Хмельницький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерно-інтегрованих технологій та робототехніки

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Система керування переміщенням робота
Назва теми

КвРАКІТ.2021050.01.03 ПЗ

Галузь знань 15 «Автоматизація та приладобудування»
Шифр, назва

Спеціальність 151 «Автоматизація та комп'ютерно-інтегровані технології»
Шифр, назва

Освітня програма «Автоматизація та комп'ютерно-інтегровані технології»
Назва

Виконав:

студент IV курсу, група АКІТс-21-1 _____ Владислав ГАЛІБРОДА
Підпис, Ім'я, ПРІЗВИЩЕ

Керівник _____ Денис МАКАРИШКІН
Підпис, дата, Ім'я, ПРІЗВИЩЕ

Нормоконтролер _____ Людмила КОРЕЦЬКА
Підпис, дата, Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
зав. кафедри автоматизації,
комп'ютерно-інтегрованих
технологій та робототехніки _____ Валерій МАРТИНЮК
Підпис, дата, Ім'я, ПРІЗВИЩЕ

«17» червня 2024 р.

Хмельницький 2024

Хмельницький національний університет

Факультет *інформаційних технологій*
Кафедра *автоматизації, комп'ютерно-інтегрованих технологій та робототехніки*
Освітній рівень *перший (бакалаврський)*
Галузь знань *15 – Автоматизація та приладобудування*
Спеціальність *151 – Автоматизація та комп'ютерно-інтегровані технології*
Освітня-професійна програма *Автоматизація та комп'ютерно-інтегровані технології*

ЗАТВЕРДЖУЮ
Зав. кафедрою АКІТтаР
Валерій МАРТИНЮК
«10» 01 2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Галіброда Владислав Олександрович

1 Тема роботи: *Система керування переміщенням робота*
керівник роботи Макаришкін Д.А., к.т.н, доцент
Затверджено наказом по університету від «15» 02 2024 р. № 8.
2 Строк подання студентом роботи на кафедру: 01.06.24р
3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування
4 Зміст пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Огляд та аналіз особливостей процесу пошуку оптимального шляху. Основна частина. Розробка проєкту алгоритму переміщення роботу. Розробка алгоритму переміщення роботу. Висновки.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали.

Завдання отримав _____

Науковий керівник _____

Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Федула М.В., доцент кафедри АКІТтаР		
Нормоконтроль	Корецька Л.О., доцент кафедри АКІТтаР		

7. Дата видачі завдання «10» 01 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів (розділів) дипломної роботи	Строк виконання етапів дипломної роботи	Примітка
1	Вступ	15.02.2024р.	Виконано
2	Огляд та аналіз особливостей процесу пошуку оптимального шляху	15.03.2024р.	Виконано
3	Основна частина	10.04.2024р.	Виконано
4	Розробка проєкту алгоритму переміщення роботу. Розробка алгоритму переміщення роботу.	10.05.2024р.	Виконано
5	Висновки	15.05.2024р.	Виконано
6	Оформлення пояснювальної записки до КРБ	25.05.2024р.	Виконано
7	Оформлення презентаційних матеріалів	1.06.2024р.	Виконано

Студент _____ В.С. Галіброда
Підпис Ініціали, прізвище

Керівник роботи _____ Д.А. Макаришкін
Підпис Ініціали, прізвище

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Система керування переміщенням робота».
Автор роботи: Галіброда Владислав Олександрович
Керівник роботи: Макаришкін Д.А. к.т.н., доцент
Пояснювальна записка: 71 с., 52 рис., 1 табл., 2 дод., 40 джерел.
Графічна частина: Презентаційні матеріали.

СИСТЕМА КЕРУВАННЯ ПЕРЕМІЩЕННЯМ, АЛГОРИТМИ ПОШУКУ ТРАЄКТОРІЇ, ГРАФИ, МОБІЛЬНА РОБОТИЗОВАНА СИСТЕМА, МОДУЛЬ КЕРУВАННЯ.

Метою роботи є розробка системи керування переміщення робота. Проведено огляд та аналіз існуючих алгоритмів пошуку оптимальної траєкторії руху робота на основі теорії графів. Було обрано алгоритм пошуку оптимальної траєкторії руху робота. Розроблено проєкт загальної архітектури всієї системи керування та загальну архітектуру модуля керування. Виконано підбір основних класів, всі розроблені коди програми наведені в додатках до пояснювальної записки.

Проведено моделювання роботу алгоритму переміщення робота із графом та роботу алгоритму пошуку оптимальної траєкторії руху із вагою ребер та евристичною функцією. Реалізовано розроблений алгоритм переміщення мобільної роботизованої системи, виконано порівняння двох використаних алгоритмів пошуку оптимальної траєкторії руху мобільної системи.

01.06.2024р.
дата

Підпис

ЗМІСТ

ВСТУП

Останнім часом, так звані, мобільні роботизовані системи (МРС) та роботизовані комплекси починають відігравати все більшу роль при виконанні різних завдань без втручання людини-оператора. Для деяких завдань, таких як дослідження незнайомої місцевості, пошук та переміщення в небезпечних для людини ділянках, навіть створюються спеціальні вузько-спеціалізовані роботи.

Дані завдання передбачають використання знань із декількох галузей, так чи інакше пов'язаних із галуззю робототехніки: наприклад, механіка, системи керування та контролю, системи прийому/передачі інформації, штучний інтелект тощо [1-5].

Однією з головних задач, яку ставлять перед МРС, є задача коректного переміщення в робочому середовищі. Одні МРС здатні використовувати власні давачі для визначення інформації про стан навколишнього середовища [1-5], інші лише рухається за заздалегідь встановленим маршрутом, третім усю інформацію передають із спеціального центру керування. Так чи інакше, завжди є якась місцевість, і якась частина програмного забезпечення (ПЗ) має вирішувати завдання переміщення на цій місцевості.

Враховуючи вказане вище постає питання вирішення задачі коректного переміщення МРС в його робочому середовищі з урахуванням різноманітності операційних систем, способів керування та прийняття рішень, роботизованих платформ, а отже, потрібне створення комплексного рішення, що реалізує наступні функції:

- обробку даних про навколишнє середовище;
- пошук шляху в навколишньому середовищі за допомогою використання різних алгоритмів;
- надання декількох стратегій пошуку;
- інтерфейс взаємодії з керуючим компонентом;
- можливість додавання алгоритмів і стратегій пошуку.

Структура навколишнього середовища часто може бути невідома заздалегідь, а людина-оператор робота може не володіти повною інформацією про її рельєф. У такому разі, МРС зазвичай здатний самостійно розпізнати зміни за допомогою давачів і або самостійно прийняти рішення, або передати отримані дані в центр керування людині-оператору [1-5]. Делегування прийняття рішення на основі отриманих нових даних від ручного аналізу та оновлення карти до програмного модуля здатне зменшити час прийняття рішення, зменшити величину помилки і повернутися до виконання основного завдання, поставленого перед МРС.

1 ОГЛЯД ТА АНАЛІЗ ОСОБЛИВОСТЕЙ ПРОЦЕСУ ПОШУКУ ОПТИМАЛЬНОГО ШЛЯХУ

Оскільки карта місцевості фактично являє собою сукупність безлічі секторів, з'єднаних один з одним, то для ефективності вирішення поставленого завдання можна уявити наявну карту як спеціальний орієнтований граф (рис. 1.1).

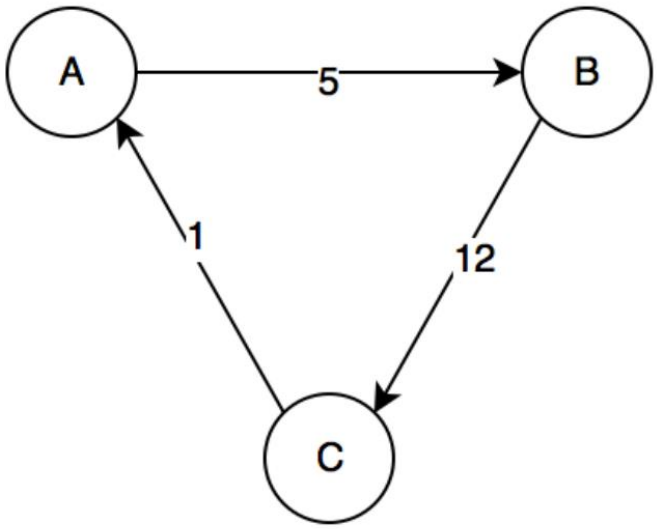


Рисунок 1.1 – Загальний вигляд зваженого графу

Тоді сектори карти - це вершини графа, а ребра - це можливість переміщення між сусідніми секторами [6-9]. Орієнтований граф G в свою чергу, це впорядкована пара множин:

$$G = (V, A), \tag{1.1}$$

де V - непорожня множина вершин або вузлів, A - множина (упорядкованих) пар різних вершин, які носять назву дуги або орієнтовані ребра [6-9].

Дуга - це впорядкована пара вершин U, визначається як:

$$U = (v, w), \tag{1.2}$$

де v - початкова вершина, w - кінцева вершина.

Можна сказати, що дуга, описана формулою 1.2, веде від початкової вершини v до кінцевої вершини w [6-9]. Зважений граф (рис. 1.1) - це граф, кожному ребру якого поставлено у відповідність деяке значення, зване вагою ребра [6-9].

Карту, що передається з центру керування, зручно уявити саме як зважений граф (див. рис. 1.1), який описує навколишнє середовище на карті, де вага ребра графа позначає вартість переміщення між інцидентними вершинами. У цьому випадку пошук траєкторії шляху з нижчою сумарною вагою дасть найбільш оптимальний шлях.

Вершини графа - це сектори, а ребра - можливість переміщення в сусідні сектори. Обов'язкова умова - до списку вершин цього графа мають входити точки початку та кінця руху. Після побудови графу допустимих шляхів пошук оптимального шляху на карті зводиться до завдання пошуку шляху у зваженому графі [6-9].

Однак, у процесі переміщення МРС побудованим шляхом може надійти на сенсори нова інформація про стан місцевості, а отже, зміняться значення в вершинах графа та, відповідно і ваги ребер, деякі ребра можуть зникнути, а можуть бути утворені нові. Отже, необхідно вміти перебудовувати шлях у випадку, якщо виявлені МРС перемінні його зачіпають.

1.1 Постановка завдання побудови оптимального шляху для МРС

1.1.1 Постановка завдання руху

Задаємо зважений орієнтований граф G типу [6-9]:

$$G = (V, E), \tag{1.3}$$

де V - множина секторів або вершин графа; E - множина шляхів між суміжними секторами - вершинами графа.

Вартість переміщення між суміжними вершинами v і w відповідає вказаній формулі:

$$0 \le c(v,w) \le n, \tag{1.4}$$

де $c(v,w)$ - вартість переміщення між вершинами v та w; n - максимально допустиме значення, за якого існує ребро графа між вершинами v та w

Початкова вершина f, приймається такою що:

$$f \in V. \tag{1.5}$$

Кінцева вершина t, приймається такою що:

$$t \in E. \tag{1.6}$$

Потрібно побудувати оптимальну траєкторію шляху K(f, t).

1.1.2 Вхідні дані

Карта із необхідними даними про ділянку місцевості представляє собою текстовий файл із розширенням типу «.map» [10]. Усередині файлу повинна міститись інформація про обмежену ділянку місцевості, представлену в цифровому вигляді в наступному форматі запису: карта містить заголовок та перемінні, а крім того, може містити необхідні коментарі.

Структура заголовка наступна: TM XXX.XX YYYY.YYY, dX dY, де T – перемінна яка вказує тип даних (I - integer, B - byte, D – double, F - float), M – перемінна для одиниць вимірювання координат положення (G - географічні

координати, M - метри, K - кілометри, C - сантиметри), XX.XX – перемінна для x-координати опорної точки карти. Опорною точкою називається точка в першому шарі комірок, відносно якої виконуються обрахунки координати лівого нижнього кута першого осередку для кожного наступного шару карти. YYYY.YYY – перемінна для y-координати опорної точки карти (координати можуть задаватися у абсолютних географічних або відносних геодезичних координатах), dX – перемінна, що задає крок сітки за координатою x (або горизонтальний розмір комірки), dY – перемінна, що задає крок сітки за координатою y (або вертикальний розмір комірки). Може набувати від'ємних значень для побудови карт за принципом руху «зверху – вниз».

Структура перемінних, що відповідають за опис рядів комірок карти наступна - кожен рядок повинен складатись із пари значень OFFSET LENGTH, де OFFSET - зміщення поточного ряду клітинок відносно положення x-координати опорної точки. Вимірюється в кроках сітки (або кількості комірок). Може приймати як позитивні значення, що означає зміщення праворуч, так і від'ємні значення (відповідно, зміщення ліворуч). Для першого ряду даний параметр зазвичай приймає значення 0, але може також набувати і ненульове значення. Значення даного параметру повинне знаходитись в допустимому діапазоні того типу перемінних, який задано в заголовку карти, наприклад, для типу перемінних B (byte) - у діапазоні значень від мінус 128 до +127. LENGTH – відповідно, довжина поточного ряду в комірках. Завжди приймає позитивне ціле число для типу даних byte у діапазоні від мінус 128 до +127, для інших типів перемінних - у діапазоні типу даних int.

У процесі руху за встановленим маршрутом, MPC може отримати оновлені дані з сенсорів. Такі дані можуть мати вплив на траєкторію подальшого просування, оскільки можуть заблокувати поточну побудовану траєкторію руху або, навпаки, відкрити більш оптимальний шлях. Буде задаватись у вигляді трійок значень: «номер ряду», «номер квадрата в ряду (або величина зміщення в ряду)», нове значення.

Початкове та кінцеве положення МРС буде задаватись у вигляді пар значень: «номер ряду», «номер квадрата в ряду (або величина зміщення в ряду)». Самі значення будуть задані у вигляді послідовності байтових величин - по одному байту на номер ряду та на номер квадрата (зміщення) [10].

1.1.3 Перелік встановлених вимог

1 Підготовка модуля до руху:

1.1 Створення модуля зовнішнім компонентом;

1.2 Вибір алгоритму пошуку траєкторії;

1.3 Вибір способу підрахунку вартості переміщення:

- вибір способу підрахунку вартості переміщення між секторами;

- вибір способу підрахунку евристичної функції.

1.4 Отримання вхідних даних:

1.4.1 Отримання даних про карту:

- створення зваженого орієнтованого графа

- встановлення роздільника даних від мета-даних.

1.4.2 Отримання даних про початкову точку шляху:

- створення вершини графа початкової точки шляху;

- встановлення вершини початкової точки шляху на графі.

1.4.3 Отримання даних про кінцеву точку шляху:

- створення вершини кінцевої точки шляху на графі;

- встановлення вершини кінцевої точки шляху на графі.

2 Пошук шляху.

2.1 Ініціалізація алгоритму пошуку:

- створення об'єкта обраного в п.1.2 даного алгоритму пошуку для виконання пошуку шляху;

- встановлення початкової точки шляху відповідно до вершини графа згідно із п. 2.2;

- встановлення кінцевої точки шляху відповідно до вершини згідно із п.2.3;

- ініціалізація перемінних алгоритму.

2.2 Виконання пошуку оптимального шляху.

2.3 Обробка результату пошуку оптимального шляху:

- якщо сталася помилка:

    - створення структури перемінних для передачі статусу помилки;

    - заповнення структури статусом та відповідним кодом помилки.

- якщо траєкторія шляху побудована:

    - створення структури перемінних для зберігання траєкторії шляху;

    - заповнення структури перемінними про побудовану траєкторію шляху:

        - заповнення виконується від початкової вершини до кінцевої;

        - кожна вершина описана парою «ряд, зміщення в ряду»;

2.4 Передання результату пошуку оптимального шляху модулю керування.

3. Якщо змінилася проблема пошуку.

3.1. Отримання змінених вхідних перемінних:

- отримання змін про місцезнаходження МРС;

- передача змін про місцезнаходження МРС в алгоритм пошуку;

- отримання змін про стан частин карти;

- передавання змін про кожен сектор карти в алгоритм.

3.2 Обробка змін у перемінних:

- зміна вершини графа, що відповідає позиції МРС;

- зміна даних про кожну вершину графа, отриману в пункті передавання змін про кожен сектор карти в алгоритм.

3.3. Перепланування маршруту руху МРС:

- якщо сталася помилка:

    - створення структури перемінних для передачі статусу помилки;

    - заповнення необхідної структури статусом і кодом помилки.

- якщо шлях вже побудовано:

    - створення структури перемінних для зберігання траєкторії шляху;

- заповнення структури перемінних про побудовану траєкторію шляху:

- заповнення виконується від поточного місця розташування МРС до кінцевого положення;

- кожна вершина представлена парою «ряд, зміщення в ряду».

3.4. Передача нової траєкторії маршруту модулю керування [6-9].

1.2 Огляд та аналіз існуючих алгоритмів пошуку

На сучасному етапі існує декілька алгоритмів пошуку, які не просто вирішують задачу побудови шляху у зваженому орієнтованому графі, а й розроблені спеціально для використання саме на роботизованих платформах [11].

1.2.1 Алгоритм A*

Алгоритм A* - це вид алгоритму пошуку за першим найкращим збігом, який визначає у зваженому графі маршрут найменшої вартості від початкової вершини до обраної кінцевої [12]. Даний вид алгоритму пошуку є найбільш популярним рішенням.

Алгоритм у процесі своєї роботи розраховує спеціальну функцію, яка носить назву евристичної. Завдання даної функції - давати наближене значення шляху, що залишився із конкретної вершини до кінцевої вершини графа. Ґрунтуючись на результатах евристичної функції алгоритм пошуку вибудовує порядок обходу вершин для вирішення поставленої задачі [4].

Поведінка алгоритму під час пошуку спирається на те, як реалізована евристична функція. Логічно, що не можна просто використати будь-яку функцію, яка найбільше сподобалась, і почати її використовувати, а отже вибір функції буде повністю залежати від постановки завдання. Оскільки алгоритм пошуку A* [12] використовується здебільшого для побудови переміщення по

поверхні, покритій координатною сіткою, опишемо основні функції, які будуть використовуватись в переважній більшості випадків:

- якщо переміщення МРС можливе в чотирьох напрямках, то варто вибрати манхеттенську відстань [13]:

$$h(v) = |v.x - w.x| + |v.y - w.y|. \quad (1.7)$$

- відстань Чебишева [14, 15] використовується, у випадку коли рух можливий ще й по діагоналі, проте штрафів за таку поведінку не передбачається:

$$h(v) = \max\left(|v.x - w.x|, |v.y - w.y|\right). \quad (1.8)$$

- якщо немає обмежень за напрямками руху МРС, то можна використовувати звичайну евклідову відстань по прямій [13-15].

$$h(v) = \sqrt{\left(v.x - w.x\right)^2 + \left(v.y - w.y\right)^2}. \quad (1.9)$$

У процесі роботи алгоритму пошуку для вершин розраховується наступна функція:

$$f(v) = g(v) + h(v), \quad (1.10)$$

де $g(v)$ - найменша вартість шляху до вершини v зі стартової вершини графа; $h(v)$ - евристичне наближення вартості шляху від вершини v до кінцевої вершини графа.

Насправді, функція $f(v)$ - оціночна довжина траєкторії шляху до встановленої мети, яка складається з тієї відстані, яку вже вдалося пройти $g(v)$ і

відстані, що залишилася, яка оцінюється за допомогою використання евристичної функції $h(v)$. Виходячи з цього, чим менше значення $f(v)$, тим швидше алгоритм пошуку добереться до вершини v, тому що саме через неї, виходить переміститись до кінцевої вершини найшвидше. Відкриті таким чином вершини зберігаються в черзі з пріоритетом за значенням $f(v)$. Серед усіх маршрутів, що потенційно що ведуть до вирішення поставленої задачі, алгоритм A* переглядає спочатку ті, які завдяки наявній інформації про вершини, а насправді завдяки результату визначення евристичної функції конкретної вершини, у даний момент часу є найбільш перспективними.

Проблема даного виду алгоритму в тому, що потрібно постійно зберігати нерозглянуті вершини у списку з пріоритетом, а розглянуті в іншому «закритому» списку. Крім того, кожна вершина повинна мати посилання на батьківську вершину, щоб потім можна було відновити побудовану траєкторію маршруту. Уже за помірної кількості вершин у встановленому графі, алгоритм пошуку починає активно споживати доступну пам'ять [13-15].

1.2.2 Алгоритм пошуку Iterative deepening A*

Вид алгоритму пошуку IDA* призначений для пошуку траєкторію шляху в зваженому орієнтованому графі і здатний знайти найкоротшу траєкторію шляху між встановленою початковою вершиною та множиною кінцевих вершин [16]. Алгоритм пошуку є варіантом пошуку в глибину з ітеративним поглибленням [1-5]. Основна перевага даного виду алгоритму - запозичення ідеї використання евристичної функції за аналогією з видом алгоритму A*. Результат евристичної функції використовується алгоритмом пошуку для того, щоб оцінити відстань, що залишилася до кінцевої вершини.

На відміну від звичайного пошуку в глибину, даний алгоритм концентрується на розгляді найбільш перспективних з погляду оцінюваної відстані вершин і тому не обходить різні підграфи на одну й ту саму глибину. На відміну від алгоритму A* даний алгоритм пошуку не використовує техніку

динамічного програмування (тобто загально відомий спосіб вирішення складних завдань шляхом розбиття їх на більш простіші підзадачі) і тому розглядає одні й ті самі вершини безліч разів.

Звичайний пошук у глибину з ітеративним поглибленням використовує заздалегідь визначене значення як обмеження пошуку за глибиною. Алгоритм IDA* для цієї мети використовує значення наступної функції [16]:

$$f(n) = g(n) + h(n), \tag{1.11}$$

де $g(n)$ - вартість траєкторії шляху від початкової вершини до вершини n графа; $h(n)$ - значення евристичної функції з оціночною вартістю шляху від вершини n до кінцевої вершини графа.

Алгоритм гарантує знаходження найкоротшого шляху, якщо евристична функція є допустимою [16], тобто:

$$h(n) \le h^*(n), \tag{1.12}$$

де $h^*(n)$ - істинна вартість найкоротшого шляху з вершини n до кінцевої вершини (так звана «ідеальна евристика») для всіх вершин.

Основна перевага алгоритму виду IDA* [16] проявляється, коли є жорсткі обмеження на кількість пам'яті, що може бути спожита. Оскільки даний алгоритм не запам'ятовує інших вершин, крім тих, що розглянуті на поточній ітерації, то і кількість пам'яті, що споживається, буде лінійно залежати від довжини кінцевого рішення.

1.2.3 Алгоритм пошуку D*

Алгоритм D* - вид алгоритму, що призначений для вирішення проблем пошуку траєкторії шляху в середовищах, де МРС необхідно переміщуватись до

заданої цільової точки в невідомому навколишньому оточенні [17]. Даний вид алгоритму робить припущення про невідому частину навколишнього середовища (наприклад, про відсутність перешкод) і знаходить найкоротший шлях від свого поточного місця перебування до кінцевої точки, спираючись на ці припущення. Потім МРС виконує рух побудованим шляхом. Щойно він якимось способом отримує нову інформацію про навколишній простір, у якому перебуває (дані з центру контролю, власні давачі, інші МРС тощо), то він бере її до відома та, якщо необхідно, виконує перебудову найкоротшого шляху до встановленої кінцевої вершини зі своєї поточної позиції. Даний процес повторюється доти, доки МРС не досягне кінцевої точки або поки не з'ясує, що задана точка більше не може бути досягнутою [2].

Коли переміщення МРС виконується в невідомому навколишньому середовищі, то нові дані можуть з'являтися дуже часто, тому процес перерахунку траєкторії шляху повинен відбуватися досить швидко. Це досягається за рахунок використання інкрементного підходу – алгоритм пошуку D* інтенсивно використовує історію попередніх пошуків [2].

Як і алгоритм виду A*, алгоритм виду D* використовує список вершин, які підлягають розгляду. Основна робоча ідея алгоритму пошуку D* заснована на використанні спеціалізованих станів вершин [18]:

- New – вершину графа ще не розглядали;

- Open - вершина графа перебуває у списку вершин;

- Closed – вершина графа вже не перебуває у списку вершин;

- Raise - вказує, що вартість вершини графа збільшилася порівняно з тим, коли вона востаннє перебувала у списку вершин;

- Lower - вказує, що вартість вершини графа зменшилася порівняно з тим, коли вона востаннє перебувала у списку вершин.

Алгоритм пошуку D* [17] ітеративно витягує зі списку вершину графа і обчислює її значення. Потім поширює зміни від цієї вершини графа на всі сусідні вершини графа і поміщає їх у список вершин. Даний процес називається

розкриттям або розширенням вершини графу. На відміну від класичного алгоритму пошуку A* [18], який слідує за побудованим шляхом від початкової вершини до кінцевої вершини, алгоритм пошуку D* [17] починає пошук від кінцевої вершини графу до початкової вершини. Кожна розкрита вершина графу має посилання на наступну вершину, що веде до встановленої мети і знає точну вартість шляху до неї.

Коли чергова розкрита вершина графа - початкова, алгоритм пошуку закінчує свою роботу і шлях до кінцевої вершини графу може бути просто побудований за посиланнями між вершинами [2].

Коли приходять оновлені дані, які впливають на побудовану траєкторію шляху, то вершини з вагами, які змінилися, знову поміщають до списку вершин, але тепер вони вже позначені як RAISE. Однак, до того як вершина графа з RAISE збільшить свою вартість, алгоритм пошуку перевіряє всі сусідні вершини графу на предмет того, чи зможуть вони зменшити вартість даної RAISE вершини. Якщо ні, то стан RIASE передається всім вершинам графа, які містять посилання на вершину, що змінилася. Потім дані вершини перераховуються і RAISE стан передається далі, формуючи своєрідну хвилю змін. Коли вершина з RAISE може зменшити свою вартість, то посилання знову оновлюється, передаючи сусіднім вершинам уже стан LOWER [2].

З таким підходом велика кількість вершин графу не зачіпається при зміні станів взагалі і, таким чином, алгоритм пошуку обробляє тільки вершини графу, на які вплинули виявлені під час переміщення зміни.

1.2.4 Алгоритм пошуку Lifelong Planning A*

Алгоритм LPA* - вид алгоритму пошуку найкоротшого шляху у зваженому орієнтованому графі, де структура графа невідома заздалегідь або постійно змінюється [19].

Основою даного алгоритму пошуку є обчислення та використання функції $g(s)$ - найменшої вартості шляху з початкової вершини графу у вершину s.

Значення даної функції для алгоритму пошуку LPA* дуже схоже на значення аналогічної функції з алгоритму пошуку A*, за винятком того, що тепер нас цікавлять тільки $g(s)$ - значення відомих вершин на даній ітерації [19].

Для кожної вершини алгоритм визначає дві множини суміжних, позначених наступними формулами:

$$Succ(s) \subseteq V, \tag{1.13}$$

де $Succ(s)$ - множина вершин графу, що виходять із вершини s.

$$Pred(s) \subseteq V, \tag{1.14}$$

де $Pred(s)$ - множина вершин графу, що входять у вершину s.

Наступна функція використовується даним алгоритмом пошуку для підрахунку ваги ребра (s, s'), тобто повертає його фактичну вартість:

$$0 \le c(s, s') \le +\infty \tag{1.15}$$

При цьому ребро буде рівним $+\infty$ тоді і тільки тоді, коли насправді ребро не існує взагалі.

Крім описаних вище значень і функцій, даний алгоритм пошуку вводить нові визначення: rhs -значення та спеціальну функцію $key(s)$. rhs-значення - це така функція $rhs(s)$, яка повертає потенційну мінімальну відстань від початкової вершини s до вершини згідно з наступним рівнянням [19]:

$$rhs(s) = \begin{cases} 0, & \text{s - початкова вершина} \\ \min_{s' \in Pred(s)}(g(s') + c(s, s')), & \text{інакше} \end{cases} \tag{1.16}$$

Як видно з формули 1.16, rhs-значення використовує мінімальне значення з мінімальних відстаней від початкової вершини графу до вершин, які входять у дану вершину s. Таке значення буде нам давати оцінку відстані від початкової вершини графу до вершини s з урахуванням «кроку вперед».

Функція $key(s)$ носить назву ключ вершини. Ключ обчислюється наступним чином, що повертає вектор із двох значень $k_1(s), k_2(s)$:

$$key(s) = \left[k_1(s), k_2(s)\right], \tag{1.17}$$

де $k_1(s) = \min(g(s), rhs(s)) + h(s,t)$, $k_2(s) = \min(g(s), rhs(s))$.

Описане у формулі 1.16 значення відіграє дуже важливу роль в алгоритмі пошуку LPA*. Разом з іншим значенням, $g(s)$, воно використовується в ідеї насичення вершин графа, яка допомагає ефективніше проводити перерахунок шляху [20].

Вершина s носить назву насичена у випадку, якщо:

$$g(s) = rhs(s). \tag{1.18}$$

Вершина s носить назву перенасичена у випадку, якщо:

$$g(s) > rhs(s). \tag{1.19}$$

Вершина s носить назву ненасичена у випадку, якщо:

$$g(s) < rhs(s). \tag{1.20}$$

Евристична функція $h(s, s')$ повинна бути додатною і виконувати

нерівність трикутника, тобто відповідати вимогам, вказаним в наступних формулах [19, 20]:

$$f(s,t)=0; \tag{1.21}$$

$$h(s,t)\le c(s,s')+h(s',t), \tag{1.22}$$

де $s\in V, s'\in Succ(s)$.

Якщо наприкінці пошуку шляху $g(t)$ дорівнює $+\infty$, то вважається, що алгоритм пошуку у цілому зазнав невдачі, тобто не зміг знайти траєкторію шляху від початкової вершини графу до кінцевої. Але, оскільки алгоритм пошуку розроблено для пошуку в графі з змінними вагами ребер, то під час його реалізації зазвичай додають спеціальний механізм очікування змін у графі, тому що після чергових змін траєкторія шляху може знайтися.

Функція ініціалізації вихідного графа встановлює для всіх вершин крім стартової значення $g(s)$ та $rhs(s)$ рівними безкінечності. Очевидно, що для стартової вершини її rhs-значення є рівним нулю, і що мінімальна відстань від стартової вершини до самої себе повинна також дорівнювати нулю [19, 20]. Повернуті функцією (1.17) значення сортуються в лексографічному порядку, тобто спочатку сортується відносно $k_1(s)$, потім відносно $k_2(s)$.

Алгоритм пошуку декілька разів обчислює значення $g(s)$ у ненасичених вершин графу у неспадаючому порядку їхніх ключів. Такий перерахунок даного значення називається розширенням вершини графів. Алгоритм пошуку продовжує виконуватися доти, доки кінцева вершина графу не стане насиченою і на верху списку представлених до розгляду вершин не опиниться вершина з більшим значенням функції 1.17, ніж у кінцевої вершини.

Таким чином, алгоритм пошуку LPA* обчислює довжину найкоротшого шляху між двома вершинами графу, використовуючи при цьому дані з

попередніх ітерацій.

У гіршому випадку, точніше коли всі ребра навколо поточної вершини графу змінюють свою вагу, алгоритм пошуку відтворює, фактично, послідовні виклики алгоритму пошуку A* [19, 20].

1.2.5 Алгоритм пошуку D* Lite

Алгоритм D* Lite - вид алгоритму, що базується на алгоритмі пошуку LPA*, але крім того, використовує ідею обліку переміщення МРС на карті з алгоритмом пошуку D*. Даний алгоритм пошуку здатний визначати відстань між поточною вершиною графу, у якій, припустімо, рухається МРС, і кінцевою вершиною при кожній зміні графа в той час, як МРС рухається вздовж знайденого шляху [19].

Алгоритм пошуку покликаний вирішувати завдання оновлення значення функції $g(s)$ в процесі руху МРС найкоротшим шляхом у графі при надходженні нової інформації.

Оскільки під час руху найкоротшим шляхом шлях може тільки скорочуватися і відбувається зміна тільки стартової вершини, то алгоритм пошуку D* Lite використовує ідею з алгоритму пошуку LPA* [19,20].

По-перше, алгоритм пошуку змінює напрямок пошуку в графі. Тепер функція $g(s)$ зберігає мінімальну відому відстань від кінцевої вершини графа t до початкової f. Властивості залишаються такими ж, як в алгоритмі пошуку LPA*.

Евристична функція так само трохи змінюється. Тепер вона повинна бути додатною і обернено-стійкою для всіх вершин $s \in S, s' \in \Pr ed(s)$, тобто:

$$h(f, f) = 0, \tag{1.23}$$

де f - початкова вершина графу.

$$h(f,s) \le h(f,s') + c(s,s'). \quad (1.24)$$

Очевидно, що під час руху МРС початкова вершина графа f змінюється, тому дані властивості мають виконуватися для всіх вершин графа, що розглядається.

Додаткова умова закінчення роботи алгоритму пошуку також змінюється, тобто у випадку $g(f)$, що дорівнює +∞, шлях не знайдено на даній ітерації. Інакше шлях знайдено і МРС може переміщатисч по ньому.

Також слід вказати, що на етапі підготовки не потрібно ініціалізувати абсолютно всі вершини графу. Це важливо, тому що кількість вершин графу може бути дуже великою, і лише деякі з них будуть пройдені МРС у процесі виконання переміщення. Також, це дає можливість додавання або видалення ребер без втрати стійкості всіх підграфів даного графа [21].

Тепер, після первинного розрахунку шляху, ми будемо переміщувати МРС, тобто фактично стартову вершину графу таким чином, щоб нова вершина графу відповідала умові:

$$f = \min_{s' \in Succ(f)} (c(f,s') + g(s')). \quad (1.25)$$

Якщо після чергового виконання переміщення граф змінився, то для всіх орієнтованих ребер (v,w) зі зміненими вагами необхідно оновити вагу ребра (v,w) і перерахувати $rhs(u)$ та $g(u)$. Потім, для всіх вершин графу, які ще перебувають у списку ненасичених, потрібно оновити результат функції $key(s)$ (див. 1.17), після чого знову розрахувати шлях.

У такій інтерпретації алгоритм пошуку має заново обчислювати ключі вершин, на основі яких вони розташовані в пріоритетній черзі, щоразу, коли МРС помічає зміни у вазі ребер у процесі виконання переміщення. Поки ключі не будуть перераховані, вони не можуть задовольняти встановленим пріоритетам

у черзі, оскільки ґрунтувалися на значенні евристичної функції, яка була обчислена з урахуванням старого положення МРС. Однак, постійне упорядкування вершин графу у пріоритетній черзі дуже дорога операція, особливо за великої кількості повторень. Алгоритм пошуку D* Lite використовує підхід, що дає змогу уникати такого постійного впорядкування. Тому евристична функція (1.24) повинна бути додатною і задовольняти наступну умову [21]:

$$h(s,s') \le c^*(s,s'), \tag{1.26}$$

де $c^*(s,s')$ - вартість найкоротшого шляху з вершини графу $s \in Succ(s)$ до вершини графу $s' \in S$ та

$$h(s,s'') \le h\left(s,s'\right) + h\left(s',s''\right), \tag{1.27}$$

де $s',s'' \in Succ(s)$.

Це означає, що тепер евристична функція має підтримувати нерівність трикутника для всіх вершин графу, і, крім того, має виконуватися властивість (1.26), навіть якщо враховувати те, що вершини можуть бути не суміжними [19].

Після того, як МРС перемістився з вершини s до вершини наприклад s' і там виявив зміни у вазі ребер, перший елемент у пріоритетній черзі може зменшиться максимум на величину $h(s,s')$, оскільки дане значення бере участь в обчисленні пріоритету вершини графу в черзі (1.17). Другий компонент у ключі не залежить від функції h і, отже, не змінюється.

Таким чином, для того щоб зберегти нижні межі, алгоритм пошуку D* Lite [19-21] повинен віднімати $h(s,s')$ з першої компоненти ключів усіх вершин у пріоритетній черзі. Однак, оскільки $h(s,s')$ однакова для всіх вершин графу у пріоритетній черги, то порядок від такого віднімання не зміниться. А коли обраховуються нові пріоритети, то перша компонента ключа буде знижена на те

ж саме значення $h(s,s')$, порівняно з тими ж компонентами вершин графу, що вже що перебувають у черзі. Тому, необхідно додавати величину $h(s,s')$ до першої компоненті ключа вершини графу, що перераховується, щоразу, як вага ребер у графі змінюється. Якщо МРС знову рухається і знову виявляє зміни в графі, то необхідно знову виконувати додавання величини $h(s,s')$. Ці зміни накопичуються алгоритмом пошуку у перемінній K_m, а отже, значення перемінної K_m потрібно додавати до перших компонентів ключів щоразу, коли пріоритети обраховуються заново. Тоді, після того, як МРС переміститься, порядок вершин графу у пріоритетній черзі не зміниться, а отже й елементи в черзі не потрібно впорядковувати заново [21].

Однак, на момент зміни, черга містить набір вершин графу зі старими значеннями пріоритетів, а точніше старими значеннями перших компонент ключів. Тому, після того як із черги вилучається вершина графу з найменшим значенням ключа і запам'ятовується в K_{old}, то ключ перераховується - щоб дізнатися який пріоритет насправді повинна була мати вершина графу. Тому якщо старе значення менше за нове, то вершина графу поміщається назад у чергу, але вже з новим значенням ключа. Таким чином, буде гарантовано, що пріоритети всіх вершин графу у пріоритетній черзі - це дійсно нижні межі з урахуванням переміщення МРС, а значить і з урахуванням зміни модифікатора K_m. Якщо старе значення більше, приймаємо, що ключ не змінився, оскільки K_{old} уже містив найменше значення, повернуте раніше викликаною функцією підрахунку ключа. У даному випадку, алгоритм пошуку розкриває вершину графу.

За допомогою введення ключового модифікатора K_m, і відкладеного оновлення ключів вершин графів вдалося прибрати з ітерації алгоритму пошуку безліч операцій, які витрачалися на оновлення черги [19].

1.3 Постановка завдань для розробки алгоритму переміщення роботу

Для досягнення поставленої мети кваліфікаційної роботи бакалавра необхідно вирішити наступні завдання:

- розглянути задачу побудови траєкторії шляху MPC;
- сформулювати необхідні вимоги до системи;
- виконати аналіз алгоритмів пошуку оптимального шляху;
- розробити прототип програмної реалізації алгоритмів пошуку оптимального шляху;
- розробити програмний модуль на мові Java [22-24];
- виконати тестування створеного модулю та алгоритмів.

1.4 Висновки до першого розділу

Проведено огляд та аналіз існуючих алгоритмів пошуку оптимальної траєкторії руху робота на основі графів.

Виконано постановку завдань для розробки алгоритму переміщення мобільного роботу.

2 РОЗРОБКА ПРОЄКТУ АЛГОРИТМУ ПЕРЕМІЩЕННЯ РОБОТУ

2.1 Обґрунтування вибору алгоритму пошуку

Нижче представимо порівняльну таблицю розглянутих вище алгоритмів, у якій зазначено наступні параметри: використання інкрементного та евристичного підходу під час розв'язання поставленої задачі, відносне споживання пам'яті, відносний час виконання та здатність враховувати мінливе положення МРС [12-21].

Таблиця 2.1 - Порівняння алгоритмів пошуку шляху МРС [12-21]

Алгоритм	Інкрементний пошук	Евристичний пошук	Час на виконання	Використання пам’яті	Врахування положення МРС
A*	-	+	задовільно	погане	-
IDA*	-	+	погано	добре	-
D*	+	-	задовільно	задовільно	+
LPA*	+	+	добре	задовільно	-
D* Lite	+	+	добре	задовільно	+

Для реалізації в МРС ми спинились на застосуванні двох алгоритмів пошуку, а саме: D* Lite та IDA*.

Алгоритм пошуку D* Lite ми обрали тому, що він дозволяє поєднувати в собі ідеї інкрементного та евристичного пошуків і чудово справляється із завданням переміщення МРС в невідомому навколишньому середовищі [25]. Даний алгоритм ми приймаємо за базовий для використання в МРС, що проєктується і, якщо будуть дозволяти наявні ресурси, будемо застосовувати алгоритм пошуку для розв'язання задачі переміщення МРС.

Алгоритм пошуку D* Lite прекрасно підходить для розв'язання задачі пошуку шляху з декількох причин. Будучи реалізованим він матиме наступні переваги:

- пошук оптимальної траєкторії шляху у зваженому орієнтованому графі [6-9];

- здатність ефективно перепланувати траєкторію шляху у разі зміни графа;

- використовувати відкладену перебудову пріоритетної черги, для зменшення кількість необхідних операцій із вершинами;

- розглядати лише ті вершини, що потенційно ведуть до кінцевої мети [21];

- не потребує збереження траєкторії шляху в процесі побудови;

- не потребує попередньої ініціалізації всіх вершин обраного графа;

- виконує зміну евристичної функції для зміни умови оптимальності траєкторії шляху.

На рисунку 2.2 представлено ініціалізаційний метод, а також методи, що відповідають за обчислення ключа вершини та розкриття вершини графа [6-9].

```
1: function CalcKey(s)
2:     return [min(g(s), rhs(s)) + h(f, s) + K_m, min(g(s), rhs(s))]
3: end function

4: function Initialize()
5:     U = ∅
6:     K_m = 0
7:     for s ∈ V do
8:         rhs(s) = g(s) = +∞
9:     end for
10:    rhs(t) = 0
11:    U.insert(t, CalcKey(t))
12: end function

13: function UpdateVertex(u)
14:    if u ≠ t then
15:        rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'))
16:    end if
17:    if u ∈ U then
18:        U.remove(u)
19:    end if
20:    if g(u) ≠ rhs(u) then
21:        U.insert(u, CalcKey(u))
22:    end if
23: end function
```

Рисунок 2.2 – Загальний вигляд псевдокоду алгоритму пошуку D* Lite [21]

На рисунку 2.3 представлено основний цикл алгоритму пошуку, де з пріоритетної черги ітераційно витягують вершини графа, обчислюють їхні значення, розкривають сусідні вершини графа [6-9].

```
24: function COMPUTESHORTESTPATH()
25:     while U.topKey() < calcKey(f) or rhs(f) ≠ g(f) do
26:         K_old = U.topKey()
27:         u = U.pop()
28:         if K_old < CalcKey(u) then
29:             U.insert(u, calcKey(u))
30:         end if
31:         if g(u) > rhs(u) then
32:             g(u) = rhs(u)
33:             for s ∈ Pred(u) do
34:                 updateVertex(s)
35:             end for
36:         else
37:             g(u) = +∞
38:             for s ∈ Pred(u) ∪ u do
39:                 updateVertex(s)
40:             end for
41:         end if
42:     end while
43: end function
```

Рисунок 2.3 – Псевдокод основного циклу алгоритму пошуку D* Lite [21]

На рисунку 2.4 представлено метод, що запускає пошук, а також дії, необхідні до виконання при виявленні змін у процесі проходження по побудованим раніше шляхом.

Ще одним алгоритмом пошуку для реалізації процесу переміщення МРС було обрано алгоритм пошуку IDA*. Хоча алгоритм пошуку і не використовує новітні техніки інкрементного пошуку, а отже, не зможе показати ефективні результати в середовищі, що змінюється досить динамічно, проте алгоритм пошуку IDA* чудово підходить для МРС із обмеженим обсягом вбудованої

пам'яті і здатний за адекватний проміжок часу розв'язати задачу пошуку шляху в добре відомому навколишньому середовищі [16]. Передбачається, що, його використання в МРС буде застосовуватися для розв'язання задач локального пошуку на невеликих ділянках, при цьому забезпечуючи мінімальне споживання вбудованої пам'яті.

```
1: function MAIN()
2:     s_last = f
3:     initialize()
4:     computeShortestPath()
5:     while f ≠ t do
6:         f = s', s' = min_{s'∈Succ(f)}(c(f, s') + g(s'))
7:     end while
8:     if changed then
9:         K_m = K_m + h(s_last, f)
10:        s_last = f
11:        for u ∈ changed do
12:            updateVertex(u)
13:        end for
14:    end if
15: end function
```

Рисунок 2.4 – Загальний вигляд методу в псевдокоді алгоритму пошуку D* Lite [21]

Будучи реалізованим алгоритм пошуку IDA*, володітиме наступними перевагами:

- оперує здебільшого примітивними типами даних, ніж об'єктами;

- споживання вбудованої пам'яті лінійно залежне від довжини траєкторії шляху [19];

- не потребує попередньої ініціалізації всіх вершин графа;

- використовує евристичну функцію для оптимізації глибини пошуку;

- змінюючи евристичну функцію можна змінювати умову оптимальності траєкторії шляху.

На рисунку 2.5 наведено псевдокод алгоритму пошуку IDA*, що містить методи початкової ініціалізації, основного циклу та перевірки закінчення роботи алгоритму пошуку [16].

```
1: function isGoal(node)
2:     if node = goal then
3:         return true
4:     else
5:         return false
6:     end if
7: end function

8: function search(node, g, bound)
9:     f = g + h(node)
10:    if f > bound then
11:        return f
12:    end if
13:    if isGoal(node) then
14:        return FOUND
15:    end if
16:    min = +∞
17:    for s ∈ Succ(node) do
18:        t = search(s, g + cost(node, s), bound)
19:        if t = FOUND then
20:            return FOUND
21:        end if
22:        if t < min then
23:            min = t
24:        end if
25:    end for
26:    return min
27: end function

28: function main(root)
29:    bound = h(root)
30:    while true do
31:        t = search(root, 0, bound)
32:        if t = FOUND then
33:            return bound
34:        end if
35:        if t = +∞ then
36:            return NOT FOUND
37:        end if
38:        bound = t
39:    end while
40: end function
```

Рисунок 2.5 – Загальний вигляд псевдокоду алгоритму пошуку IDA* [16]

Як видно з рисунка 2.5, алгоритм пошуку IDA* використовує два основні методи під час своєї роботи і справді не має жодних структур, що зберігають розглянуті вершини графа.

2.2 Розробка проєкту загальної архітектури системи

Для зручності опису надалі ми будемо використовувати спеціальне слово «модуль», як позначення створюваного проєкту програмної реалізації алгоритмів пошуку, що розробляється. На рисунку 2.6 представлено діаграму прецедентів для створюваного модуля.

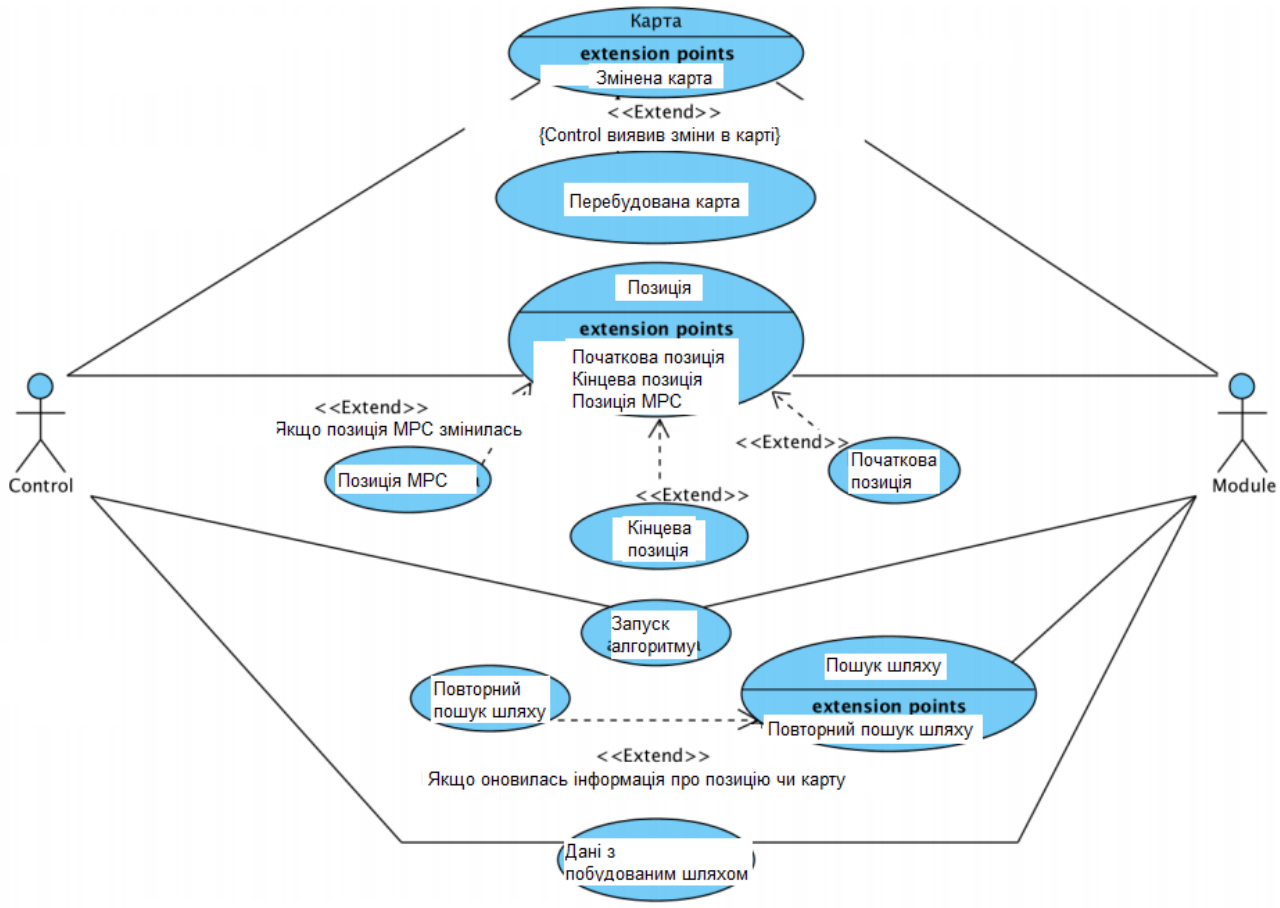


Рисунок 2.6 – Загальний вигляд діаграми прецедентів

Модуль має бути здатний приймати набір даних у встановленому форматі від зовнішнього керівного модуля або компонента, позначеного як Control (див. рис. 2.6).

Передбачається, що модуль МРС як прототип програмою реалізації алгоритмів пошуку може бути впроваджений у різні проєкти роботизованих платформ як частина програмної складової самого МРС, для ухвалення рішень щодо переміщення самостійно, як, наприклад, представлено на рис. 2.7, так і на боці віддалених компонентів керування або різних центрів керування, як, наприклад, представлено на рис. 2.8.

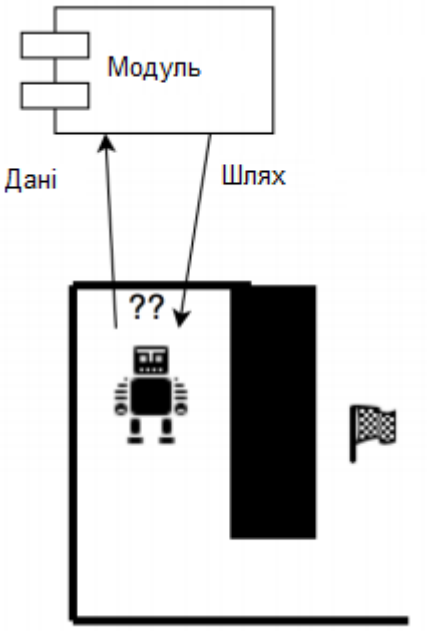


Рисунок 2.7 – Загальний вигляд процедури звернення до модулю з боку МРС

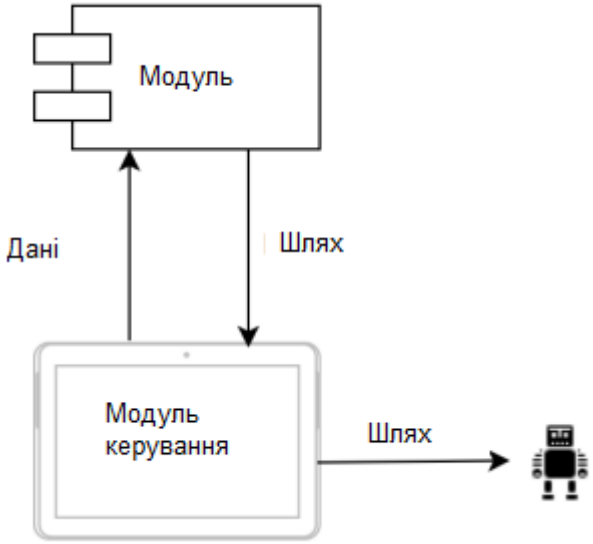


Рисунок 2.8 – Загальний вигляд процедури звернення до модуля на стороні модуля керування

Компонент керування є зовнішнім щодо модуля МРС, що розробляється і відповідає за створення модуля та керування його життєвим циклом. Саме від даного модулю керування будуть надходити дані про поточну карту в певному форматі, інформація про початкове розташування МРС на цій карті, про кінцеву позицію, про зміни, які надходять від давачів і з центру керування. Так само, цей компонент ініціалізує сам модуль і керує запуском всього алгоритму пошуку. Після того як алгоритм пошуку, реалізований у модулі керування закінчить свою роботу, модуль керування повертає результат з інформацією про побудовану траєкторію шляху, якщо такий буде знайдено, або буде генерувати повідомлення про помилку.

2.3 Розробка архітектури модуля керування МРС

Оскільки модуль керування для МРС проєктується для взаємодії з різноманітними зовнішніми системами, буде розумним застосувати так званий «фасадний» клас [26-29], для уніфікації звернення до алгоритмів пошуку оптимальної траєкторії шляху. Для цього створюємо клас PathStrategy.

Даний клас буде відповідати за прийом даних про реальну карту, початкову та кінцеву позиції на ній і різні зміни, які буде передавати компонент керування.

Для того щоб інформацію про реальну карту можна було представити у вигляді графа, необхідно на основі даних, що надійшли, створити необхідні вершини та ребра графа – цим буде займатись клас MapData. Даний клас буде уніфікований для роботи всіх реалізованих у модулі керування алгоритмів пошуку. Клас MapData буде відповідати за зберігання стану графа, знатиме все про вершини та ребра, що розташовані між ними, графа.

Оскільки у кожної вершини графа є певний набір станів, крім того, що вона має координати на реальній карті, то буде зручним виділити вершини графа в окремий клас. Це дасть змогу алгоритмам пошуку та класу MapData працювати

з ними, як із повноцінними об'єктами. Для вершин графа ми створюємо спеціальний клас MapCell. Схематично даний процес показано на рисунку 2.9.

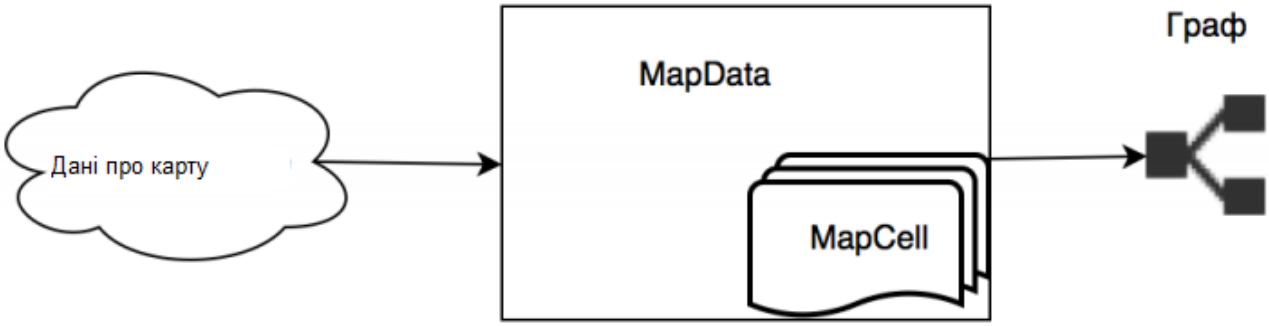


Рисунок 2.9 - Схема перетворення даних у граф

Оскільки модуль керування передбачає реалізацію кількох алгоритмів пошуку, та необхідно завчасно закласти можливості розширення функціональності, тобто додавання нових алгоритмів пошуку, ми прийняли рішення про створення окремого інтерфейсу IAlgorithm. Тоді клас PathStrategy зможе створювати різні реалізації даного інтерфейсу IAlgorithm, закладені в модулі керування, а надалі додавання чергового алгоритму пошуку зводитиметься лише до написання окремої реалізації. На рисунку 2.10 схематично відображено дану ідею [26-29].

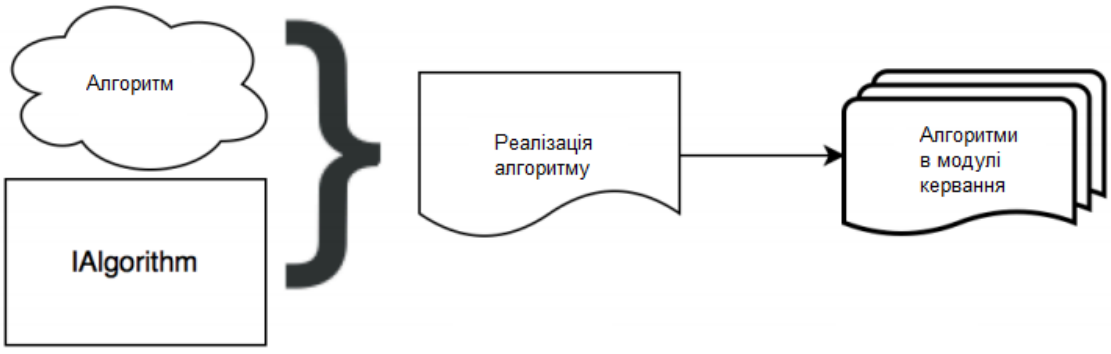


Рисунок 2.10 - Схема додавання нового алгоритму в модуль

При цьому необхідно надати можливість задавати ваги ребер графу і евристичну функцію окремо, оскільки умови переміщення МРС і, тим більше, вартість його переміщення по реальній карті можуть сильно відрізнятися на

різних системах. Для цього ми створюємо інтерфейс IDistanceCost, реалізації якого будуть використовуватися реалізаціями інтерфейсу IAlgorithm.

Нижче на рисунку 2.11 представлено узагальнену структуру модуля керування [26-29].

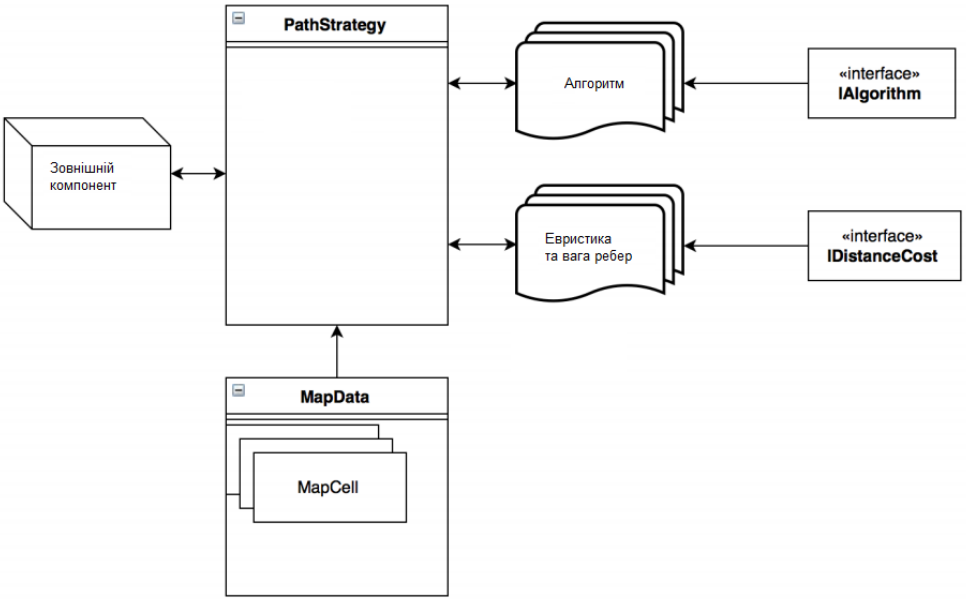


Рисунок 2.11 – Загальний вигляд структури модуля керування

2.4 Підбір основних класів [26-29]

На рисунку 2.12 представлено основні класи модуля та зв'язки між ними.

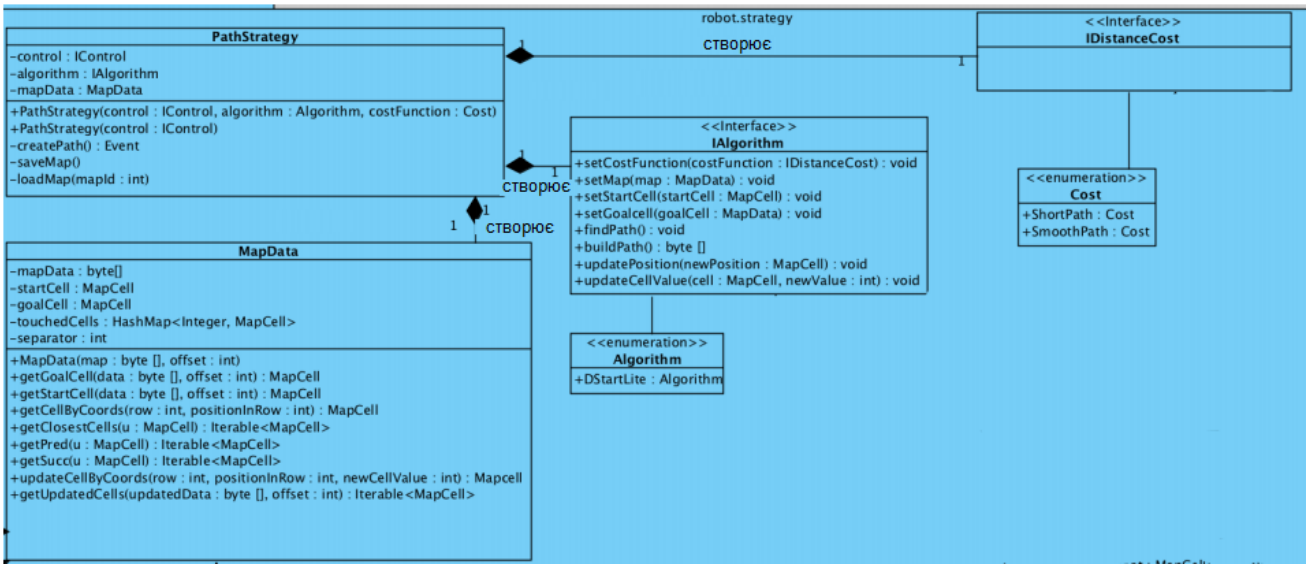


Рисунок 2.12 - Діаграма класів модуля керування

Класи MapData та MapCell на рисунку 2.13 представлено більш детально. Зазначено наявні залежності з колекціями мови Java [22-24].

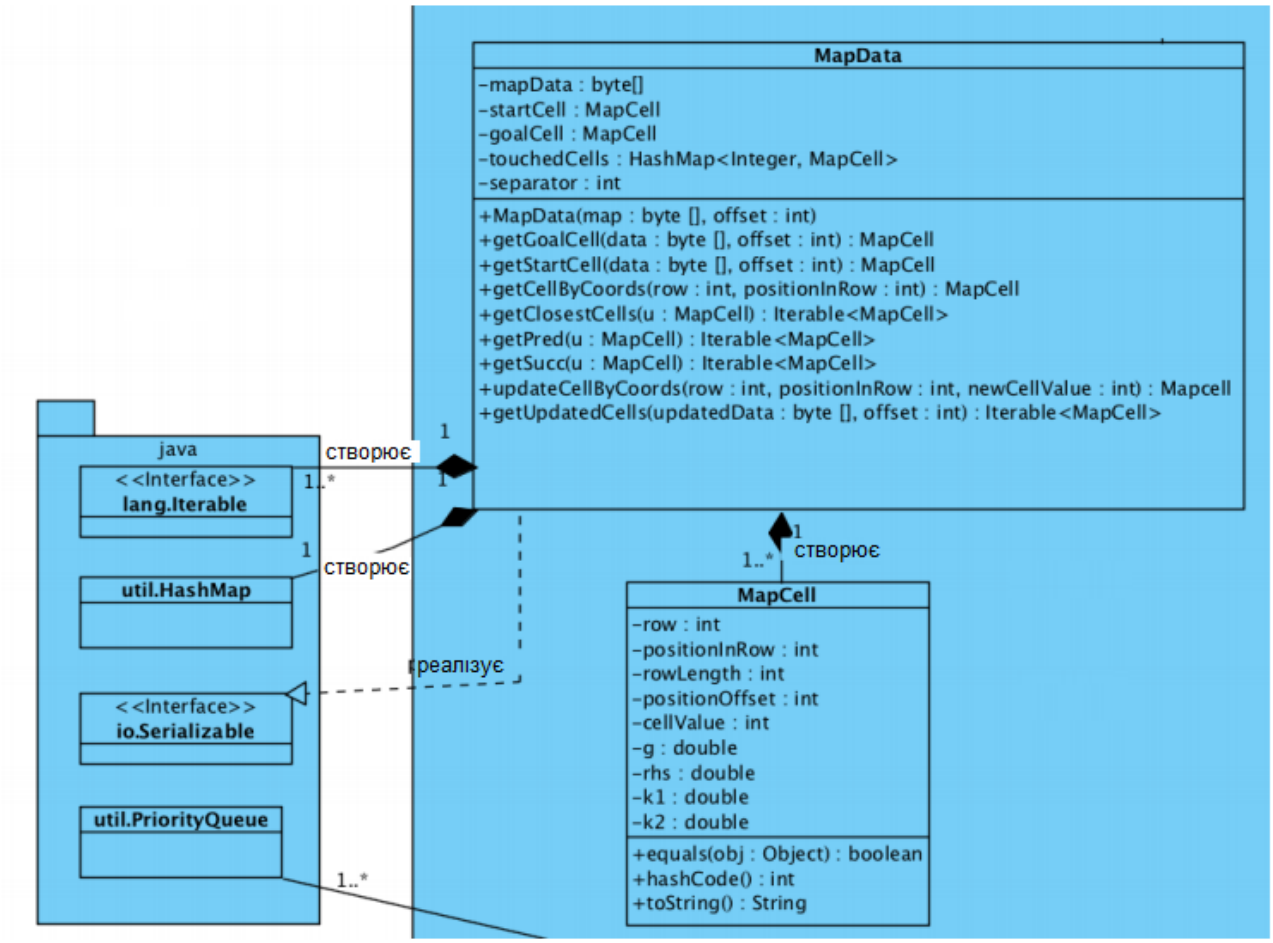


Рисунок 2.13 - Діаграма класів MapData та MapCell роботи з графом

На рисунку 2.14 показано класи, що реалізують алгоритми пошуку.

Перерахування Algorithm використовується зовнішньою системою для вибору алгоритму пошуку під час створення об'єкта модуля керування.

Всі розроблені коди програми наведені в додатках до пояснювальної записки.

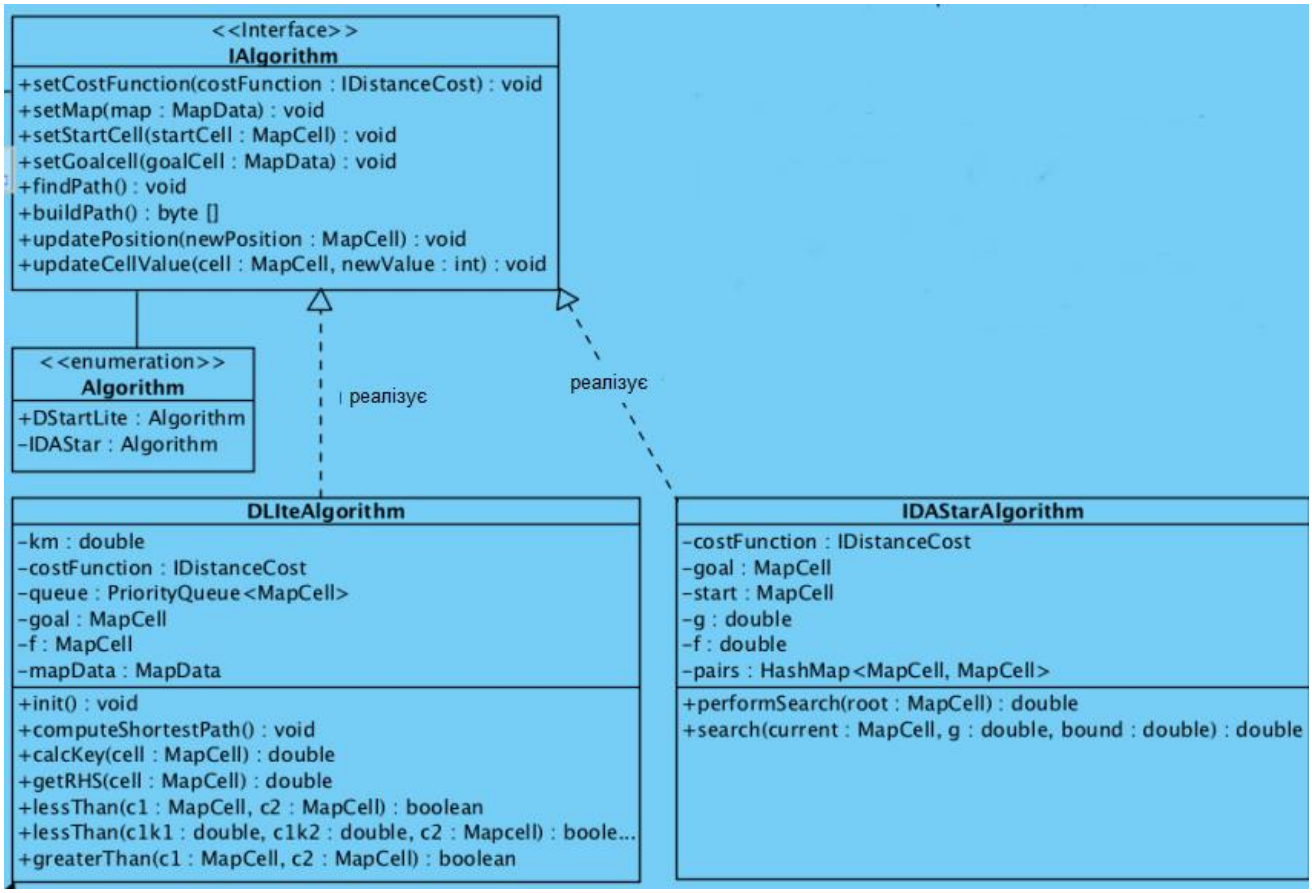


Рисунок 2.14 - Діаграма класів алгоритмів пошуку Algorithm

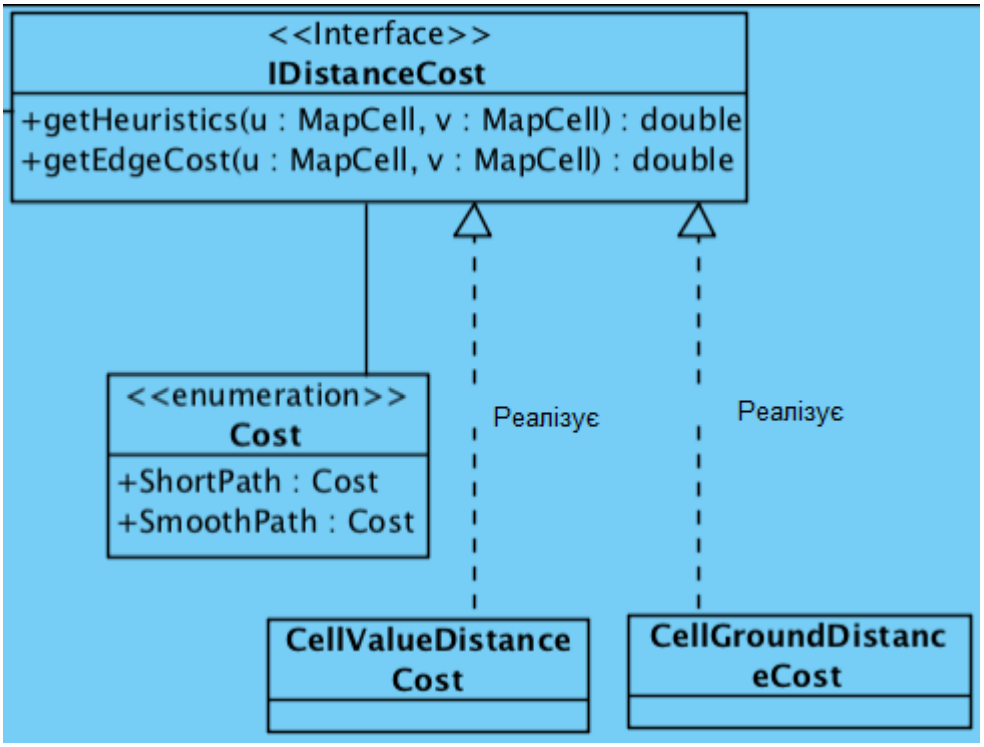


Рисунок 2.15 – Загальний вигляд діаграма класів вартості ваг і евристичної функції

Виходячи з різноманітності можливих платформ, де модуль керування може знайти використання, а також враховуючи особисті вподобання, було обрано наступну мову програмування - Java. Як середовище розробки використовується Eclipse IDE, оскільки Java забезпечує кросплатформеність, що дозволяє отримати додаткову гнучкість [22-24].

Мова і, відповідно, платформа Java мають чудову масштабованість, що допомагає легко створювати спеціальні додатки для пристроїв з обмеженими ресурсами, адаптуючи ПЗ спочатку написане для персональних комп'ютерів [22-24].

Модулю керування не потрібна будь-яка база даних, тому що всі завдання будуть вирішуватись локально і за досить короткий час. Передбачається, що модуль керування може бути вивантажено з пам'яті системи щойно він виконає поставлене завдання.

2.5 Висновки до другого розділу

Виконано необхідне обґрунтування вибору алгоритму пошуку, обрано за основу два алгоритми пошуку. Розроблено проєкт загальної архітектури всієї системи та загальну архітектуру модуля керування. Виконано підбір основних класів, всі розроблені коди програми наведені в додатках до пояснювальної записки.

3 РОЗРОБКА АЛГОРИТМУ ПЕРЕМІЩЕННЯ РОБОТУ

3.1 Робота алгоритму переміщення із графом

Клас MapCell інкапсулює в собі інформацію про одну конкретну клітинку на реальній карті. Для цього в класі передбачено низку полів, що зберігають необхідну інформацію про клітину [26-29]:

- positionInRow - зберігає порядковий номер (зсув) клітинки в ряду, індексація з 0;

- row - зберігає значення ряду, якому належить клітинка на реальній карті, індексація з 0;

- rowLength - зберігає загальну кількість клітинок у ряді;

- cellValue - значення клітинки, що визначає її прохідність МРС;

- positionOffset - зберігає зміщення ряду від початкової позиції розташування.

Наступні поля використовуються вже безпосередньо алгоритмами пошуку, тобто класами, що відповідають за реалізацію внутрішнього інтерфейсу IAlgorithm для вирішення поставленого завдання побудови шляху:

- rhs - або right-hand side - значення вершини графа, потенційна мінімальна відстань від поточної до цільової вершини графу;

- g - g значення вершини графу, мінімально відома відстань від цільової до поточної вершин графу;

- k2 - перемінна, яку алгоритми пошуку можуть використовувати на свій вибір. Наприклад, реалізація алгоритму пошуку D* Lite [21] може використовувати дану перемінну в якості другої компоненти двокомпонентного ключа вершини графу в пріоритетній черзі;

- k1 - перемінна, яку алгоритми пошуку можуть використовувати на свій вибір. Наприклад, реалізація алгоритму пошуку D* Lite [21] використовує дану перемінну в якості першої компоненти двокомпонентного ключа вершини графу в пріоритетній черзі.

Також, клас MapCell перевизначає декілька методів свого старшого класу Object:

- hashCode - оскільки клас MapCell перевизначає метод порівняння об'єктів, то за контрактом у мові Java, необхідно перевизначати і даний метод [26-29]. Крім того, у класі MapData, використовується спеціальна структура HashMap для зберігання об'єктів даного класу, яка використовує результат методу hashCode для обчислення хеш-функції кожного об'єкта. Хороша хеш-функція повинна задовольняти наступним властивостям [30-35]:

- мінімальна кількість колізій;

- швидке обчислення.

- equals - метод, що використовується для порівняння двох об'єктів класу MapCell. Може використовуватися і безпосередньо, але в основному в поточній реалізації використовується для коректного зберігання оброблюваних вершин графу у структурованій колекції. Вершини графу вважаються рівними, якщо збігаються їхні позиції, тобто перемінні типу - row та positionInRow.

Клітинки на реальній карті однозначно визначаються своїм положенням, а отже, можуть бути представлені за допомогою використання пари значень [x, y], де x - номер ряду, а y - номер зсуву в ряду. Однак, хеш-функція повинна повертати одне єдине ціле число.

Для розв'язання даної проблеми була використана функції Szudzik [36], яка дозволяє унікальним чином представити два натуральних числа одним:

$$[x,y]=\begin{cases} y^2+x, x\neq \max(x,y); \\ x^2+x+y, x=\max(x,y). \end{cases} \quad (3.1)$$

- toString - метод перевизначається для більш читабельного відображення достовірної інформації про клітинку під час налагодження та тестування роботи МРС [30-35].

Програмний код класу MapCell представлено в додатку А1.

Клас MapData забезпечує відображення реальної карти в граф. Оскільки дані від компонента Control транспортуються у байтовому закодованому вигляді і спочатку не підготовлені для виконання розв’язку поставленої задачі, то цей клас використовується для трансформації прийнятих масивів даних в об'єкти, придатні для роботи алгоритму пошуку. Крім того, даний клас MapData зберігає інформацію про оброблені вершини графу в міру роботи алгоритму пошуку, що дає змогу надалі повернутися до проблеми пошуку траєкторії шляху, якщо раптом знадобиться зупинити пошук або тимчасово вивантажити модуль керування із пам'яті [26-29].

Слід пам’ятати, що в поточній реалізації ми маємо вісім можливих напрямків переміщень. Така реалізація досить ефективна під час побудови траєкторії шляху і не надто ускладнює отримання суміжних вершин графу. Вага, втім, може бути змінена на нескінченність, якщо МРС виявить, що насправді ребро непрохідне [6-9].

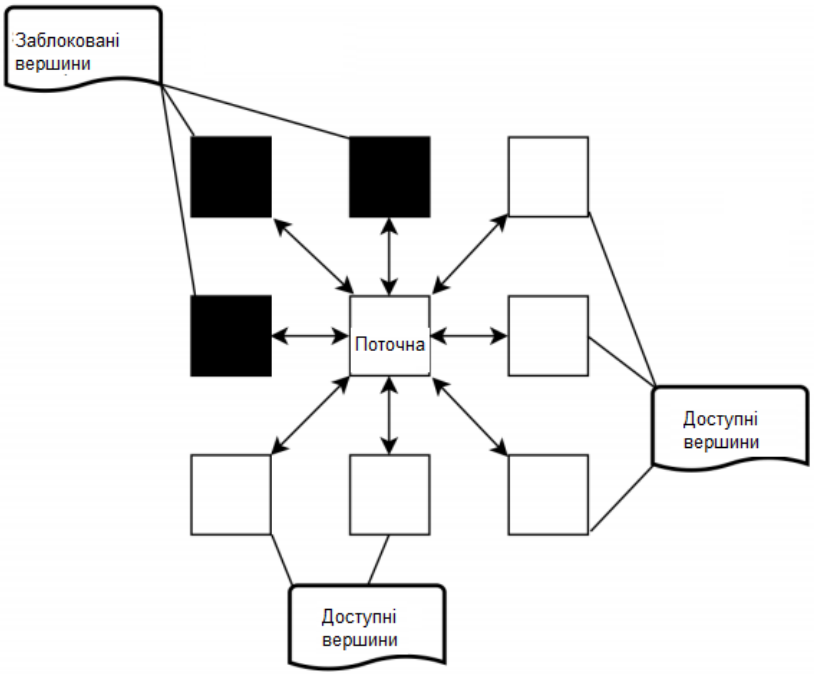


Рисунок 3.1 - Напрямки переміщень у графі

У полі mapData зберігається оригінальний масив даних реальної карти. Для того, щоб відокремити мета інформацію про реальну карту (ряди, кількість

клітинок у ряді, базове зміщення) використовується спеціальне байтове значення-роздільник Byte.MAX_VALUE. Його індекс у масиві зберігається у перемінній separator, це дозволяє ефективніше виявляти необхідну інформацію про клітинки, використовуючи його в якості зміщення даних вершини від мета-даних.

Щоб не звертатися до масиву щоразу, коли модулю керування необхідна інформація про конкретну клітинку, а так само, для того, щоб зберігати обраховані значення під час роботи алгоритму пошуку, створено структуру HashMap, засновану на хеш-таблицях, яка зберігає пари ключ-значення [26-29]. Дана структура зберігається у перемінній touchedCells і містить у собі дані про всі клітинки, які були коли-небудь запитані алгоритмом пошуку.

Основним «робочим» методом класу є getCellByCoords, який виконує відразу декілька функцій:

- обчислення унікального хеш необхідної клітинки карти за допомогою використання функції Szudzik [36];

- знаходження у колекції touchedCells необхідної клітинки карти з визначеними параметрами;

- у випадку якщо даної клітинки карти в колекції немає, виконання обчислень зсуву в масиві, які відповідають запитаним параметрам ряду і позиції в ряду, створення спеціального об'єкту MapCell, що відповідає потрібній клітині, і додавання в колекцію.

Таким чином модуль керування не створює ті об'єкти клітинок карти, до яких не доходить алгоритм пошуку під час обчислення траєкторії шляху, а отже, не займає зайву вбудовану пам'ять.

Усі операції, що надалі відбуваються над даними клітинками карти, так само фіксуються в спочатку створених об'єктах. Метод getCellByCoords буде використовуватись також і іншими методами класу MapData.

- getClosestCells - метод для отримання всіх вершин графа, сусідніх із запитуваною. Оскільки МРС може рухатись у восьми напрямках (праворуч,

ліворуч та по діагоналях, вгору, вниз), то даний метод може повернути максимум вісім клітин карти;

- getSucc і getPred - методи для отримання клітин карти, що виходять із поточної і тієї, що входить до неї відповідно. Дані методи представлені тільки для поділу логіки використання, оскільки обидва використовуються лише для виклику методу getClosestCells. Така реалізація через те, що в представленому графі є траєкторія шляху для будь-якої пари вершин графу, навіть незважаючи на те, що вага ребер деяких вершин графу може бути рівною +∞;

- getStartCell і getGoalCell - методи для отримання клітин карти, що представляють початкову та кінцеву вершину графу відповідно.

Метод getUpdatedCells використовується для оновлення даних змінених клітин карти. Він викликає для кожної з цих клітин карти метод updateCellByCoords, призначений для оновлення інформації про конкретну клітину карти, оскільки в модулі керування реалізовано інкрементний алгоритм пошуку D* Lite, який здатний враховувати зміни в графі та перебудовувати траєкторію шляху використовуючи попередні варіанти пошуки. Метод виконує наступні дії:

- оновлює значення клітинки карти в масиві mapData;

- знаходить та, відповідно, оновлює значення в об'єкті з колекції touchedCells, якщо клітина карта, яка шукається знаходиться в колекції;

- якщо клітини карти в колекції немає, її буде створено та додано [26-29].

Дана реалізація дає можливість не створювати зайвих об'єктів, якщо вони не належать до побудованої траєкторії шляху або змінених ваг, навіть якщо відповідні їм клітини карти змінилися під час переміщення МРС. Програмний код класу MapData представлено в додатку А.

3.2 Робота алгоритму з вагою ребер та евристичною функцією

Інтерфейс IDistanceCost надає нам можливість реалізувати класи, що

відповідають за визначення ваги ребер графу і опрацювати результат евристичної функції. Інтерфейс є частиною модуля керування і складається з двох методів, які потрібно реалізувати:

- getEdgeCost - даний метод повертає вагу ребра графа між двома переданими вершинами графа, а саме з першої в другу;

- getHeuristics - евристична функція, яка буде використовуватися алгоритмом пошуку для оцінки відстані від однієї вершини графу до іншої.

Програмний код інтерфейсу IDistanceCost представлено в додатку А.

У модулі керування також використовуються дві базові реалізації. Клас CellGroundDistanceCost використовує діагональну відстань як евристичну функцію і константи «1» та «1.4» як вартість переміщення MPC між сусідніми клітинами карти для вертикальних-горизонтальних і діагональних клітин карти відповідно.

Клас CellValueDistanceCost виконристовується для визначення значення висотності клітин карти. Тому за такої ж евристичної функції як у CellGroundDistanceCost, вартість переміщення між клітинами карти буде обчислюватись як модуль різниці між висотністю. Вихідний код класів, що реалізують інтерфейс IDistanceCost представлено в додатку А.

3.3 Робота з алгоритмами пошуку

Інтерфейс IAlgorithm дозволяє реалізувати класи, які представляють алгоритми пошуку. Методи, визначені в інтерфейсі, виконують лише базові функції, за допомогою яких алгоритм пошуку отримує дані про реальну карту та початкову і кінцеву вершини графу.

- setStartCell/setGoalCell - встановити початкову та кінцеву вершини графу;

- setMap - передає посилання на об'єкт MapData;

- setCostFunction - передає алгоритму пошуку посилання на реалізацію IDistanceCost, даючи змогу використовувати методи, які там розташовані;

- updatePosition/updateCellValue - передає оновлені значення поточного розташування MPC і зміни в значенні вершини графу відповідно.

Так само присутня пара керуючих методів, для запуску алгоритму пошуку і для отримання результату його роботи:

- buildPath - повертає масив байт, яким виконано кодування побудованого алгоритмом шляху – або пари значень [x, y], де x - номер ряду, а y - номер зсуву у ряду;

- findPath - починає пошук траєкторії шляху згідно з наявними даними [27-35].

Програмний код використовуваного інтерфейсу IAlgorithm представлено в додатку А.

Вхідною точкою розроблюваного модуля керування є клас PathStrategy. Який відповідає за комунікацію із зовнішнім компонентом Control, а також за створення та керування об'єктами всередині модуля керування. Так само, відповідає за прийом всіх вхідних даних і викликів методів від компонента Control, і перетворення у відповідні дані та виклики алгоритму пошуку. У своєму конструкторі клас PathStrategy запам'ятовує посилання на об'єкт Control і створює для модуля об'єкти класу IAlgorithm та IDistanceCost відповідно до переданих параметрів.

Ми робимо припущення, що даний клас PathStrategy має бути спадкоємцем або реалізовувати інтерфейс у тій системі, де відбуватиметься підключення модуля керування. З метою тестування було створено методи, що виконують емуляцію описаної вище поведінки:

- loadData - встановлює поточну карту для пошуку, стартові та початкові вершини графу, передає оновлені відомості про реальний стан карти;

- run - запускає пошук траєкторії шляху;

- exit - завершує виконання алгоритму пошуку, якщо воно все ще триває, витирає поточну карту, за якою ведеться пошук і звільняє займані ресурси [26-29].

Програмний код даного класу PathStrategy представлено в додатку А.

3.3.1 Використання алгоритму пошуку D* Lite [21]

У класі DLiteAlgorithm зосереджена реалізація алгоритму пошуку траєкторії шляху D* Lite з усіма необхідними допоміжними структурами та функціями, які можливо умовно розділити на дві групи: методи, що безпосередньо стосуються роботи алгоритму пошуку та допоміжні методи.

Допоміжні методи, а саме: lessThan і greaterThan використовуються для порівняння ключів клітин карти між собою, для коректного сортування пріоритетної черги.

Методи init, computeShortestPath, updateVertex, calcKey виконуються саме ті функції, що описані в псевдокоді в попередньому розділі, тобто ініціалізують компоненти алгоритму пошуку, обчислюють шлях, оновлюють під час роботи параметри вершин-клітин карти та обраховують ключі для пріоритетної черги. Метод getRHS обчислює rhs-значення.

Після того як метод computeShortestPath завершує своє виконання, можна спробувати побудувати траєкторію шляху, що і робить виклик методу createPath. Для створення масиву із даними про побудовану траєкторію шляху за вказаною специфікацією або повідомити про помилку, що траєкторію шляху не знайдено. Потім створений масив буде передано в головний клас PathStrategy для подальшого опрацювання.

Для зберігання вершин графу за пріоритетом їхніх ключів використовується спеціальна колекція PriorityQueue [30-35]. З можливістю зберігання вершини графу у відсортованому порядку, ґрунтуючись на переданій реалізації класу Comparator. Ключем для вершини графа слугує вже відома нам функція Szudzik [36], значення – сама вершина графа.

Програмний код класу DStarAlgorithm представлено в додатку А.

3.3.2 Використання алгоритму пошуку IDA* [19]

У реалізації алгоритму пошуку IDA* оголошуються перемінні g для

підрахунку вартості досягнення поточної вершини графа, яку розглядає алгоритм пошуку, і *f*-оціночну вартість найкоротшого шляху з початкової в кінцеву через поточну вершину.

Так само для того, щоб запам'ятовувати поточний пройдений шлях, створено спеціальну колекцію HashMap [30-35], в якій і ключем, і значенням є об'єкт класу MapCell. Пари поміщаються в колекцію в певному відношенні: ключ – це поточна розглянута вершина графа, а значення - вихідна вершина графа, що мінімізує відстань, що залишилася до кінцевої вершини графа.

Після закінчення роботи алгоритму пошуку IDA* [19] залишається тільки витягти вершини в наступній послідовності:

- встановити стартову вершину графа поточним ключем;

- послідовно витягувати вершини графу, які є значенням поточної, встановлюючи знову витягнуте значення як поточну вершину.

Згідно із псевдокодом представленим в другому розділі даної роботи, алгоритм пошуку реалізовано двома основними методами:

- search - метод, що виконує основний пошук. На кожній ітерації виконує перевірку поточної вершини графу на збіг з кінцевою і потім вибирає серед безлічі вихідних вершин ту, яка, на його думку, швидше приведе до виконання розв’язку задачі;

- performSearch - ініціалізаційний метод алгоритму пошуку, що позначає до розгляду стартову вершину графу і запускає основний цикл пошуку через постійні рекурсивні виклики, описаного вище, методу search [30-35].

Програмний код класу IDAStarAlgorithm представлено в додатку А.

3.4 Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

З метою реалізації розробленого алгоритму переміщення МРС було розроблено окремий клас, який імітує поведінку компонента керування в частині

створення модуля керування і передачі всіх необхідних параметрів. Такий підхід надасть можливість швидко і з мінімальними витратами подивитися, як поводиться модуль керування на реальних картах із різними типами перешкод і протестувати коректність реалізованих алгоритмів пошуку.

Для реалізації і випробування було створено карти місцевості, на яких і було заміряно показники реалізованих алгоритмів пошуку [19, 21].

Карта із лінійною перешкодою надає дві найкоротші траєкторії шляху: перешкоду можна обійти ліворуч або праворуч [37-39].

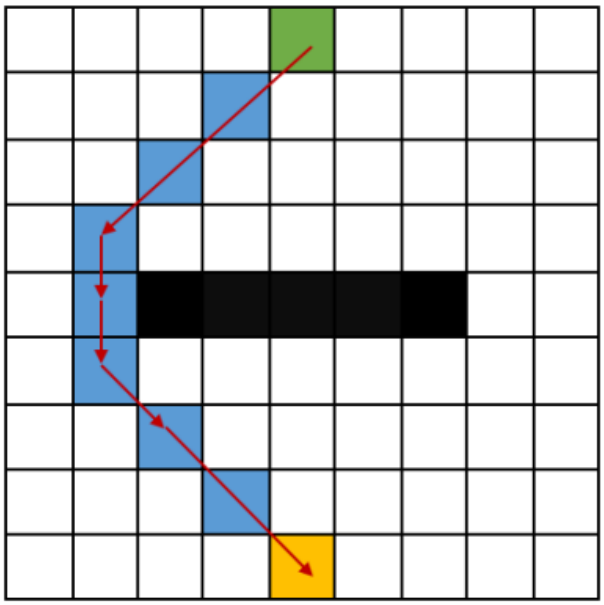


Рисунок 3.2 – Загальний вигляд траєкторії шляху на карті із лінійними перешкодами

Як представлено на рисунку 3.3, алгоритм пошуку D* Lite [21] показує помітний виграш у кількості виконуваних операцій. Через досить легку перешкоду, час роботи алгоритмів пошуку майже однаковий. Натомість, алгоритм пошуку IDA* показує значно менше споживання пам'яті.

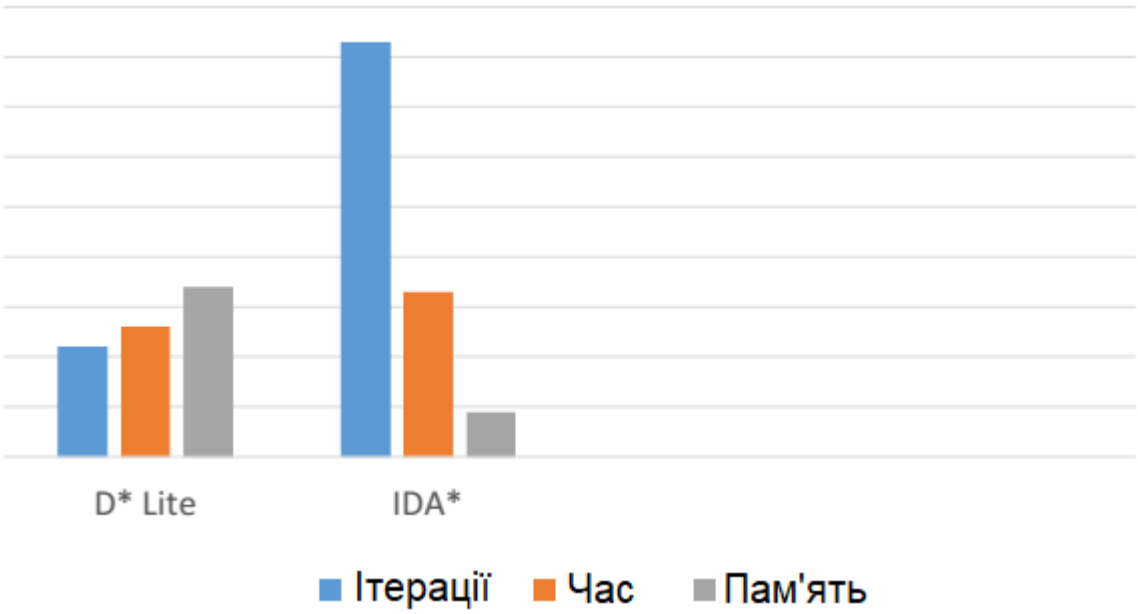


Рисунок 3.3 - Графік порівняння роботи алгоритмів пошуку на карті із лінійною перешкодою

Тепер подивимося на роботу алгоритмів пошуку після виявлення блокування на деякій частині траєкторії шляху.

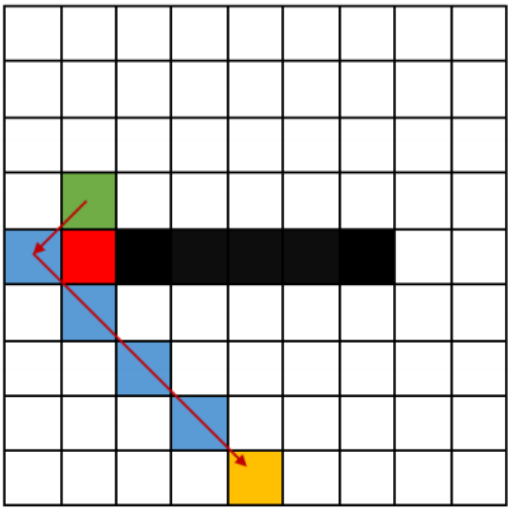


Рисунок 3.4 – Зміна траєкторії шляху на карті із лінійною перешкодою

Алгоритм пошуку IDA* [19] показує суттєвий виграш, хоча й виконує пошук із нульового положення. Це можливо пояснити тим, що нова проблема має дуже простий і прямолінійний розв'язок. Крім того, ми можемо вважати, що карта була повністю достовірна.

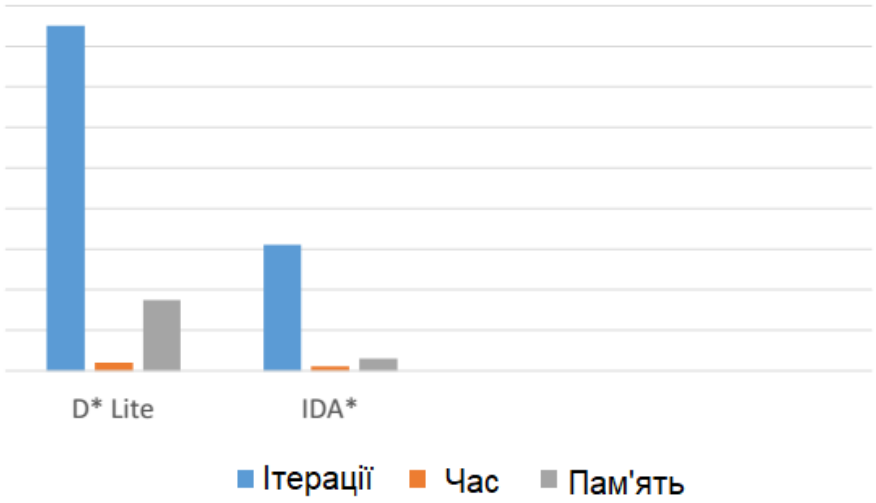


Рисунок 3.5 – Графік порівняння роботи алгоритмів пошуку на карті з лінійною перешкодою після зміни карти

Карта з Т-подібною перешкодою так само, як і карта з лінійною перешкодою, спочатку володіє двома однаковими за оптимальністю траєкторії шляху.

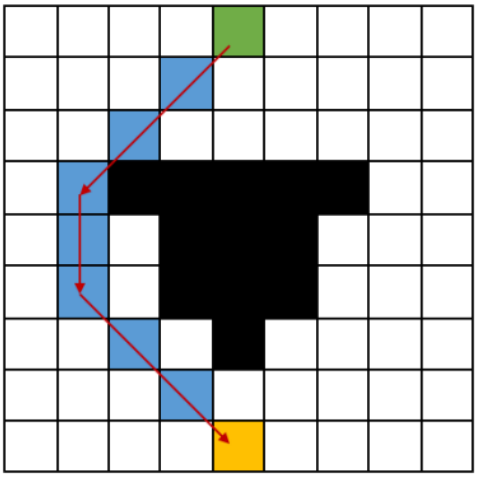


Рисунок 3.6 – Загальний вигляд траєкторії шляху на карті з Т-подібною перешкодою

Оскільки дана карта однакового розміру з картою з лінійною перешкодою (див. рис. 3.2), і точки початку та кінця шляху теж мають збіг, то вся різниця тільки у формі перешкоди та кількості непрохідних клітин - яких стало більше. Що накладає певний відбиток і на роботу алгоритмів пошуку: алгоритм пошуку

IDA* [19] наблизився до алгоритму пошуку D*Lite [21]. Так вийшло через те, що на реальній карті залишилося менше прохідних клітин.

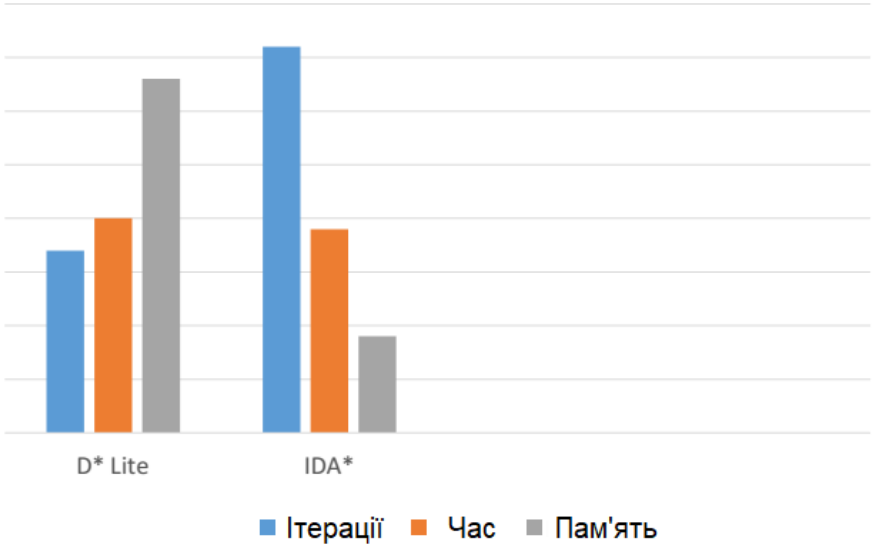


Рисунок 3.7 – Графік порівняння роботи алгоритмів пошуку на карті з Т-подібною перешкодою

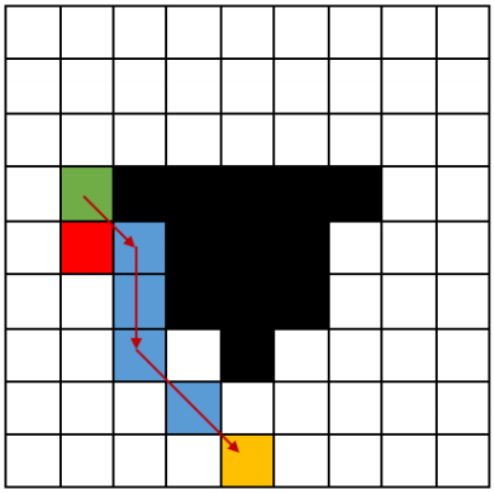


Рисунок 3.8 - Зміна траєкторії шляху на карті з Т-подібною перешкодою

Після блокування однієї з клітинок карти шляху, алгоритм пошуку IDA* [19] знову у виграші за параметрами ітерацій і пам'яті, але гірший за часом виконання операцій. Такий виграш з боку алгоритму пошуку IDA* [19] забезпечується досить прямолінійним рішенням, тобто евристична функція, що спрямовує пошук, дає дуже точну інформацію про реальну траєкторію шляху.

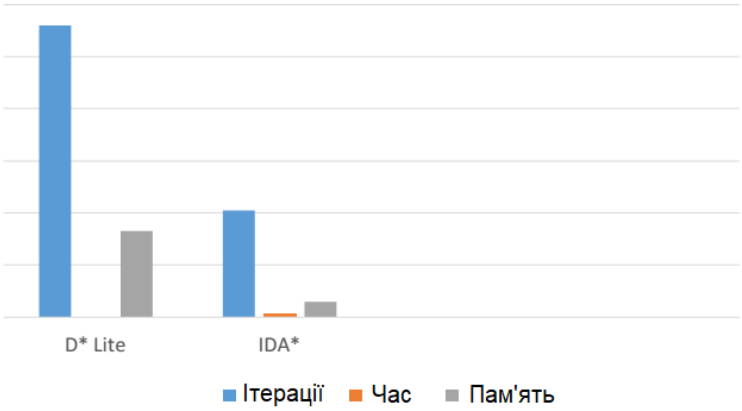


Рисунок 3.9 - Графік порівняння роботи алгоритмів на карті з Т-подібною перешкодою після зміни карти

Карта з U-подібною перешкодою вже дещо складніша через складну форму перешкоди.

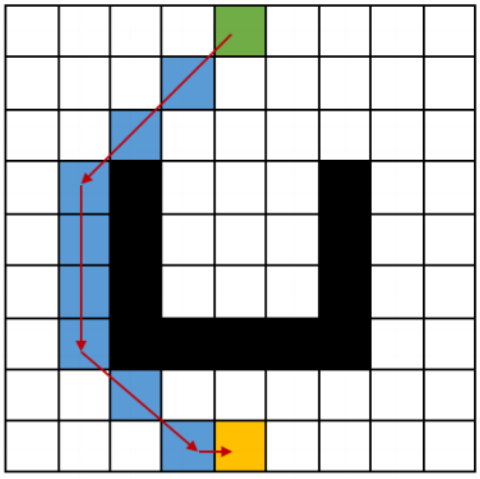


Рисунок 3.10 – Загальний вигляд траєкторії шляху на карті з U-подібною перешкодою

Під час спроби пройти фігуру наскрізь у пошуках розв'язку алгоритм пошуку IDA* [19], не пам'ятаючи попередніх ітерацій, весь час потрапляє, так би мовити, в «пастку». Це добре видно на рисунку 3.11. Виграш порівняно з алгоритмом пошуку D* Lite [21] тільки в частині використання вбудованої пам'яті.

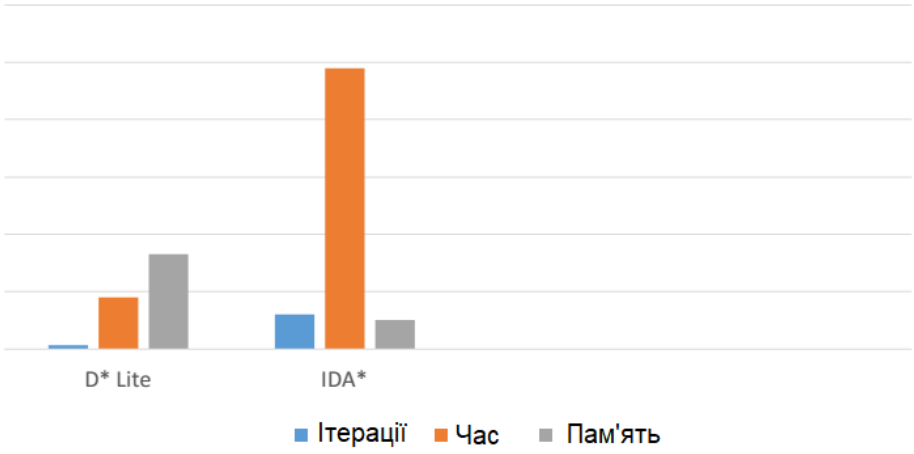


Рисунок 3.11 - Графік порівняння роботи алгоритмів пошуку на карті з U-подібною перешкодою

Після зміни добре видно переваги алгоритму пошуку D* Lite [21] у динамічно змінюваному навколишньому середовищі.

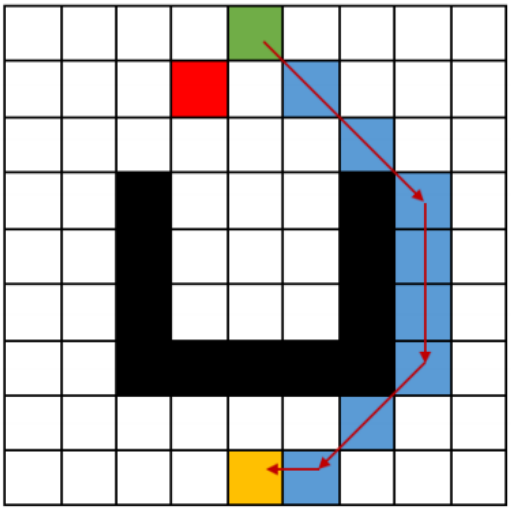


Рисунок 3.12 - Зміна траєкторії шляху на карті з U-подібною перешкодою

Як показує графічна залежність на рисунку 3.13, відрив алгоритму пошуку D* Lite [21] зріс, порівняно з початковим пошуком шляху (див. рис. 3.11). Справа в тому, що алгоритм пошуку D* Lite [21] шукає у напрямку від кінцевої вершини графу до початкової і для нього заблокована вершина графу з'являється тільки під кінець шляху [37-39].

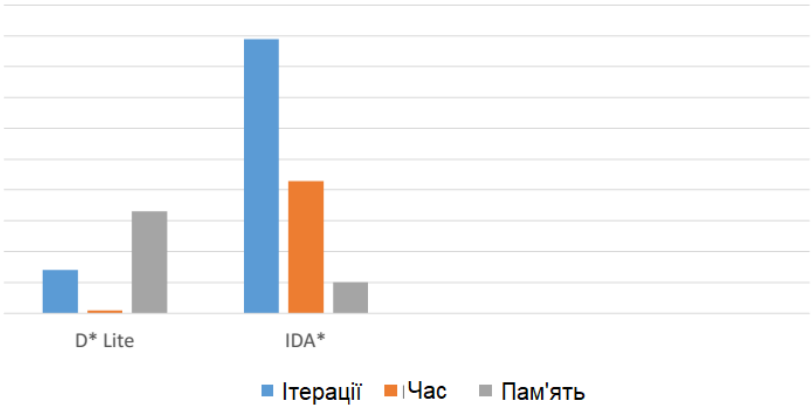


Рисунок 3.13 – Графік порівняння роботи алгоритмів пошуку на карті з U-подібною перешкодою після зміни карти

Карта з П-подібною перешкодою - вдосконалений варіант попереднього завдання. Стартова точка цього разу розташована всередині перешкоди, а кінцева – за її спиною. Спочатку алгоритмам пошуку доведеться вибратися з перешкоди, виконуючи рух у протилежному від кінцевої точки напрямку.

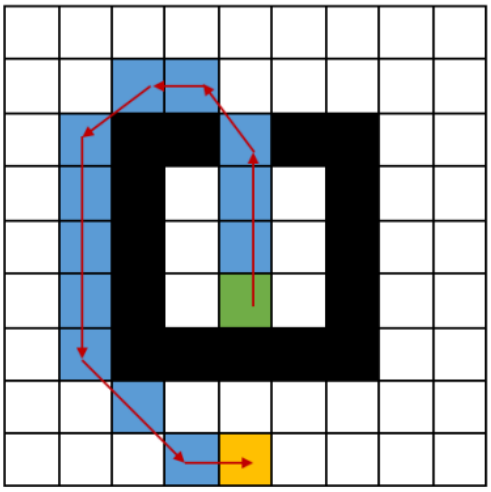


Рисунок 3.14 – Загальний вигляд траєкторії шляху на карті з П-подібною перешкодою

Очікувано, алгоритм пошуку D* Lite [21] швидше справляється із поставленим завданням і цього разу не так вже й сильно програє за кількістю використаної вбудованої пам'яті.

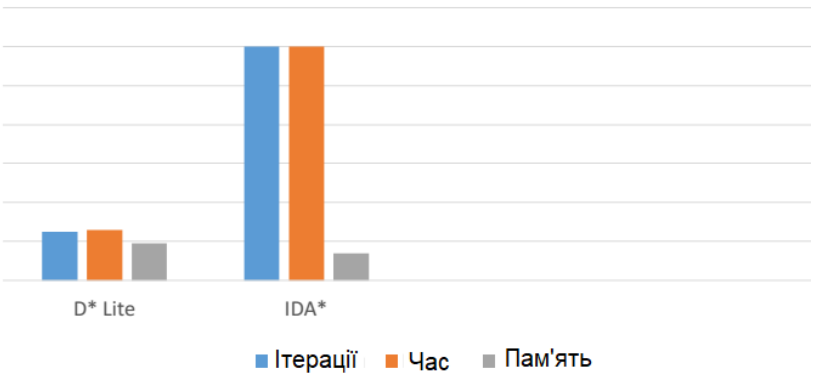


Рисунок 3.15 – Графік порівняння роботи алгоритмів пошуку на карті з П-подібною перешкодою

Ми спеціально внесли таку легку зміну в карту, щоб показати, що не завжди доцільно використовувати один і той самий алгоритм пошуку.

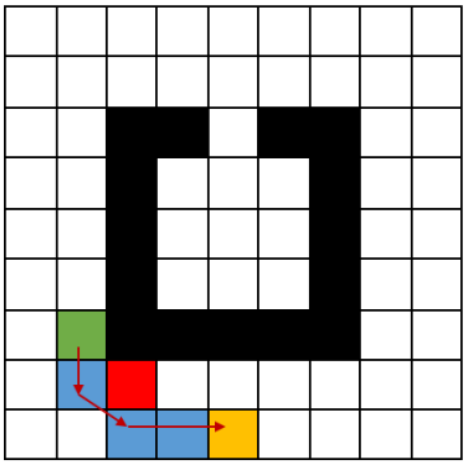


Рисунок 3.16 – Зміна траєкторії шляху на карті з П-подібною перешкодою

Алгоритм пошуку IDA* [19] очікувано виявився кращим. Якщо зменшити область пошуку до проблеми, що залишилася, очевидно, що початкова і кінцева точки знаходяться вже дуже близько, а навколо залишилося зовсім небагато клітин карти [37-39].

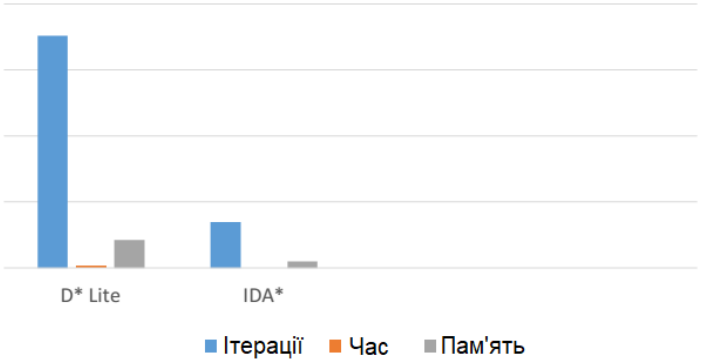


Рисунок 3.17 – Графік порівняння роботи алгоритмів пошуку на карті з П-подібною перешкодою після зміни карти

Карта з трикутною перешкодою - досить цікава вправа. Оптимальна траєкторія шляху тут лише одна, фігура досить складна і алгоритм пошуку D* Lite [21] очікувано спрацьовує швидше.

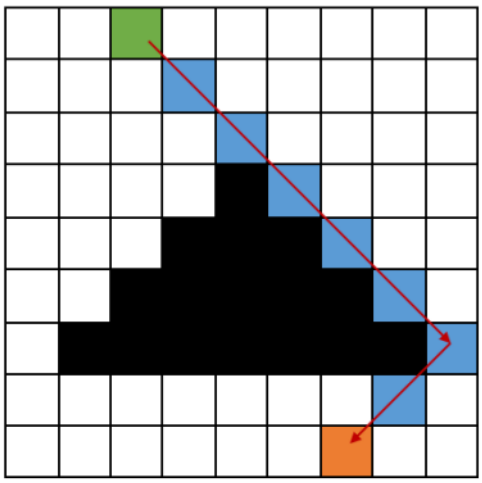


Рисунок 3.18 – Загальний вигляд траєкторії шляху на карті з трикутною перешкодою

Як і завжди, програш за використаною пам'яттю у алгоритму пошуку D* Lite [21] очевидний. Зате добре помітно, наскільки швидше даний алгоритм вирішує поставлене завдання. Як показано на рисунку 3.19, при вдвічі вищому споживанні пам'яті показує в рази кращий результат за параметрами ітерацій та загального часу роботи.

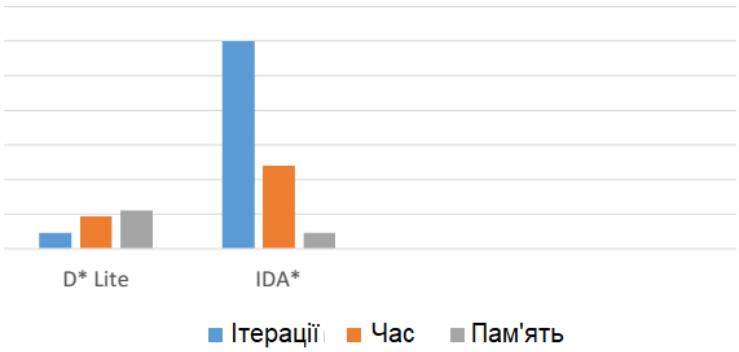


Рисунок 3.19 – Графік порівняння роботи алгоритмів пошуку на карті з трикутною перешкодою

Тепер виконаємо блокування не просто сусідньої клітинки, а взагалі будь-яку близьку до раніше побудованої траєкторії шляху.

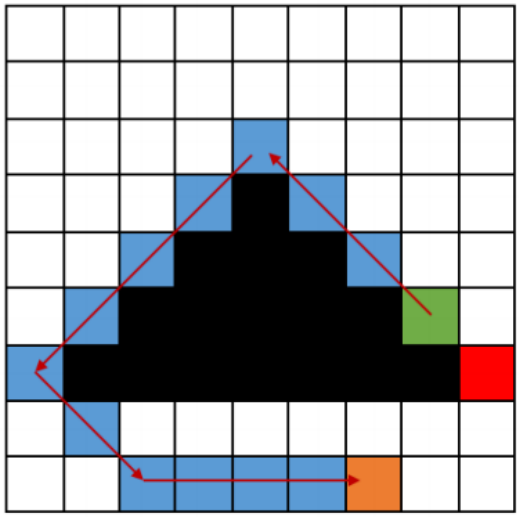


Рисунок 3.20 – Зміна траєкторії шляху на карті з трикутною перешкодою

На рисунку 3.21 під час порівняння даних з рисунком 3.19 добре видно, що для алгоритму пошуку D* Lite [21] завдання було розв'язано за менший проміжок часу, ніж спочатку, а ось для алгоритму пошуку IDA* [19] такий пошук рівнозначний абсолютно новому. Через те, що евристична функція показує один напрямок пошуку, а переміщуватись для вирішення поставленого завдання потрібно спочатку в протилежному напрямку, алгоритм пошуку IDA* [19] витратив досить суттєву кількість часу, що й представлено на рисунку 3.21. У даному тесті добре показано переваги інкрементного пошуку алгоритму

пошуку D* Lite [21].

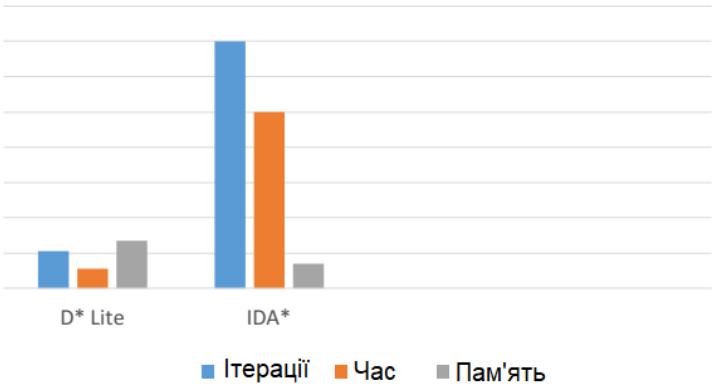


Рисунок 3.21 – Графік порівняння роботи алгоритмів пошуку на карті з трикутною перешкодою після зміни карти

Карта з перешкодою у формі зірки характеризується більшою кількістю заблокованих клітинок картки по відношенню до вільних клітинок карти.

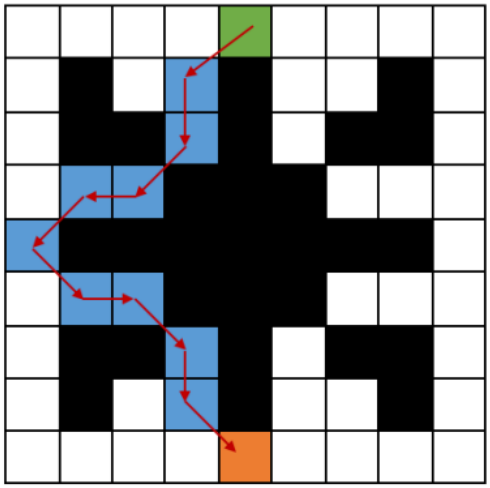


Рисунок 3.22 – Загальний вигляд траєкторії шляху на карті з перешкодою у формі зірки

Через складність фігури перешкоди алгоритм пошуку IDA* [19] традиційно програє, навіть споживання вбудованої пам'яті цього разу майже збігається. Цікаво, що через розташування початкової та кінцевої точок на карті і форми перешкоди алгоритм пошуку IDA* [19] намагався весь час рухатися вниз, періодично натикаючись на так звані «промені» перешкоди. Крім того, симетричність перешкоди щодо центральної лінії тільки ускладнювала

поставлене завдання. Звідси такий програш алгоритму пошуку D* Lite [21] за кількістю звернень до вершин графу [37-39].

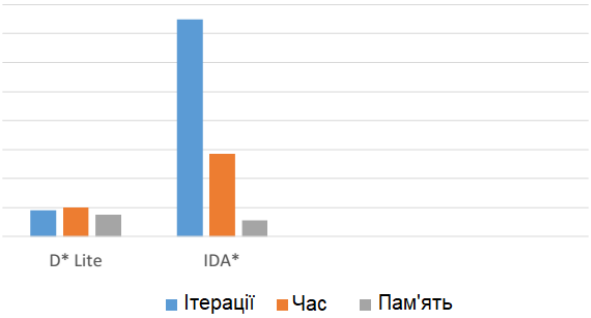


Рисунок 3.23 – Графік порівняння роботи алгоритмів пошуку на карті з перешкодою у формі зірки

Тут склалась ситуація, досить схожа із завданням, представленим на рисунку 3.22. Мала кількість варіантів, що залишилися, і використання евристики допомагають алгоритму пошуку IDA* [19] виграти.

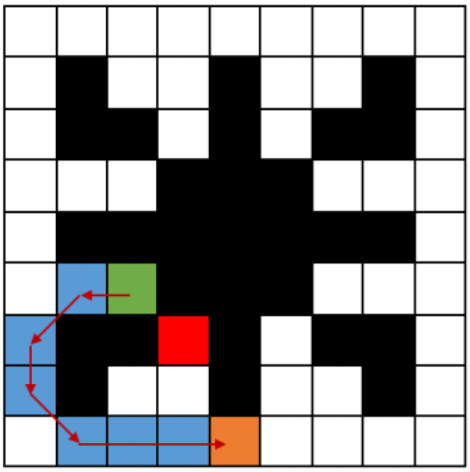


Рисунок 3.24 – Зміна траєкторії шляху на карті з перешкодою у формі зірки

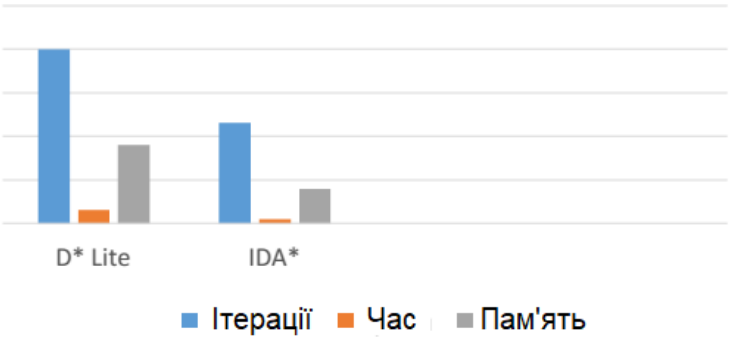


Рисунок 3.25 – Графік порівняння роботи алгоритмів пошуку на карті з перешкодою у формі зірки після зміни карти

Ускладнюємо завдання - карта з лабіринтом.

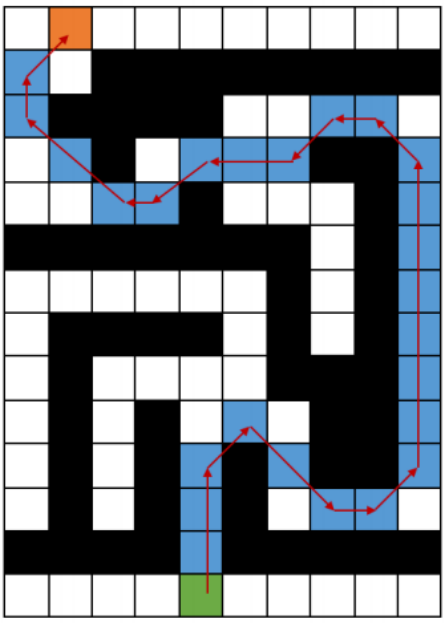


Рисунок 3.26 – Загальний вигляд траєкторії шляху на карті з лабіринтом

У лабіринту три шляхи, що ведуть у замкнутий простір, а евристична функція надто оптимістична щодо реального найкоротшого шляху. Що й відображається на результатах тесту представлених на рисунку 3.27.

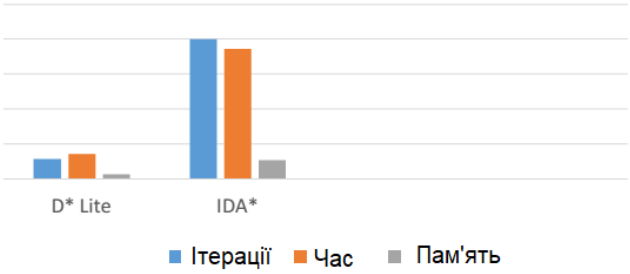


Рисунок 3.27 – Графік порівняння роботи алгоритмів пошуку на карті з лабіринтом

Після внесення зміни на карту навіть відносна близькість стартової та кінцевої точок не рятує алгоритм пошуку IDA* [19] від великої кількості ітерацій під час пошуку оптимального рішення [37-39].

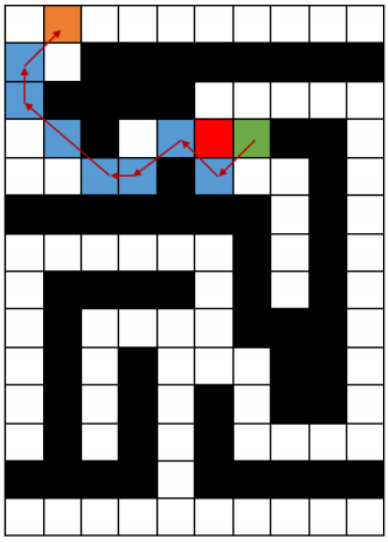


Рисунок 3.28 – Зміна траєкторії шляху на карті з лабіринтом

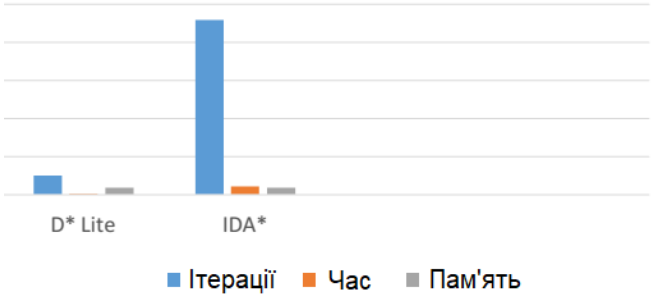


Рисунок 3.29 – Графік порівняння роботи алгоритмів пошуку на карті з лабіринтом після внесення зміни на карту

Карта, на якій розташована непрохідна перешкода - окрема проблема для алгоритму пошуку IDA* [19]. Якщо алгоритм пошуку D* Lite [21] закінчує пошук коли огляне всі клітинки розташовані навколо заблокованої області (оскільки він шукає від кінцевої вершини до початкової вершини), то алгоритм пошуку IDA* [19] ніколи не визначить, що рішення проблеми немає. Однак, реалізація алгоритму пошуку IDA* [19] у розроблюваному модулі керування дозволяє закінчити пошук спираючись на кількість здійснених ітерацій [37-39].

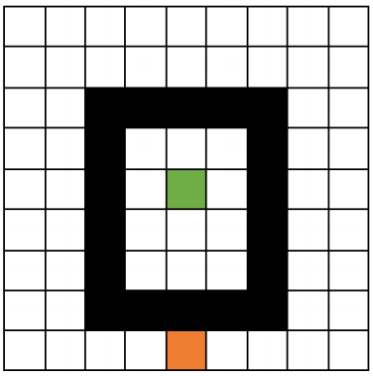


Рисунок 3.30 – Загальний вигляд карти із непрохідною перешкодою

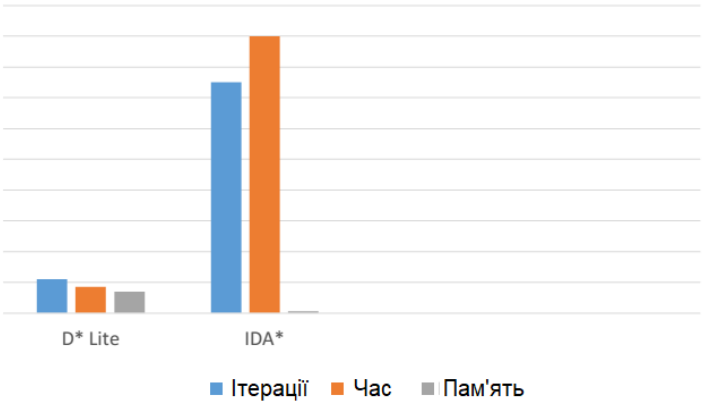


Рисунок 3.31 – Графік порівняння роботи алгоритмів пошуку на карті з непрохідною перешкодою

Після зміни карти і появи шляху алгоритм пошуку D* Lite [21] практично миттєво знайшов розв'язок поставленого завдання, оскільки, володіючи інкрементним пошуком, пам'ятав свою попередню спробу вирішення задачі. Такий підхід зробив його відрив від алгоритму пошуку IDA* [19] ще більшим, ніж зазвичай, що представлено на рисунку 3.33 [37-39].

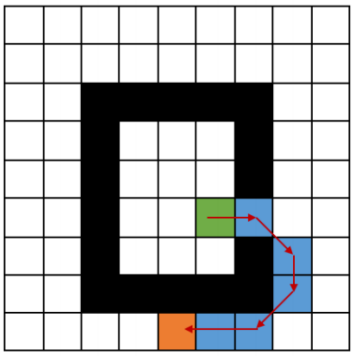


Рисунок 3.32 - Зміна шляху на карті з початково непрохідною перешкодою

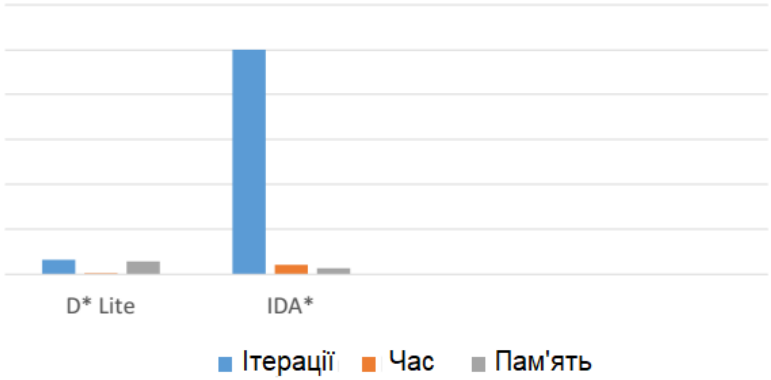


Рисунок 3.33 – Графік порівняння роботи алгоритмів пошуку на карті з початково непрохідною перешкодою після внесення зміни на карту

Як видно з проведених симуляцій, у різних ситуаціях у виграші виявляються різні алгоритми пошуку. Однозначно можна вказати, що якщо потрібно ефективне використання вбудованої пам'яті і можна пожертвувати часом виконання обрахунків, то алгоритм пошуку IDA* [19] відмінно вирішує основні поставлені завдання, щоправда лише після невеликих доопрацювань реалізації алгоритму пошуку, що не будуть давати змоги алгоритму пошуку зациклюватись в деяких випадках. Якщо ж питання вбудованої пам'яті не стоїть гостро, то алгоритм пошуку D* Lite [21] прекрасно впорається із поставленим завданням. З результатів перевірки очевидно, що чим складніші та різноманітніші перешкоди реалізовані на карті, тим більше потенційний виграш алгоритму пошуку D* Lite [21], особливо, якщо прийняти до уваги природу багатьох місцевостей схильну до змін.

Також розглянемо різні способи обчислення ваги ребер графу та евристичні функцій. Задаючи різні евристичні функції ми можемо керувати напрямком пошуку, допомагаючи алгоритмам пошуку шляху приймати рішення щодо того, які вершини графу кращі для просування до кінцевої цілі.

Вага ребер графу, своєю чергою, може враховувати не лише фактичне місце розташування клітин карти відносно одна одної, а й інші дані, що можуть накладати свої умови на вартість переміщення MPC між даними клітинами. Наприклад, на реальній місцевості можуть використовуватись різні види

штрафів за переміщення по воді [37-39].

В якості прикладу створено спеціальну тестову карту, представлену на рисунку 3.34.

	1	2	3	4	5
A	-2	1	7	14	21
B	12	2	11	-7	-126
C	126	3	21	13	126
D	2	4	126	8	126
E	7	5	6	7	0

Рисунок 3.34 – Загальний вигляд тестової карти із зазначенням висот клітин

Особливістю даної карти є додаткові значення, що відображають перепад висоти клітини карти відносно нульового значення. Таким чином буде відображено, що можливо керувати поняттям оптимальності побудованого шляху.

Як видно із характеристик тестової карти, найкоротший у координатному сенсі шлях проходить через групу вершин [2A], [3B], [4B], [3C], [4C], [4D], про те більш плавний шлях, що, в свою чергу, враховує перепади висот, через стовпець №2 і ряд E, тобто - [2A], [2B], [2C], [2D], [2E], [3E], [4E], [4D].

Переглянемо результат роботи алгоритму пошуку, запущеного з реалізацією з класу CellGroundDistanceCost (евристична функція – діагональна відстань, вага ребер графу - константні значення) [37-39].

	1	2	3	4	5
A	-2	1	7	14	21
B	12	2	11	-7	-126
C	126	3	21	13	126
D	2	4	126	8	126
E	7	5	6	7	0

Рисунок 3.35 – Загальний вигляд шляху, заснований на найкоротшій відстані

Алгоритм пошуку побудував шлях через вершини [2A], [2B], [3C], [4D], як ми і припускали.

Тепер оцінимо результат з реалізацією з класу CellValueDistanceCost (евристична функція - діагональна відстань, вага ребер графу - константні значення, додані до модуля різниці висот) (рис. 3.36).

	1	2	3	4	5
A	-2	1	7	14	21
B	12	2	11	-7	-126
C	126	3	21	13	126
D	2	4	126	8	126
E	7	5	6	7	0

Рисунок 3.36 – Загальний вигляд шляху, заснований на різниці висот

Траєкторія шляху тепер дійсно найкоротша з врахуванням різниці параметра висотності у вершинах: [2A], [2B], [2C], [2D], [3E], [4D].

3.5 Висновки до третього розділу

Змодельовано роботу алгоритму переміщення із графом та роботу алгоритму пошуку оптимальної траєкторії руху з вагою ребер та евристичною функцією.

Реалізовано розроблений алгоритм переміщення MPC, виконано порівняння двох використаних алгоритмів пошуку оптимальної траєкторії руху MPC.

ВИСНОВКИ

Основним результатом кваліфікаційної роботи бакалавра є створення проєкту програмної реалізації алгоритмів для використання в МРС в якості модуля пошуку оптимального шляху на карті.

У процесі розв'язання поставленої задачі в робочому середовищі МРС (з урахуванням різноманітності існуючих МРС, операційних систем, способів керування чи прийняття рішень) ми розробили проєкт програмної реалізації алгоритмів у вигляді модуля, який здатний виконувати наступні функції:

- виконання обробки отриманих даних про навколишнє середовище та створення згідно отриманих даних внутрішньої структури типу «граф»;

- пошук оптимальної траєкторії шляху за допомогою використання різних алгоритмів, серед яких присутні наступні: орієнтування на скорочення часу пошуку, на розв'язання складних завдань, на економне споживання ресурсів;

- надання МРС, декількох стратегій пошуку, тобто фактичних комбінацій функцій підрахунку вартості переміщення та алгоритмів пошуку, із використанням евристичної функції;

- можливість розширення функціональності модуля керування МРС шляхом додавання користувальницьких реалізацій алгоритмів пошуку, евристичних функцій та способів підрахунку вартості переміщення МРС.

Так як алгоритми прописані із використанням мови JAVA, модуль керування легко портується на будь-які інші мови програмування. Архітектура проєкту розроблена таким чином, що інтеграція в наявні МРС повинна бути простою і не витратною. Модуль керування використовує реалізацію двох найпоширеніших алгоритмів пошуку для розв'язання досить великого кола завдань пошуку оптимального шляху, при цьому, даючи змогу надалі легко розширити діапазон використовуваних алгоритмів пошуку і навіть, областей використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Principles of Robot Motion: Theory, Algorithms, and Implementations / Howie Choset, Kevin M. Lynch, Seth Hutchinson. – Boston.: A Bradford Book, 2005. – 632 с.

2. Anthony (Tony) Stentz. Optimal and efficient path planning for partially-known environments / Anthony (Tony) Stentz // Proceeding of the IEEE International Conference on Robotics and Automation (ICRA'94). – 1994. - №4. – С. 3310-3317.

3. J.C. Latombe. Robot Motion Planning. – Boston.: Kluwer Academic Publishers, 1991. – 672 с.

4. Stefan Edelkamp, Stefan Schroedl. Heuristic search: theory and applications, 1 edition. - San Francisco.: Morgan Kaufmann, 2011. - 712 с.

5. Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search / Richard E. Korf // Artificial Intelligence. – 1985. - № 27. – С. 97-109.

6. Теорія графів. [Електронний ресурс]: навч. посіб. для здобувачів ступеня бакалавра за освітньою програмою «Комп'ютерний моніторинг та геометричне моделювання процесів і систем» спеціальності 122 «Комп'ютерні науки» / І.М. Кузьменко; КПІ ім. Ігоря Сікорського. - Електронні текстові дані (1 файл: 1,7 Мбайт). — Київ: КПІ ім. Ігоря Сікорського, 2020. - 71 с.

7. Кузьменко В. В., Швачич Г. Г, Рижанкова Г. І., Пасинков В. М. Основи дискретної математики. Розділ "Елементи теорії графів": Конспект лекцій. – Дніпропетровськ: НМетАУ, 2004. – 38с.

8. Kenneth H. Rosen Discrete Mathematics and Its Applications 2002 by McGraw-Hill Science, 928 p.

9. Бондаренко М.Ф., Білоус Н.В., Руткас А.Г. Комп'ютерна дискретна математика. - Харків: "Компанія Сміт", 2004. - 480 с.

10. .MAP Розширення файлу [Електронний ресурс] – Режим доступу: https://rocketdrivers.com/uk/file-extensions/map-4661

11. David T. Wooden. Graph-based Path Planning for Mobile Robots.

[Електронний ресурс] – Режим доступу:

https://smartech.gatech.edu/bitstream/handle/1853/14055/wooden_david_t_200611_phd.pdf

12. Алгоритм пошуку A*. [Електронний ресурс] – Режим доступу:

https://www.wikidata.uk-ua.nina.az/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83_A*.html

13. Moszyńska, M., & Richter, W. D. (2012). Reverse triangle inequality. Antinorms and semi-antinorms. Studia Scientiarum Mathematicarum Hungarica, 49(1), 120-138.

14. Огірко О. І., Галайко Н. В. Теорія ймовірностей та математична статистика: навчальний посібник / О. І. Огірко, Н. В. Галайко. – Львів: ЛьвДУВС, 2017. – 292 с.

15. Спеціальні розділи математики. Статистичний аналіз даних у середовищі STATISTICA [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології», спеціалізації «Комп’ютерно-інтегровані технології сталих хімічних виробництв» / КПІ ім. Ігоря Сікорського ; уклад.: І. М. Джигирей, Д. М. Складанний. – Електронні текстові данні (1 файл: 1,41 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2019. – 74 с.

16. Artificial Intellingence. Articles on artificial intelligence. Iterative Deepening A Star. [Електронний ресурс] – Режим доступу:

http://intelligence.worldofcomputing.net/aisearch/iterative-deepening-a-star.html

17. Anthony Stentz. The Focussed D* Algorithm for Real-Time Replanning. [Електронний ресурс] – Режим доступу:

http://www.frc.ri.cmu.edu/~axs/doc/ijcai95.pdf

18. Anytime Dynamic A*: An Anytime, Replanning Algorithm / Maxim Likhachev, Dave Ferguson, Geoff Gordon та ін. [Електронний ресурс] – Режим

доступу:

http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf

19. Zarembo I., Kodors S. Pathfinding Algorithm Efficiency Analysis in 2D Grid [Text] / Zarembo I., Kodors S. – 9th International Scientific and Practical Conference, 2013 – P. 46-50.

20. Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong Planning A*. [Електронний ресурс] – Режим доступу:

http://www.cs.cmu.edu/~./maxim/files/aij04.pdf

21. Sven Koenig, Maxim Likhachev. D* Lite: [Електронний ресурс] – Режим доступу:

http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

22. Oracle Help Center. Object (Java Platform SE 7). [Електронний ресурс] – Режим доступу:

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

23. Копитко М.Ф. Основи програмування мовою Java: Тексти лекцій. − Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002.− 83 с.

24. Кадомський К.К. Java. Теорія і практика: навчальний посібник для студентів природничих спеціальностей університетів / Кадомський К.К., Ніколюк П.К. – Вінниця: Донну, 2019. – 197 с.

25. Incremental Heuristic Search in Artificial Intelligence / Sven Koenig, Maxim Likhachev, Yaxin Liu, David Furcy. [Електронний ресурс] – Режим доступу:

http://idmlab.org/bib/abstracts/papers/aimag04a.pdf

26. Бублик В.В. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.: іл. ISBN 978-966-97182-1-1.

27. Настенко Д. В. Об’єктно-орієнтоване програмування. Частина 1. Основи об’єктно-орієнтованого програмування на мові C# [Електронний ресурс] : навчальний посібник для бакалаврів напряму підготовки 6.050701 «Електротехніка та електротехнології» програми професійного спрямування

«Системи управління виробництвом та розподілом електроенергії» / Д. В. Настенко, А. Б. Нестерко ; НТУУ «КПІ». – Електронні текстові дані (1 файл: 931,2 Кбайт). – Київ : НТУУ «КПІ», 2016. – 76 с. – Назва з екрана.

28. Єфіменко С.В. Методичний посібник з курсу «Об’єктно-орієнтоване програмування. Мови С/С++». – К.: КНУ ім. Т. Шевченко, 2021. – 124 с.

29. Основи об’єктно-орієнтованого програмування : навч. посібник / Гришанович Т. О., Глинчук Л. Я.; ВНУ імені Лесі Українки. Електронні текстові данні (1 файл: 998 КБ). Луцьк : ВНУ імені Лесі Українки, 2022. – 120 с.

30. Креневич А.П. Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с.

31. Ю.Є. Грудзинський. Алгоритми та структури даних. Навчальний посібник / Ю.Є. Грудзинський // К.: НТУУ "КПІ ім. Ігоря Сікорського", 2022. – 215 с.

32. Коротєєва Т.О. Алгоритми та структури даних. Навчальний посібник. – Львів: Видавництво Львівської політехніки, 2014. – 280 с.

33. R. Stephens, Essential Algorithms. A Practical Approach to Computer Algorithms Using Python and C#, 2nd Edition ред., Indianapolis: John Wiley & Sons, Inc., 2019, p. 782.

34. R. Thareja, Data Structures Using C, 2nd Edition ред., New Delhi: Oxford University Press, 2014, p. 557.

35. Kent D. Lee. Data Structures and Algorithms with Python / Kent D. Lee, Steve Hubbard., – Springer, 2015. – 363 с.

36. Matthew Szudzik. An Elegant Pairing Function. [Електронний ресурс] – Режим доступу:

http://szudzik.com/ElegantPairing.pdf

37. Лисенко О. І., Тачиніна О. М., Пономаренко С. О., Гуйда О. Г. Теорія оптимальних розгалужених траєкторій / О. І. Лисенко, О. М. Тачиніна, С. О. Пономаренко, О. Г. Гуйда – К: КПІ ім. Ігоря Сікорського., 7БЦ, 2023. – 260 с. ISBN 978-617-549-163-8

38. Автоматизоване планування оптимальних траєкторій переміщення мобільних мехатронних пристроїв. [Електронний ресурс] – Режим доступу: https://learn.ztu.edu.ua/pluginfile.php/308148/mod_resource/content/1/%2B%D0%9B9%20-%2010_%D0%A0%D0%A2%D1%82%D0%B0%D0%9C%D0%A2.pdf

39. Петрівський, Я. Б., Петрівський, В. Я., Шевченко, В. Л. і Сініцин, І. П. (2023) «Оптимізація траєкторії руху датчиків з урахуванням важливості ділянок території моніторингу та ймовірності виявлення об’єктів», International Scientific Technical Journal "Problems of Control and Informatics", 67(2), с. 6–21. doi: 10.34229/2786-6505-2022-2-1.

40. Кваліфікаційна робота : методичні вказівки щодо її виконання для студентів спеціальності 151 «Автоматизація та комп’ютерно-інтегровані технології» / уклад.: Ю.В. Форкун, Г.І. Радельчук, І.В. Форкун, А.С. Каштальян, В.В. Мартинюк. Хмельницький : ХНУ, 2020. – 50 с.

ДОДАТКИ

Додаток А1

Код класів роботи із графом

```
//Клас вершини
public class MapCell implements Serializable {
int row, positionInRow;// позиція вершини на карті
int rowLength; // довжина всього рядка
int positionOffset; // зміщення в рядку
int cellValue; // значення вершини
double k1, k2; // збереження випадкових значень
double g, rhs; //g та rhs значення вершини
// Конструктор
public MapCell(int row, int positionInRow, int cellValue){
this.row = row;
this.positionInRow = positionInRow;
this.cellValue = cellValue;
g = rhs = Double.POSITIVE_INFINITY;
}
// Вершини рівні тільки у випадку, якщо це одна і та сама вершина
// співпадають рядок і стовпець
@Override
public boolean equals(Object obj) {
if(obj instanceof MapCell){
return row == ((MapCell)obj).row && positionInRow ==
((MapCell)obj).positionInRow;
}
return false;
}
// функція Сзудзика для хешу
@Override
```

```
public int hashCode() {
return row >= positionInRow ? row * row + row + positionInRow : row +
positionInRow * positionInRow;
}
@Override
public String toString() {
return "["+row+","+positionInRow+"]";
}
}
// Клас графу
public class MapData implements Serializable{
private byte[] mapData; // реальні дані
private MapCell startCell;// початкова вершина
private MapCell goalCell;// кінцева вершина
// Тут зберігаємо об’єкти вершин, які створили
private HashMap<Integer, MapCell> touchedCells = new HashMap<Integer, MapCell>();
int separator;// відокремлює карту від мета-даних в масиві
// Конструктор
public MapData(byte[] map, int dataOffset){
mapData = new byte[map.length - dataOffset];
System.arraycopy(map, dataOffset, mapData, 0, mapData.length);
for(int i = 0; i < mapData.length; i++){
System.out.println(mapData[i]);
if(mapData[i] == Byte.MIN_VALUE){
separator = i;
break;
}
}
```

```
        }
        public int getMapID(){
        return (int)map[0];
        }
        public MapCell getGoalCell(byte[] data, int dataOffset){
        goalCell = getCellByCoords(data[dataOffset], data[dataOffset+1]);
        return goalCell;
        }
        public MapCell getStartCell(byte[] data, int dataOffset){
        startCell = getCellByCoords(data[dataOffset], data[dataOffset+1]);
        return startCell;
        }
        // Повертає вершину по її координатам
        public MapCell getCellByCoords(int row, int positionInRow){
        // Сзудзик для хешу ключа вершини графу
        int cellKey = row >= positionInRow ? row * row + row + positionInRow : row
+ positionInRow * positionInRow;
        if(touchedCells.containsKey(cellKey)){
        return touchedCells.get(cellKey);
        }
        //Знайдемо дані вершини в масиві
        int posOfRow = 0;
        int rowOffset = 0;
        while(posOfRow < row*2){
        rowOffset+=mapData[posOfRow+1];
        posOfRow+=2;
        }
        // починаємо з позиції індексу
        int dataIndex = (separator+1)+rowOffset+positionInRow;
```

```
MapCell cell = new MapCell(row, positionInRow, mapData[dataIndex]);
cell.rowLength = mapData[row*2+1];
cell.positionOffset = mapData[row*2];
touchedCells.put(cellKey, cell);
return cell;
}
// Ближні вершини
private Iterable<MapCell> getClosestCells(MapCell u){
HashSet<MapCell> set = new HashSet<MapCell>();
MapCell cell = null;
//ліва
if(u.positionInRow > 0){
cell = getCellByCoords(u.row, u.positionInRow-1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
//права
if(u.positionInRow+1 < u.rowLength){
cell = getCellByCoords(u.row, u.positionInRow+1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
//верхня
if(u.row > 0){
//верхня по центру
int topRowOffset = mapData[(u.row-1)*2];
int topRowPosition = u.positionInRow-topRowOffset+u.positionOffset;
if(topRowPosition < mapData[(u.row-1)*2+1] && topRowPosition >= 0){
cell = getCellByCoords(u.row-1, topRowPosition);
```

```
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
//верхня ліва
if(topRowPosition-1 < mapData[(u.row-1)*2+1] && topRowPosition-1 >=
0){
cell = getCellByCoords(u.row-1, topRowPosition-1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
//верхня права
if(topRowPosition+1 < mapData[(u.row-1)*2+1] && topRowPosition+1 >=
0){
cell = getCellByCoords(u.row-1, topRowPosition+1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
}
//нижні вершини
if(u.row*2+1+1 != separator){
//нижні по центру
int bottomRowOffset = mapData[(u.row+1)*2];
int bottomRowPosition = u.positionInRowbottomRowOffset+u.positionOffset;
if(bottomRowPosition < mapData[(u.row+1)*2+1] && bottomRowPosition
>= 0){
cell = getCellByCoords(u.row+1, bottomRowPosition);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
```

```
//нижня ліва
if(bottomRowPosition-1 < mapData[(u.row+1)*2+1] &&
bottomRowPosition-1 >= 0){
cell = getCellByCoords(u.row+1, bottomRowPosition-1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
//нижня права
if(bottomRowPosition+1 < mapData[(u.row+1)*2+1] &&
bottomRowPosition+1 >= 0){
cell = getCellByCoords(u.row+1, bottomRowPosition+1);
if(Math.abs(cell.cellValue) < 120)
set.add(cell);
}
}
return set;
}
// Вхідні вершини
public Iterable<MapCell> getPred(MapCell u){
return getClosestCells(u);
}
// Висхідні вершини
public Iterable<MapCell> getSucc(MapCell u){
return getClosestCells(u);
}
//Оновлення даних про вершини
private MapCell updateCellByCoords(int row, int positionInRow, byte
newCellValue){
cellCounter = 0;
```

```
int posOfRow = 0;
int rowOffset = 0;
while(posOfRow < row*2){
rowOffset+=mapData[posOfRow+1];
posOfRow+=2;
}
int dataIndex = (separator+1)+rowOffset+positionInRow;
//оновлення даних в масиві
mapData[dataIndex] = newCellValue;
//оновити вершину по позиціям
MapCell cell = getCellByCoords(row, positionInRow);
cell.cellValue = newCellValue;
return cell;
}
// оновити інформацію про вершини
public Iterable<MapCell> getUpdatedCells(byte[] updatedData, int offset){
HashSet<MapCell> set = new HashSet<MapCell>();
MapCell currentCell = null;
for(int i = offset; i < updatedData.length; i+=3){
// поміняти значення в масиві та в комірках
byte row = updatedData[i];
byte positionInRow = updatedData[i+1];
byte newValue = updatedData[i+2];
currentCell = updateCellByCoords(row, positionInRow, newValue);
set.add(currentCell);
}
return set;
}
// Серіалізація
```

```
private void writeObject(java.io.ObjectOutputStream stream)
 throws IOException {
 stream.writeObject(mapData);
 stream.writeObject(startCell);
 stream.writeObject(goalCell);
 stream.writeInt(separator);
 }
// Десеріадізація
private void readObject(java.io.ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
 mapData = (byte[]) stream.readObject();
 startCell = (MapCell) stream.readObject();
 goalCell = (MapCell) stream.readObject();
 separator = stream.readInt();
 }
}
```

Додаток А2

Код для інтерфейсів

```
// Інтерфейс алгоритмів
public interface IAlgorithm {
// Перерахунок алгоритмі, які доступні
public enum Algoritm{
DStarLite,
IDAStar
}
// Передає реалізацію IDistanceCost
void setCostFunction(IDistanceCost costFunction);
// Встановити граф
void setMap(MapData map);
// Початкова вершина
void setStartCell(MapCell startCell);
// Кінцева вершина
void setGoalCell(MapCell goalCell);
//Почати пошук
void findPath();
//Побудувати знайдений шлях
byte[] buildPath();
// Змінилось положення MPC
void updatePosition(MapCell newPosition);
// Змінилось значення вершин
void updateCellValue(MapCell cell, int newValue);
}
// Інтерфейс підрахунку ваги та евристики
public interface IDistanceCost {
// Перерахунок реалізацій, які доступні
```

```
public enum Cost{
ShortPath,
SmoothPath
}
// Еврістична функція
double getHeuristics(MapCell u, MapCell v);
//Вага ребра між вершинами u та v
double getEdgeCost(MapCell u, MapCell v);
}
```

Додаток А3

Код класів підрахунку ваги та евристичних функцій

```
// Клас для найкоротшого шляху по координатах
public class CellGroundDistanceCost implements IDistanceCost{
@Override
public double getHeuristics(MapCell u, MapCell v) {
// Діагональна відстань
int xDist = Math.abs(u.row-v.row);
int yDist = Math.abs(u.positionInRow-u.positionOffset - v.positionInRowv.positionOffset);
if(xDist > yDist){
return 1.4*yDist + (xDist-yDist);
}
else {
return 1.4*xDist + (yDist-xDist);
}
}
@Override
public double getEdgeCost(MapCell u, MapCell v) {
//по горизонталі вартість = 1
if(u.row == v.row){
return 1;
}
// по вертикалі вартість = 1
else if(u.positionInRow + u.positionOffset == v.positionInRow + v.positionOffset){
return 1;
}
// по діагоналі вартість = 1.4
```

```
else{
return 1.4;
}
}
}
// Клас для найкоротшого шляху по координатах з врахуванням висоти комірки
public class CellValueDistanceCost implements IDistanceCost{
@Override
public double getHeuristics(MapCell u, MapCell v) {
// Діагональна відстань
int xDist = Math.abs(u.row-v.row);
int yDist = Math.abs(u.positionInRow-u.positionOffset - v.positionInRowv.positionOffset);
if(xDist > yDist){
return 1.4*yDist + (xDist-yDist);
}
else {
return 1.4*xDist + (yDist-xDist);
}
}
@Override
public double getEdgeCost(MapCell u, MapCell v) {
// При модулі висоти > 120 комірка є непрохідною
if(Math.abs(v.cellValue) > 120){
return Double.POSITIVE_INFINITY;
}
// Враховуємо модуль різниці значень вершин
if(u.row == v.row){
```

```
return 1+Math.abs(u.cellValue - v.cellValue);
}
else if(u.positionInRow + u.positionOffset == v.positionInRow + v.positionOffset){
return 1+Math.abs(u.cellValue - v.cellValue);
}
else{
return 1+1.4*Math.abs(u.cellValue - v.cellValue);
}
}
}
```

Додаток А4

Код для фасадного класу

```
// Інтерфейсний клас модулю
public class PathStrategy{
private final IAlgorithm algorithm;
private MapData mapData;
//Конструктор базовий
//Використовує DStarLite та ShortPath за замовчуванням
public PathStrategy(Object control){
this(control, Algoritm.DStarLite, Cost.ShortPath);
}
//Повністю налаштовуваний конструктор
public PathStrategy(Object control, IAlgorithm.Algoritm algorithm,
IDistanceCost.Cost
distanceCost) {
this.control = control;
switch (algorithm) {
case IDAStar:
this.algorithm = new IDAStarAlgorithm();
break;
case DStarLite:
default:
this.algorithm = new DLiteAlgorithm();
break;
}
switch (distanceCost) {
case SmoothPath:
this.algorithm.setCostFunction(new CellValueDistanceCost());
break;
```

```
case ShortPath:
default:
this.algorithm.setCostFunction(new CellGroundDistanceCost());
break;
}
}
// Запускаємо пошук, передаємо ID карти
public void run(int mapId){
if(mapData == null){
loadData(mapId);
}
if(mapData != null){
algorithm.findPath();
createPath();
}
}
// Завершити роботу модуля. Зберегти дані останнього пошуку
public void exit(int code){
saveData();
}
// Передача даних
public boolean loadData(byte[] data) throws IOException {
switch (data[0]) {
//Надійшла карта
case 0:
mapData = new MapData(data, 2);
algorithm.setMap(mapData);
break;
//
```

```
case 1:
if(data[2] == 0){
//set start cell for current map
//mapData.setStartCell(data.getData(), 3);
algorithm.setStartCell(mapData.getStartCell(data, 3));
//algorithm.setStartCell(data.getData(), 3);
}
else if(data[2] == 1){
//set goal cell for current map
//mapData.setGoalCell(data.getData(), 3);
algorithm.setGoalCell(mapData.getGoalCell(data, 3));
//algorithm.setGoalCell(data.getData(), 3);
}
else if(data[2] == 2){
//new robot position with updated data for map
MapCell newPosition = mapData.getCellByCoords(data[3], data[4]);
algorithm.updatePosition(newPosition);
for(MapCell c : mapData.getUpdatedCells(data, 5)){
algorithm.updateCellValue(c, c.cellValue);
}
}
break;
default:
break;
}
return true;
}
private byte[] createPath(){
return algorithm.buildPath();
```

```
}
// Зберігаємо дані останньої карти
private void saveData(){
FileOutputStream fos;
try {
fos = new FileOutputStream(mapData.getMapID()+".search");
ObjectOutputStream oos = new ObjectOutputStream(fos);
//serialize MapData
oos.writeObject(mapData);
oos.flush();
oos.close();
fos.flush();
fos.close();
} catch (FileNotFoundException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
}
}
// Завантажуємо із збережених карту по її ID
private void loadData(int id){
FileInputStream fis;
try {
fis = new FileInputStream(id+".search");
ObjectInputStream oin = new ObjectInputStream(fis);
mapData = (MapData) oin.readObject();
oin.close();
fis.close();
} catch (FileNotFoundException e) {
```

```
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
}
}
}
```

Додаток А5

Код для класів алгоритмів

```
//Реалізація алгоритму IDA*
public class IDAStarAlgorithm implements IAlgorithm{
IDistanceCost costFunction; //евристика та вага ребер
private MapCell goal, start; //початкова та кінцева вершини
private MapData mapData; //граф
double g; // шлях старт->поточна
double f; // g + евристика
@Override
public void setCostFunction(IDistanceCost costFunction) {
this.costFunction = costFunction;
}
@Override
public void setMap(MapData map) {
this.mapData = map;
}
@Override
public void setStartCell(MapCell startCell) {
this.start = startCell;
}
@Override
public void setGoalCell(MapCell goalCell) {
this.goal = goalCell;
}
@Override
public void findPath() {
double result = performSearch(start);
if(result == Double.POSITIVE_INFINITY){
```

```
System.out.println("Path not found");
}
}
//Ініціалізація та початок пошуку
private double performSearch(MapCell startCell){
double max = costFunction.getHeuristics(start, goal);
while(true){
double t = search(start, 0, max, 0);
if(t == Double.NEGATIVE_INFINITY){
return max;
}
if( t == Double.POSITIVE_INFINITY){
return Double.POSITIVE_INFINITY;
}
max = t;
}
}
// Поточний шлях
private Map<MapCell, MapCell> pairs = new HashMap<MapCell, MapCell>();
// Розкриття вихідних вершин
private double search(MapCell current, double g, double max, double outLevel){
// обмеження глибини рекурсії
if(outLevel > max*2) return Double.POSITIVE_INFINITY
f = g + costFunction.getHeuristics(current, goal);
if(f > max) return f; //зберігаємо вартість шляху
if(goal == current){ //якщо дісталися до кінця
return Double.NEGATIVE_INFINITY;
}
```

```
double min = Double.POSITIVE_INFINITY;
for(MapCell v : mapData.getSucc(current)){
double t = search(v, g+costFunction.getEdgeCost(current, v), max, outLevel +
1);
if(t == Double.NEGATIVE_INFINITY){
//current це предок поточного по напрямку
// до фінішу, v - потомок
pairs.put(current, v);
return Double.NEGATIVE_INFINITY;
}
if (t < min) min = t;
}
return min; // повертаємо найкоротшу відстань
}
@Override
public byte[] buildPath() {
ByteArrayOutputStream bos = new ByteArrayOutputStream();
MapCell cell = start;
// Шляху немає, якщо у початкової вершини немає значення в колекції
if(pairs.get(cell) == null){
bos.reset();
String error = "Error 1";
bos.write(error.getBytes(), 0, error.length());
return bos.toByteArray();
}
//зберігаємо початкову вершину
bos.write(cell.row);
bos.write(cell.positionInRow);
while(true){
```

```
MapCell child = pairs.get(cell);
// зберігаємо вершину шляху
bos.write(child.row);
bos.write(child.positionInRow);
// Коли прийшли до кінця
if(child == goal){
// вихід із циклу
break;
}
cell = child;
}
return bos.toByteArray();
}
// Не інкрементний пошук, починаємо заново
@Override
public void updatePosition(MapCell newPosition) {
iteration = 0;
start = newPosition;
pairs.clear();
}
@Override
public void updateCellValue(MapCell cell, int newValue) {
// Не використовується
}
//Реалізація D* Lite
public class DLiteAlgorithm implements IAlgorithm{
private IDistanceCost costFunction;
private double km; //модифікатор для збереження пройденої відстані
private PriorityQueue<MapCell> queue; //збереження вершин для розгляду
```

```
private MapCell goal, f, last; // кінцева, поточна, остання змінена
private MapData mapData; //граф
//Ключ для сортування в приорітеті черги
private MapCell calcKey(MapCell cell){
cell.k1 = Math.min(cell.g, cell.rhs) + costFunction.getHeuristics(f, cell) + km;
cell.k2 = Math.min(cell.g, cell.rhs);
return cell;
}
private final void init(){
last = f;
queue = new PriorityQueue<MapCell>(30, new Comparator<MapCell>() {
@Override
public int compare(MapCell c1, MapCell c2) {
if(lessThan(c1, c2)) return -1;
else if(greaterThan(c1, c2)) return 1;
else return 0;
}
});
km = 0;
//для того, щоб вершина потрапила в чергу
// робимо її не насиченою
goal.rhs = 0;
queue.add(calcKey(goal));
}
//пробуємо оновити параметри вершини
private void updateVertex(MapCell u){
if(u != goal ){
//визначаємо rhs
u.rhs = getRHS(u);
```

```
}
if(queue.contains(u)){
queue.remove(u);
}
//якщо вершина не насичена
//додаємо в чергу,
// перераховуємо ключ
if(u.g != u.rhs){
queue.add(calcKey(u));
}
}
//Запуск алгоритму
private void computeShortestPath(){
while(lessThan(queue.peek(), calcKey(f)) || f.rhs != f.g){
double k1_Old = queue.peek().k1;
double k2_Old = queue.peek().k2;
MapCell u = queue.poll();
// Виконуємо порівняння із старим значенням
if( lessThan(k1_Old, k2_Old, calcKey(u)) ){
System.out.println("Key updated for "+ u);
queue.add(calcKey(u));
}
//Насичення
if(u.g > u.rhs){
u.g = u.rhs;
for(MapCell s : mapData.getPred(u)) updateVertex(s);
}
//шлях через вершину змінився
else{
```

```
u.g = Double.POSITIVE_INFINITY;
for(MapCell s : mapData.getPred(u)) updateVertex(s);
updateVertex(u);
}
}
}
//RHS значення вершини, пошук виконуємо серед висхідних
private double getRHS(MapCell cell){
double min = Double.POSITIVE_INFINITY;
for(MapCell s : mapData.getSucc(cell)){
double val = costFunction.getEdgeCost(cell, s) + s.g;
if(val < min) min = val;
}
return min;
}
// Частина компаратору черги
private boolean lessThan(MapCell c1, MapCell c2){
if(c1 == null){
System.out.println("Queue is empty!");
return false;
}
if (c1.k1 < c2.k1) return true;
else if (c1.k1 > c2.k1) return false;
return c1.k2 < c2.k2;
}
//для порівняння об’єкту вершини та значення ключів
private boolean lessThan(double c1k1, double c1k2, MapCell c2){
if (c1k1 < c2.k1) return true;
else if (c1k1 > c2.k1) return false;
```

```
return c1k2 < c2.k2;
}
// Частина компаратору черги
private boolean greaterThan(MapCell c1, MapCell c2){
if (c1.k1 < c2.k1) return false;
else if (c1.k1 > c2.k1) return true;
return c1.k2 < c2.k2;
}
@Override
public void setCostFunction(IDistanceCost costFunction) {
this.costFunction = costFunction;
}
@Override
public void setMap(MapData map) {
this.mapData = map;
}
@Override
public void setStartCell(MapCell startCell) {
this.f = startCell;
}
@Override
public void setGoalCell(MapCell goalCell) {
this.goal = goalCell;
}
@Override
public void findPath() {
if(queue == null){
init();
}
```

```
computeShortestPath();
}
@Override
public byte[] buildPath() {
ByteArrayOutputStream bos = new ByteArrayOutputStream();
MapCell cell = f;
while(cell != goal){
//Якщо чергова вершина не насичена, тоді шлях не знайдено.
if(cell.g == Double.POSITIVE_INFINITY){
//Повертаємо помилку
bos.reset();
String error = "Error 1";
bos.write(error.getBytes(), 0, error.length());
return bos.toByteArray();
}
//Пишемо вершини шляху
bos.write(cell.row);
bos.write(cell.positionInRow);
double min = Double.MAX_VALUE;
MapCell minCell = null;
for(MapCell c : mapData.getSucc(cell)){
double val = costFunction.getEdgeCost(cell, c) + c.g;
if(val < min){
min = val;
minCell = c;
}
}
// вибираємо висхідну
// з мінімальним сумарним g та перехід до неї
```

```
cell = minCell;
}
// Не забути про кінцеву вершину
bos.write(cell.row);
bos.write(cell.positionInRow);
return bos.toByteArray();
}
// при оновленні положення робота
// сумуємо пройдений шлях в km
// та змінюємо початкову вершину
@Override
public void updatePosition(MapCell newPosition) {
f = newPosition;
km = km + costFunction.getHeuristics(last, f);
last = f;
}
@Override
public void updateCellValue(MapCell cell, int newValue) {
// оновлюємо вершини,
// у випадку якщо вони перестали бути насиченими
updateVertex(cell);
}
}
```

Додаток Б

презентаційні матеріали

СИСТЕМА КЕРУВАННЯ ПЕРЕМІЩЕННЯМ РОБОТА

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

МЕТА

Останнім часом, так звані, мобільні роботизовані системи (МРС) та роботизовані комплекси починають відігравати все більшу роль при виконанні різних завдань без втручання людини-оператора. Для деяких завдань, таких як дослідження незнайомої місцевості, пошук та переміщення в небезпечних для людини ділянках, навіть створюються спеціальні вузько-спеціалізовані роботи.

Однією з головних задач, яку ставлять перед МРС, є задача коректного переміщення в робочому середовищі. Одні МРС здатні використовувати власні давачі для визначення інформації про стан навколишнього середовища, інші лише рухається за заздалегідь встановленим маршрутом, третім усю інформацію передають із спеціального центру керування. Так чи інакше, завжди є якась місцевість, і якась частина програмного забезпечення (ПЗ) має вирішувати завдання переміщення на цій місцевості.

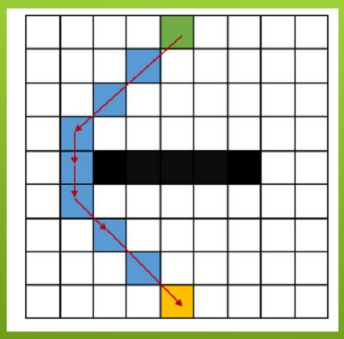
Для досягнення поставленої мети кваліфікаційної роботи бакалавра необхідно вирішити наступні завдання:

- розглянути задачу побудови траєкторії шляху МРС;
- сформулювати необхідні вимоги до системи;
- виконати аналіз алгоритмів пошуку оптимального шляху;
- розробити прототип програмної реалізації алгоритмів пошуку оптимального шляху;
- розробити програмний модуль на мові Java;
- виконати тестування створеного модулю та алгоритмів.

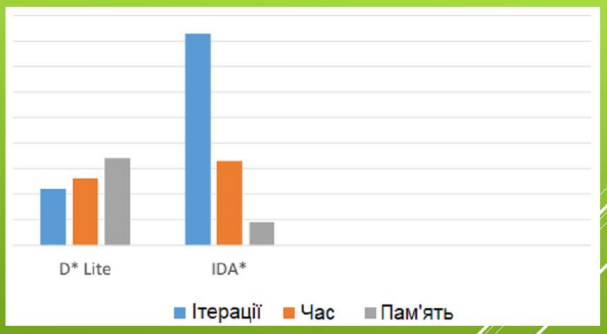
- Порівняння алгоритмів пошуку шляху МРС

Алгоритм	Інкрементний пошук	Евристичний пошук	Час на виконання	Використання пам'яті	Врахування положення МРС
A*	-	+	задовільно	погане	-
IDA*	-	+	погано	добре	-
D*	+	-	задовільно	задовільно	+
LPA*	+	+	добре	задовільно	-
D* Lite	+	+	добре	задовільно	+

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

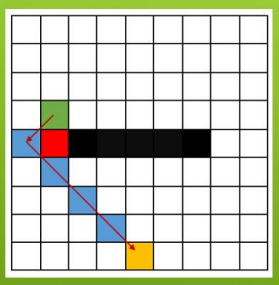


Загальний вигляд траєкторії шляху на карті із лінійними перешкодами

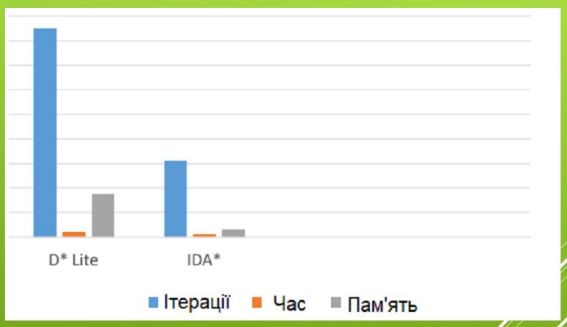


Графік порівняння роботи алгоритмів пошуку на карті із лінійною перешкодою

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

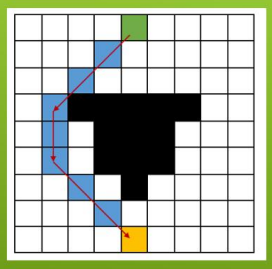


Зміна траєкторії шляху на карті із лінійною перешкодою

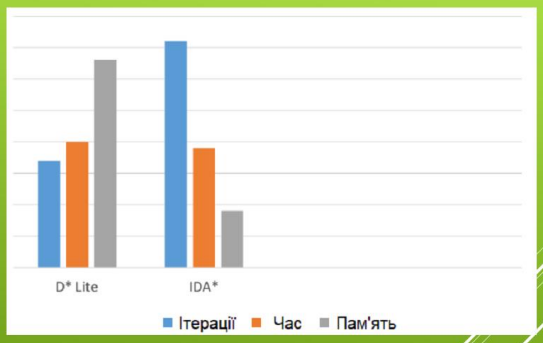


Графік порівняння роботи алгоритмів пошуку на карті з лінійною перешкодою після зміни карти

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

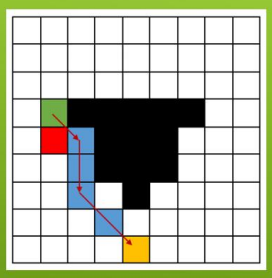


Загальний вигляд траєкторії шляху на карті з Т-подібною перешкодою

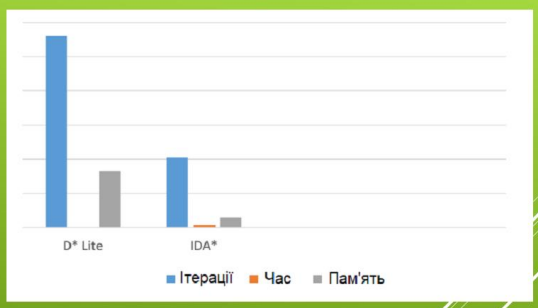


Графік порівняння роботи алгоритмів пошуку на карті з Т-подібною перешкодою

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

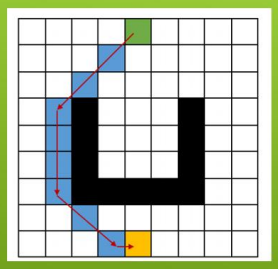


Зміна траєкторії шляху на карті з Т-подібною перешкодою



Графік порівняння роботи алгоритмів на карті з Т-подібною перешкодою після зміни карти

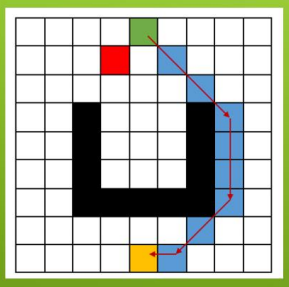
- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи



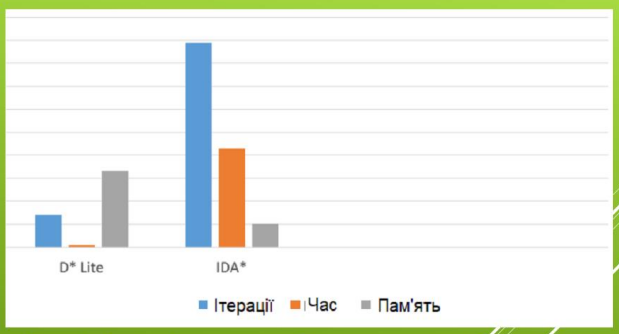
Загальний вигляд траєкторії шляху на карті з U-подібною перешкодою

Графік порівняння роботи алгоритмів пошуку на карті з U-подібною перешкодою

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

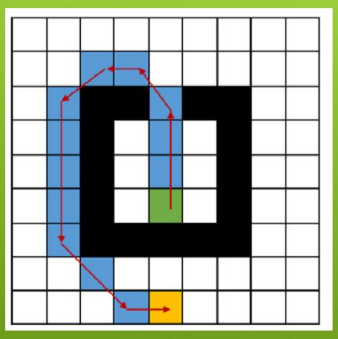


Зміна траєкторії шляху на карті з U-подібною перешкодою

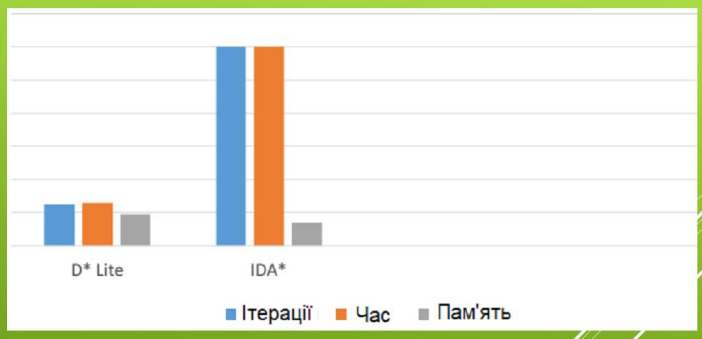


Графік порівняння роботи алгоритмів пошуку на карті з U-подібною перешкодою після зміни карти

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

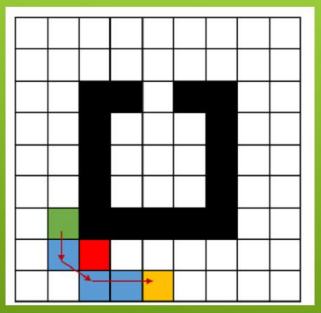


Загальний вигляд траєкторії шляху на карті з П-подібною перешкодою



Графік порівняння роботи алгоритмів пошуку на карті з П-подібною перешкодою

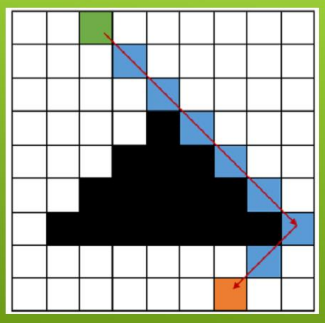
- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи



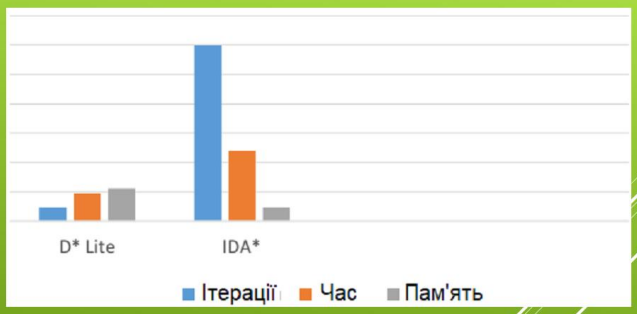
Зміна траєкторії шляху на карті з П-подібною перешкодою

Графік порівняння роботи алгоритмів пошуку на карті з П-подібною перешкодою після зміни карти

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

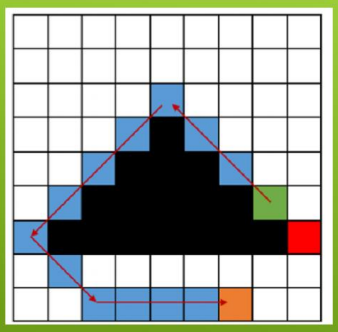


Загальний вигляд траєкторії шляху на карті з трикутною перешкодою



Графік порівняння роботи алгоритмів пошуку на карті з трикутною перешкодою

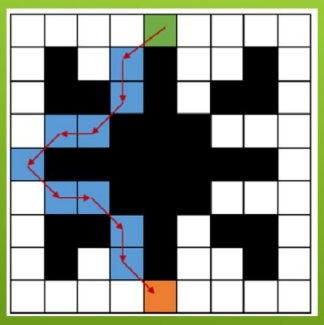
- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи



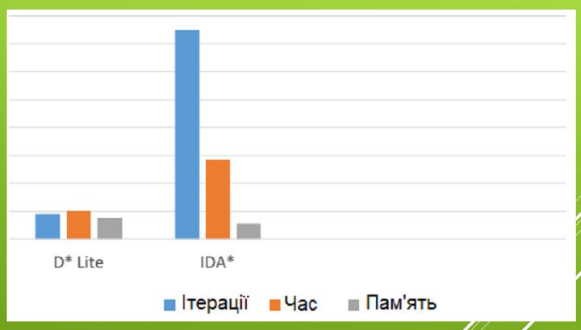
Зміна траєкторії шляху на карті з трикутною перешкодою

Графік порівняння роботи алгоритмів пошуку на карті з трикутною перешкодою після зміни карти

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

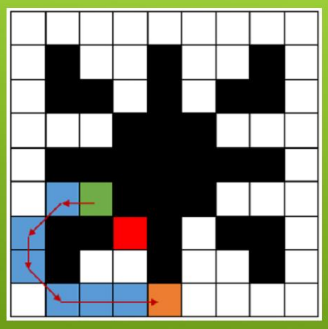


Загальний вигляд траєкторії шляху на карті з перешкодою у формі зірки

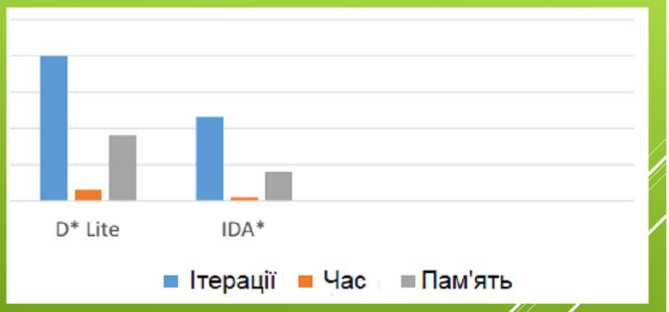


Графік порівняння роботи алгоритмів пошуку на карті з перешкодою у формі зірки

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

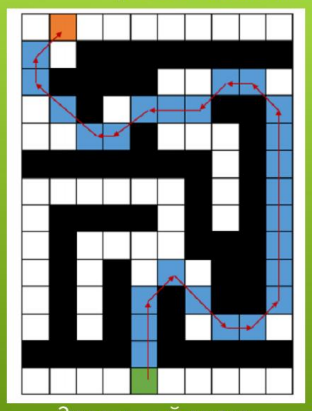


Зміна траєкторії шляху на карті з перешкодою у формі зірки

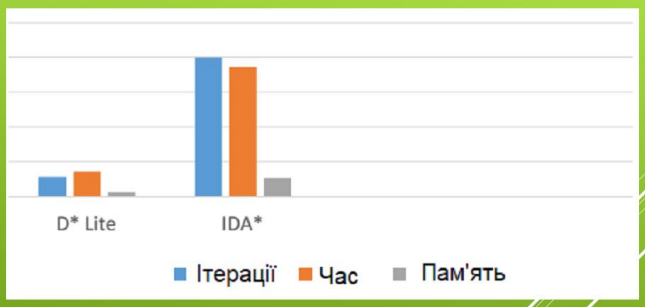


Графік порівняння роботи алгоритмів пошуку на карті з перешкодою у формі зірки після зміни карти

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

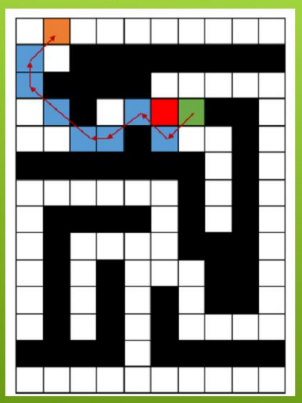


Загальний вигляд траєкторії шляху на карті з лабіринтом

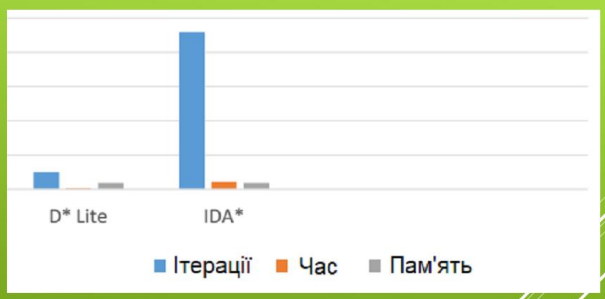


Графік порівняння роботи алгоритмів пошуку на карті з лабіринтом

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

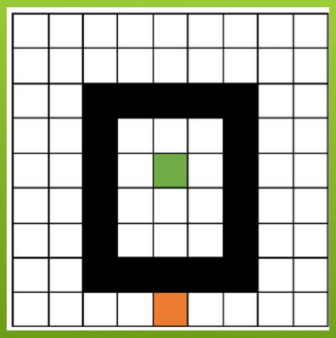


Зміна траєкторії шляху на карті з лабіринтом



Графік порівняння роботи алгоритмів пошуку на карті з лабіринтом після внесення зміни на карту

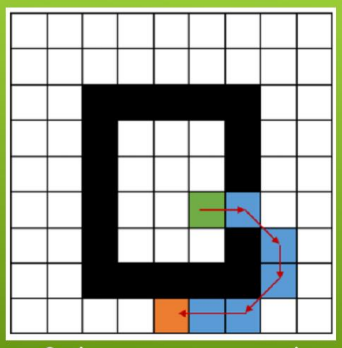
- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи



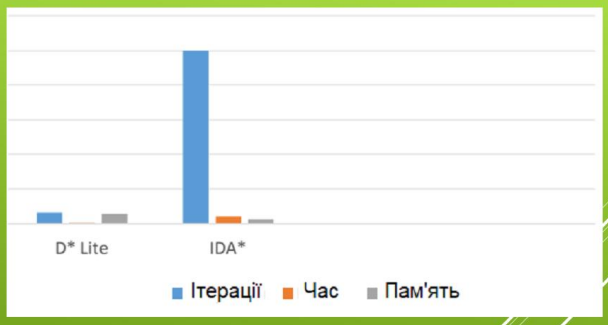
Загальний вигляд карти із непрохідною перешкодою

Графік порівняння роботи алгоритмів пошуку на карті з непрохідною перешкодою

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи



Зміна шляху на карті з початково непрохідною перешкодою

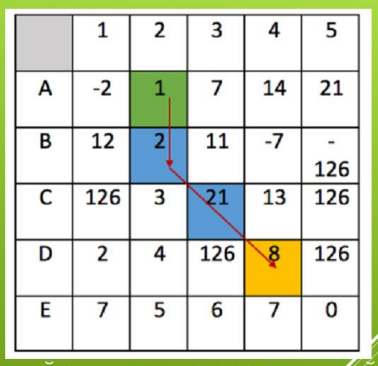


Графік порівняння роботи алгоритмів пошуку на карті з початково непрохідною перешкодою після внесення зміни на карту

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

	1	2	3	4	5
A	-2	1	7	14	21
B	12	2	11	-7	-126
C	126	3	21	13	126
D	2	4	126	8	126
E	7	5	6	7	0

Загальний вигляд тестової карти із зазначенням висот клітин

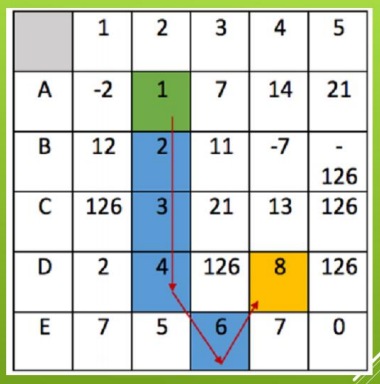


Загальний вигляд шляху, заснований на найкоротшій відстані

- Реалізація розробленого алгоритму переміщення мобільної роботизованої системи

	1	2	3	4	5
A	-2	1	7	14	21
B	12	2	11	-7	-126
C	126	3	21	13	126
D	2	4	126	8	126
E	7	5	6	7	0

Загальний вигляд тестової карти із зазначенням висот клітин



Загальний вигляд шляху, заснований на різниці висот

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Галіброда Владислав Олександрович

Тема: Система керування переміщенням робота

Спеціальність: 151 «Автоматизація та комп'ютерно-інтегровані технології»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 22 Кількість сторінок записки 71

1. Короткий зміст роботи та прийнятих рішень: створено модуль пошуку оптимального шляху на карті для системи керування переміщенням робота

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено огляд та аналіз існуючих алгоритмів пошуку оптимальної траєкторії руху робота на основі графів. Виконано постановку завдань для розробки алгоритму переміщення мобільного роботу. У другому розділі виконано обгрунтування вибору алгоритму пошуку, обрано за основу два алгоритми пошуку. Розроблено проєкт загальної архітектури всієї системи та загальну архітектуру модуля керування. Виконано підбір основних класів, всі розроблені коди програми наведені в додатках до пояснювальної записки. У третьому розділі Змодельовано роботу алгоритму переміщення із графом та роботу алгоритму пошуку оптимальної траєкторії руху з вагою ребер та евристичною функцією. Реалізовано розроблений алгоритм переміщення мобільної роботизованої системи, виконано порівняння двох використаних алгоритмів пошуку оптимальної траєкторії руху мобільної роботизованої системи.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: доцільно було б виконати експериментальні дослідження створеної системи керування

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. *Інші зауваження:* відсутні

9. Оцінка дипломної роботи: добре (B/4,25)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи)
Майдан Павло Сергійович, доцент каф. МАЕЕС ХНУ

"14" 06 2024 р.

(підпис)

Завідувачу кафедри АКІТтаР
д-ру техн.наук, проф. Мартинюку В.В.

Галіброда В.О.

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи АКІТс-21-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність плагіату ознайомлений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

01.06.2024
дата

підпис

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ АВТОМАТИЗАЦІЇ, КОМП'ЮТЕРНО-ІНТЕГРОВАНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Система керування переміщенням робота

Автор: Владислав ГАЛІБРОДА

Спеціальність: 151 Автоматизація та комп'ютерно-інтегрованих технологій

Освітня програма: Освітньо-професійна програма «Автоматизація та комп'ютерно-інтегровані технології»

Науковий керівник: к.т.н., доц. Денис МАКАРИШКІН

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, відомість документів), у структурі змісту, назвах розділів/підрозділів тощо, у назвах публікацій у переліку джерел посилання;

2) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

3) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 3,17% і адресується до 110 джерел, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Завідувач кафедри — Валерій МАРТИНЮК

Гарант освітньої програми — Юрій ФОРКУН

Керівник кваліфікаційної роботи — Денис МАКАРИШКІН

Thu Jun 13 16:53:02 EEST 2024, Федула Микола Васильович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 6%**

ID: 130299 Назва: БКР Система керування переміщенням робота Додано в БД: 2024-06-13 Автора: Владислав ГАЛІБРОДА Керівники: Денис МАКАРІШКІН Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	64537	972	1426 (2%)	18 (2%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK
by Turnitin

User name:
Кафедра АКІТіТК

Check ID:
1016357836

Check date:
13.06.2024 21:48:39 EEST

Check type:
Doc vs Internet + Library

Report date:
13.06.2024 21:59:13 EEST

User ID:
100005862

File name: **Галіброда_антиплаг**

Page count: **71** Word count: **11870** Character count: **85152** File size: **1.61 MB** File ID: **1016162148**

1274 words are marked as "ignored" and excluded from word count

Text modifications detected (similarity score might be affected)

3.17% Matches

Highest match: **1.89%** with Library source (File ID: **1016162155**)

1.36% Internet sources	110	Page 73
2.25% Library sources	37	Page 74

0% Quotes

Exclusion of quotes is off

Exclusion of references is off

0.05% Exclusions

Some exclusions were automatic (exclusion filters: matched word count less than **8 words** and **0%**)

0.01% Internet exclusions	10	Page 75
0.04% Library exclusions	2	Page 75

Modifind

Text modifications detected. Find more details in the online report.

Replaced characters	3
Suspicious formatting	11 Pages