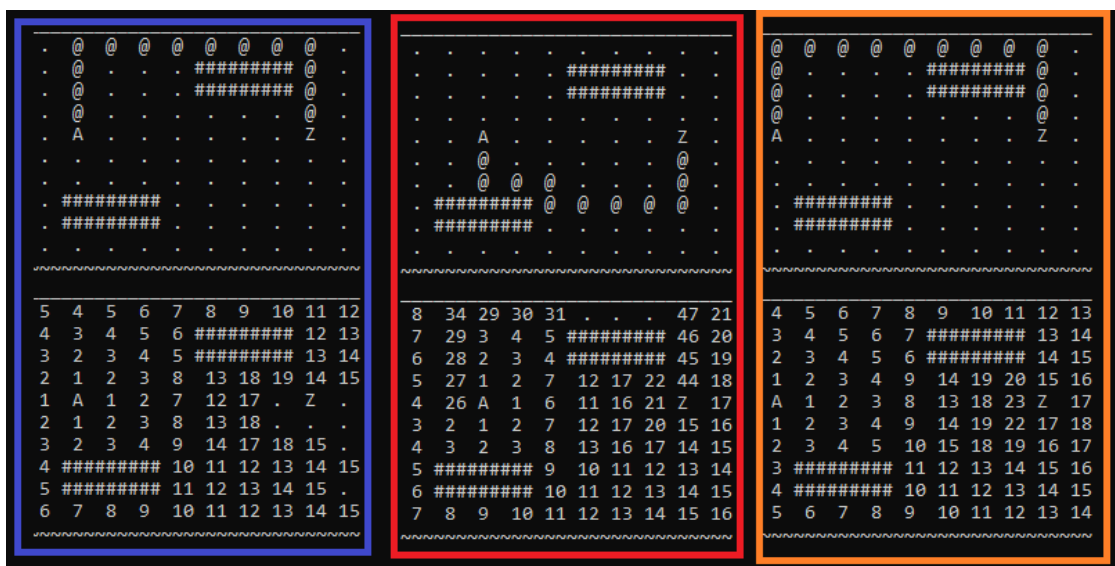


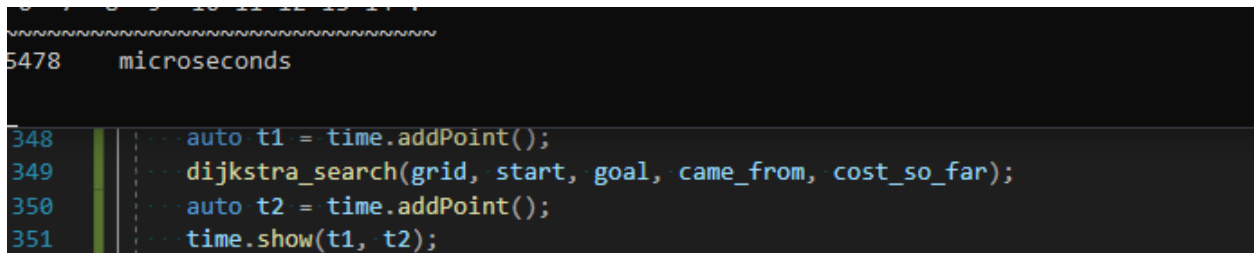
Рисунок 5.5 – Результат роботи напівкооперативного алгоритму с `turn(25)` після другого агента

Таким чином, полукооперативний алгоритм дозволяє агентам вибирати шляхи, що не конфліктують один з одним. Крім цього, завдяки динамічним вагам з часом (плин якого можна відобразити за допомогою `turn()`) шляхи не будуть резервуватися назавжди, а лише на деякий час, який визначається значенням `NEUR`.

6 ОЦІНКА ЕФЕКТИВНОСТІ МОДИФІКОВАНОГО АЛГОРИТМУ

Для порівняння швидкодії стандартного алгоритму Дейкстри та спроектованих і реалізованих у розділах 4 та 5 напівкооперативних модифікацій цього алгоритму доречним є використання створеного під часу роботи з бібліотекою Boost Graph Library класу Time, що використовує стандартну бібліотеку C++ <chrono>.

Порівняно 3 модифікації алгоритму: алгоритм Дейкстри, напівкооперативний алгоритм Дейкстри, напівкооперативний алгоритм Дейкстри з динамічною вагою. У разі графу розміром 10 на 10 алгоритм Дейкстри знайшов ціль за 5478 мікросекунд (рис. 6.1).



```

5478      microseconds

348      auto t1 = time.addPoint();
349      dijkstra_search(grid, start, goal, came_from, cost_so_far);
350      auto t2 = time.addPoint();
351      time.show(t1, t2);

```

Рис. 6.1 – Стандартний алгоритм Дейкстри

У графі тих ж самих розмірів напівкооперативний алгоритм Дейкстри знайшов ціль за 4914 мікросекунд (рис. 6.2), але додатково витратив 237 мікросекунд за змінення збільшення значень ваги у пройденому маршруті. Також було виконано повернення значень ваг вершин до минулих значень шляхом 25 викликів функції grid.turn(), що зайняло додаткові 12 мікросекунд.

ВИСНОВОК

У результаті дипломної роботи спроектовано та реалізовано напівкооперативний алгоритм раціонального планування маршрутів для декількох агентів, уникаючих взаємних конфліктів на основі використовуваних підходів рішення задач планування маршрутів. Реалізований метод пропонує власну евристичну функцію, значення якої для кожних задач може бути оптимізовано системами штучного інтелекту. Реалізований алгоритм має потенціал вдосконалення шляхом додавання графіки у таких двигунах, як Unity.



/Коган В.В./

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. G. Chapuis, R. Andonov. Warshall Algorithm. GPU-accelerated shortest paths computations for planar graphs [Електронний ресурс] – Режим доступу: <https://www.sciencedirect.com/topics/computer-science/warshall-algorithm>
2. Wahyu Hidayat. A Comparative Study of Informed and Uninformed Search Algorithm to Solve Eight-Puzzle Problem [Електронний ресурс] – Режим доступу: <https://thescipub.com/pdf/jcssp.2021.1147.1156.pdf>
3. Daviel Foead. A Systematic Literature Review of A* Pathfinding [Електронний ресурс] – Режим доступу: <https://www.sciencedirect.com/science/article/pii/S1877050921000399>
4. Patrick Lester. A* Pathfinding [Електронний ресурс] – Режим доступу: <http://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf>
5. David Silver. Cooperative Pathfinding [Електронний ресурс] – Режим доступу: <https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIWisdom.pdf>
6. Co3us. Cooperative Pathfinding [Електронний ресурс] – Режим доступу: <https://github.com/Co3us/CoopertivePathfinding>
7. Junhao Li. HPS: A C++11 High Performance Serialization Library HPS: A C++11 High Performance Serialization Library [Електронний ресурс] – Режим доступу: <https://arxiv.org/ftp/arxiv/papers/1811/1811.04556.pdf>
8. Nick Edmons, Alex Breuer, Douglas Gregor. Single-Source Shortest Paths with the Parallel Boost Graph library [Електронний ресурс] – Режим доступу: <http://users.diag.uniroma1.it/challenge9/papers/edmonds.pdf>
9. Jinmingwu Jiang. Cooperative Pathfinding based on memory-efficient Multi-agent RRT* [Електронний ресурс] – Режим доступу: <https://researchchain.net/archives/pdf/Cooperative-Pathfinding-Based-On-Multi-Agent-Rrt-Fixed-Node-2146163>
10. David Silver. Cooperative Pathfinding [Електронний ресурс] – Режим доступу: <https://ojs.aaai.org/index.php/AIIDE/article/view/18726/18503>

ДОДАТОК А. Код програми за допомогою бібліотеки Boost

```

#include <boost/graph/floyd_warshall_shortest.hpp>
#include "boost/graph/random.hpp"
#include <stdlib.h>
#include <iostream>
#include <iterator>
#include <chrono>
#include <set>
#define INF SHRT_MAX

using namespace std;
using namespace boost;
typedef boost::property<boost::edge_weight_t, int>
EdgeWeightProperty;
typedef boost::adjacency_list<boost::listS, boost::vecS,
boost::directedS, boost::no_property, EdgeWeightProperty >
DirectedGraph;
typedef boost::graph_traits<DirectedGraph>::out_edge_iterator
oe_it;
typedef boost::graph_traits<DirectedGraph>::edge_iterator e_it;
typedef boost::graph_traits<DirectedGraph>::vertex_iterator
v_it;
typedef boost::graph_traits<DirectedGraph>::vertex_descriptor
Vertex;
typedef boost::graph_traits<DirectedGraph>::edge_descriptor
Edge;
typedef std::chrono::steady_clock::time_point tp;
DirectedGraph g, f;
typedef pair<int, int> PII;
typedef vector<PII> VPII;
typedef vector<VPII> VVPII;

vector<vector<int>> floydWarshall(DirectedGraph g) {
    vector<vector<int>> distances;
    unsigned long vNum = num_vertices(g);
    distances.resize(vNum);
    pair<v_it, v_it> iterators = vertices(g);
    for (v_it it = iterators.first; it != iterators.second;
++it) {
        for (v_it it2 = iterators.first; it2 !=
iterators.second; ++it2) {
            if (it == it2) {
                distances[*it].push_back(0);
                continue;
            }
            else distances[*it].push_back(INF);
        }
    }
}

```

```

    pair<e_it, e_it> edIt = edges(g);
    for (e_it it = edIt.first; it != edIt.second; ++it) {
        distances[boost::source(*it, g)][boost::target(*it, g)]
= boost::get(boost::edge_weight_t(), g, *it);
    }

    for (int k = 0; k < vNum; ++k) {
        for (int i = 0; i < vNum; ++i) {
            for (int j = 0; j < vNum; ++j) {
                if (distances[i][j] > distances[i][k] +
distances[k][j])
                    distances[i][j] = distances[i][k] +
distances[k][j];
            }
        }
    }

    return distances;
}

vector<unsigned int> dijkstrasShortestPath(int source_node, int
node_count, VVPII& graph) {
    vector<unsigned int> dist(node_count, INF); // Assume that
the distance from source_node to other nodes is infinite
    set<PII> set_length_node; // in the beginnging, i.e
initialize the distance vector to a max value
    dist[source_node] = 0; // Distance from starting vertex to
itself is 0
    set_length_node.insert(PII(0, source_node));
    while (!set_length_node.empty()) {
        PII top = *set_length_node.begin();
        set_length_node.erase(set_length_node.begin());
        int current_source_node = top.second;
        for (auto& it : graph[current_source_node]) {
            int adj_node = it.first;
            int length_to_adjnode = it.second;
            if (dist[adj_node] > length_to_adjnode +
dist[current_source_node]) { // Edge relaxation
                if (dist[adj_node] != INF) {// If the distance
to the adjacent node is not INF, means the pair <dist, node> is
in the set

set_length_node.erase(set_length_node.find(PII(dist[adj_node],
adj_node)));
                } // Remove the pair before
updating it in the set.
                dist[adj_node] = length_to_adjnode +
dist[current_source_node];
                set_length_node.insert(PII(dist[adj_node],
adj_node));
            }
        }
    }
}

```

```

    }
}
for (int i = 0; i < node_count; i++)
    //cout << "Source Node(" << source_node << ") ->
Destination Node(" << i << ") : " << dist[i] << endl;
return dist;
}
void insertEdge(unsigned long s, unsigned long d, int w,
DirectedGraph& a) {
    add_edge(s, d, w, a);
}

void fillGraph(int n, int g1, int g2, int w1, int w2,
DirectedGraph& a, VVPII& d) { //add n edges in graphs between g1
and g2 with weights between w1 and w2
    int s = min(g1,g2); //source node
    while (n > 0) {
        int c = rand() % 6+2; //number of edges of source node
        PII tmp = {}; //for Dijkstra
        VPII tmp2 = {};
        for (int i = 0; i < c; i++) {
            int d = rand() % ((max(g1, g2)) - (min(g1, g2)) - 1)
+ g1;//any order of g numbers
            //cout << d << endl;
            int w = rand() % ((max(w1, w2)) - (min(w1, w2)) - 1)
+ w1;//any order of w numbers
            //cout << s << endl;
            if (s != d) {
                PII tmp = { w,d };
                tmp2.push_back(tmp);
                //cout << s << "\t" << d << endl;
                insertEdge(s, d, w, a);
                --n;
            }
        }
        d.push_back(tmp2);
        if (s < max(g1,g2))
            s++;
        else
            s=min(g1,g2);
    }
}

class Time {
public:
    static void show(tp t1, tp t2){ //time passed since t1
        cout << chrono::duration_cast<chrono::milliseconds>(t2 -
t1).count() << " milliseconds" << endl;
    }
    tp addPoint() {

```

```

        tp p = chrono::steady_clock::now();
        return p;
    }
};

int main()
{
    VVPII dgraph = {};
    Time time;
    auto t1 = time.addPoint();
    int nodes1 = 0;
    int nodes2 = 30;
    fillGraph(150, nodes1, nodes2, 2, 15, g, dgraph);
    //insertEdge(1, 2, 3, g); insertEdge(2, 3, 4, g);
    insertEdge(3, 1, 1, g);
    auto t2 = time.addPoint();
    //time.show(t1, t2);
    vector<vector<int>> FloydWarhsall = floydWarshall(g); //Here
    we get the minimum distances from all nodes to other ones using
    Floyd-Warshall's algorithm
    auto t3 = chrono::steady_clock::now();
    time.show(t2, t3);
    for (auto i = 0; i < FloydWarhsall.size(); i++) {
        for (auto j = 0; j < FloydWarhsall.size(); j++) {
            cout << FloydWarhsall[i][j]<< "\t";
        }
        cout << endl;
    }
    auto t4 = chrono::steady_clock::now();
    auto dijkstra = dijkstrasShortestPath(11, nodes2, dgraph);
    for (auto i = 0; i < dijkstra.size(); i++)
        cout << i << "\t" << dijkstra[i] << endl;
    cout << chrono::duration_cast<chrono::milliseconds>(t2 -
t1).count() << " milliseconds" << endl;
    auto t5 = chrono::steady_clock::now();
    //time.show(t4, t5);
    return 0;
}

```

ДОДАТОК В. Код напівкооперативного алгоритму Дейкстри з динамічними вагами

```

#include <iostream>
#include <iomanip>
#include <unordered_map>
#include <unordered_set>
#include <array>
#include <vector>
#include <utility>
#include <queue>
#include <tuple>
#include <algorithm>
#include <cstdlib>
#include <chrono>
#include <time.h>
#define HEUR 25
#define SIZE 10 //not less than 10
typedef std::chrono::steady_clock::time_point tp;

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");
        std::cout <<
        std::chrono::duration_cast<std::chrono::microseconds>(t2 -
t1).count() << '\t';
        printf("microseconds \n");
    }
};

```

```

    }
    tp addPoint() {
        tp p = std::chrono::steady_clock::now();
        return p;
    }
};

struct GridLocation {
    int x, y;
    int xGridLocation() { return this->x; }
    int yGridLocation() { return this->y; }
};

std::vector<GridLocation> visitedGrid = {};
std::vector<int> visitedInt = {};
bool same_grid(GridLocation l1, GridLocation l2) {
    return (l1.xGridLocation() == l2.xGridLocation()
        && l1.yGridLocation() == l2.yGridLocation());
}

int in_visited(std::vector<GridLocation> visitedGrid,
GridLocation loc) {
    for (auto i = 0; i < visitedGrid.size(); i++)
    {
        if (same_grid(visitedGrid[i], loc))
            return i;
    }
    return -1;
}

namespace std {
    /* implement hash function so we can put GridLocation into
    an unordered_set */
    template <> struct hash<GridLocation> {

```

```

        std::size_t operator()(const GridLocation& id) const
noexcept {
            // NOTE: better to use something like boost
hash_combine
            return std::hash<int>()(id.x ^ (id.y << 16));
        }
    };
}

```

```

struct SquareGrid {
    static std::array<GridLocation, 4> DIRS;

    int width, height;
    std::unordered_set<GridLocation> walls;

    SquareGrid(int width_, int height_)
        : width(width_), height(height_) {}

    bool in_bounds(GridLocation id) const {
        return 0 <= id.x && id.x < width
            && 0 <= id.y && id.y < height;
    }

    bool passable(GridLocation id) const {
        return walls.find(id) == walls.end();
    }

    std::vector<GridLocation> neighbors(GridLocation id) const {
        std::vector<GridLocation> results;

        for (GridLocation dir : DIRS) {

```

```

        GridLocation next{ id.x + dir.x, id.y + dir.y };
        if (in_bounds(next) && passable(next)) {
            results.push_back(next);
        }
    }

    if ((id.x + id.y) % 2 == 0) {
        // see "Ugly paths" section for an explanation:
        std::reverse(results.begin(), results.end());
    }

    return results;
}

};

std::array<GridLocation, 4> SquareGrid::DIRS = {
    /* East, West, North, South */
    GridLocation{1, 0}, GridLocation{-1, 0},
    GridLocation{0, -1}, GridLocation{0, 1}
};

// Helpers for GridLocation

bool operator == (GridLocation a, GridLocation b) {
    return a.x == b.x && a.y == b.y;
}

bool operator != (GridLocation a, GridLocation b) {
    return !(a == b);
}

```



```

bool operator < (GridLocation a, GridLocation b) {
    return std::tie(a.x, a.y) < std::tie(b.x, b.y);
}

std::basic_ostream<char>::basic_ostream&
operator<<(std::basic_ostream<char>::basic_ostream& out, const
GridLocation& loc) {
    out << '(' << loc.x << ',' << loc.y << ')';
    return out;
}

// This outputs a grid. Pass in a distances map if you want to
print
// the distances, or pass in a point_to map if you want to print
// arrows that point to the parent location, or pass in a path
vector
// if you want to draw the path.
template<class Graph>
void draw_grid(const Graph& graph,
    std::unordered_map<GridLocation, double>* distances =
nullptr,
    std::unordered_map<GridLocation, GridLocation>* point_to =
nullptr,
    std::vector<GridLocation>* path = nullptr,
    GridLocation* start = nullptr,
    GridLocation* goal = nullptr) {
    const int field_width = 3;
    std::cout << std::string(field_width * graph.width, '_') <<
'\n';
    for (int y = 0; y != graph.height; ++y) {
        for (int x = 0; x != graph.width; ++x) {
            GridLocation id{ x, y };
            if (graph.walls.find(id) != graph.walls.end()) {

```

```

        std::cout << std::string(field_width, '#');
    }
    else if (start && id == *start) {
        std::cout << " A ";
    }
    else if (goal && id == *goal) {
        std::cout << " Z ";
    }
    else if (path != nullptr && find(path->begin(),
path->end(), id) != path->end()) {
        std::cout << " @ ";
    }
    else if (point_to != nullptr && point_to->count(id))
    {
        GridLocation next = (*point_to)[id];
        if (next.x == x + 1) { std::cout << " > "; }
        else if (next.x == x - 1) { std::cout << " < "; }

        else if (next.y == y + 1) { std::cout << " v "; }

        else if (next.y == y - 1) { std::cout << " ^ "; }

        else { std::cout << " * "; }
    }
    else if (distances != nullptr && distances-
>count(id)) {
        std::cout << ' ' << std::left <<
std::setw(field_width - 1) << (*distances)[id];
    }
    else {
        std::cout << " . ";
    }
}

```

```

        std::cout << '\n';
    }

    std::cout << std::string(field_width * graph.width, '~') <<
'\n';
}

void add_rect(SquareGrid& grid, int x1, int y1, int x2, int y2)
{
    for (int x = x1; x < x2; ++x) {
        for (int y = y1; y < y2; ++y) {
            grid.walls.insert(GridLocation{ x, y });
        }
    }
}

SquareGrid make_diagram1() {
    SquareGrid grid(30, 15);
    add_rect(grid, 3, 3, 5, 12);
    add_rect(grid, 13, 4, 15, 15);
    add_rect(grid, 21, 0, 23, 7);
    add_rect(grid, 23, 5, 26, 7);
    return grid;
}

struct GridWithWeights : SquareGrid {
    int weights[SIZE][SIZE] = {};
    int cw[SIZE][SIZE] = {}; //const weights
    GridWithWeights(int w, int h) : SquareGrid(w, h) {}
    double cost(GridLocation from_node, GridLocation to_node)
const {
        /*std::cout << (std::max((forests.find(to_node) !=
forests.end() ? 5 : 1),

```

```

        (visited.find(to_node) != visited.end() ? 25 : 1)))
<< ' ' ;*/

    return
weights[to_node.yGridLocation()][to_node.xGridLocation()]

+cw[to_node.yGridLocation()][to_node.xGridLocation()];
};

void turn(int t = 1) {
    for (auto i = 0; i < SIZE; i++)
        for (auto j = 0; j < SIZE; j++)
            if (weights[i][j] > t)
                weights[i][j] -= t;
}

};

GridWithWeights make_diagram4(std::unordered_set<GridLocation>
tmp = {}) {
    GridWithWeights grid(SIZE, SIZE);
    add_rect(grid, 1, 7, 4, 9);
    for (auto i = 0; i < SIZE; i++)
        for (auto j = 0; j < SIZE; j++)
            grid.weights[i][j] = 1;
    for (auto i = 3; i < 7; i++) {
        for (auto j = 4; j < 8; j++) {
            grid.cw[i][j] = 4; //forests with weights +4
        }
    }
    /*std::cout << '\n';
    for (auto i : tmp)
        grid.visited.insert(i);
    std::unordered_set<GridLocation> tmp2 = {};
    L ex = { 5,0 };
```

```

    tmp2.insert(ex);
    for (auto i : tmp2)
        grid.visited.insert(i);
    for (auto element : grid.visited)
    {
        std::cout << '(' << element.x << ", " << element.y << "
";
    }
    std::cout << '\n';*/
    //grid.visited = { L {5,0} };
    return grid;
}

GridWithWeights edit_diagram4(GridWithWeights& grid, std::vector
<GridLocation> path)
{
    typedef GridLocation L;
    bool arr[SIZE][SIZE] = { 0 };
    for (auto i : path) {
        arr[i.yGridLocation()][i.xGridLocation()] = true;
    }
    int c = 0;
    std::unordered_set <GridLocation> tmp = {};
    for (auto i = 0; i < SIZE; i++)
        for (auto j = 0; j < SIZE; j++) {
            if (arr[i][j])
                grid.weights[i][j] += HEUR;
        }
    //grid = make_diagram4(tmp);
    return grid;
}

template<typename T, typename priority_t>

```

```

struct PriorityQueue {
    typedef std::pair<priority_t, T> PQElement;
    std::priority_queue<PQElement, std::vector<PQElement>,
        std::greater<PQElement>> elements;

    inline bool empty() const {
        return elements.empty();
    }

    inline void put(T item, priority_t priority) {
        elements.emplace(priority, item);
    }

    T get() {
        T best_item = elements.top().second;
        elements.pop();
        return best_item;
    }
};

template<typename Location, typename Graph>
void dijkstra_search
(Graph graph,
    Location start,
    Location goal,
    std::unordered_map<Location, Location>& came_from,
    std::unordered_map<Location, double>& cost_so_far)
{
    PriorityQueue<Location, double> frontier;
    frontier.put(start, 0);

```

```

came_from[start] = start;
cost_so_far[start] = 0;

while (!frontier.empty()) {
    Location current = frontier.get();

    if (current == goal) {
        break;
    }

    for (Location next : graph.neighbors(current)) {
        double new_cost = cost_so_far[current] +
graph.cost(current, next);

        if (cost_so_far.find(next) == cost_so_far.end()
            || new_cost < cost_so_far[next]) {
            cost_so_far[next] = new_cost;
            came_from[next] = current;
            frontier.put(next, new_cost);
        }
    }
}

template<typename Location>
std::vector<Location> reconstruct_path(
    Location start, Location goal,
    std::unordered_map<Location, Location> came_from
) {
    std::vector<Location> path;
    Location current = goal;

    while (current != start) { // note: this will fail if no
path found

```

```

        path.push_back(current);
        current = came_from[current];
    }

    path.push_back(start); // optional
    std::reverse(path.begin(), path.end());
    return path;
}

void add_draw_agent (int s1, int s2, int d1, int d2,
GridWithWeights& grid)
{
    GridLocation start{ s1,s2 }, goal{ d1,d2 };
    std::unordered_map<GridLocation, GridLocation> came_from;
    std::unordered_map<GridLocation, double> cost_so_far;
    dijkstra_search(grid, start, goal, came_from, cost_so_far);
    std::vector<GridLocation> path = reconstruct_path(start,
goal, came_from);
    draw_grid(grid, nullptr, nullptr, &path, &start, &goal);
    draw_grid(grid, &cost_so_far, nullptr, nullptr, &start,
&goal);
    edit_diagram4(grid, path);
};

int main() {
    Time time;
    GridWithWeights grid = make_diagram4();
    GridLocation start{ 1, 4 }, goal{ 8, 3 };

```



```

    std::unordered_map<GridLocation, GridLocation> came_from;
    std::unordered_map<GridLocation, double> cost_so_far;
    auto t1 = time.addPoint();
    dijkstra_search(grid, start, goal, came_from, cost_so_far);
    auto t2 = time.addPoint();

    draw_grid(grid, nullptr, &came_from, nullptr, &start,
&goal);

    std::cout << '\n';

    std::vector<GridLocation> path = reconstruct_path(start,
goal, came_from);

    draw_grid(grid, nullptr, nullptr, &path, &start, &goal);
    std::cout << '\n';

    draw_grid(grid, &cost_so_far, nullptr, nullptr, &start,
&goal);

    auto t3 = time.addPoint();
    grid = edit_diagram4(grid, path); //new
    auto t4 = time.addPoint();
    add_draw_agent(1, 0, 9, 8, grid);
    auto t5 = time.addPoint();
    for (auto i = 0; i < HEUR; i++)
        grid.turn();
    auto t6 = time.addPoint();
    add_draw_agent(2, 5, 7, 8, grid);

    time.show(t1, t2);
    time.show(t3, t4);
    time.show(t5, t6);

    return 0;
}

```