Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Розробка компонентів мультиагентної системи пошуку шляху у динамічних середах
(тема)

Виконав:
студент 2 курсу, групи СШІм-19-2
Дворцов Д. В.
(прізвище, ініціали)

Спеціальність 122 Комп’ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник доц. Шергін В.Л.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ В.О. Філатов
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп’ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____» ___________ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Дворцов Денис Володимирович
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка компонентів мультиагентної системи пошуку шляху у динамічних середах

затверджена наказом університету від 29 03 20 21 р. № 390Ст

2. Термін подання студентом роботи до екзаменаційної комісії _____ ___________ 20 ___ р.

3. Вихідні дані до роботи Функція: розробка шаблону гри у жанрі покрокової стратегії з реалізованим мультиагентним алгоритмом пошуку шляхів для ігрових персонажів. Перелік використовуваних програмних засобів: ОС Microsoft Windows 7 та вище, інтегроване середовище програмування Jet Brains Rider, Ігровий движок Unity 3D версії 2018 та вище. Технічне забезпечення: комп’ютер для запуску програми з не менш, ніж 1Гб оперативної пам’яті.

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі та формалізована постановка задачі

2) Опис алгоритму розв'язання задачі

3) Програмна реалізація

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)________________

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Отримання завдання на кваліфікаційну роботу	29.03.2021	виконано
2.	Аналіз завдання, пошук літератури та аналогів з теми	30.03.2021 - 03.04.2021	виконано
3.	Опрацювання літератури та аналіз об’єкту	03.04.2021 - 10.04.2021	виконано
	дослідження		
4.	Проектування програмного засобу	10.04.2021 - 12.04.2021	виконано
5.	Вибір програмних, мовних та технічних засобів для	12.04.2021 - 15.04.2021	виконано
	розробки системи		
6.	Розробка програмного засобу	15.04.2021 - 20.04.2021	виконано
7.	Аналіз результатів, отриманих за допомогою	20.04.2021 - 22.04.2021	виконано
	програмного засобу		
8.	Оформлення пояснювальної записки та програмної	22.04.2021 - 24.04.2021	виконано
	документації		
9.	Оформлення графічної частини та презентаційних	24.04.2021 - 25.04.2021	виконано
	матеріалів комп’ютерного захисту		
10.	Попередній захист	14.05.2021	
11.	Захист перед ЕК	19.05.2021	

Дата видачі завдання 29 березня 20 21 р.

Студент ________________________________
(підпис)

Керівник роботи ________________________ ________________________
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 103 с., 14 рис., 2 дод., 23 джерела

БЕЗПЕРЕРВНЕ ПЛАНУВАННЯ, ДОВІЧНЕ ПЛАНУВАННЯ, ЕВРИСТИЧНИЙ ПОШУК, ІНКРЕМЕНТНИЙ ПОШУК, МУЛЬТИАГЕНТНИЙ ПОШУК, ПЕРЕПЛАНУВАННЯ, ПОВТОРНЕ ВИКОРИСТАННЯ ПЛАНУ, ПОШУК НА ОСНОВІ ЕВРИСТИЧНОГО ПЛАНУВАННЯ, A*, CBS, D* LITE, MAPF.

Об'єкт дослідження – шаблон гри у жанрі покрокової стратегії з реалізованим мультиагентним алгоритмом пошуку шляхів для ігрових персонажів.

Мета роботи – розробка програмного засобу, що по команді користувача на переміщення персонажів виконується пошук оптимального путі до нової плитки ігрової карти, враховуючи переміщення інших персонажів на мапі.

Методи дослідження – методи використання мультиагентних алгоритмів, методи проектування ігрових додатків, методи об'єктно-орієнтованої розробки.

Результати роботи – розроблений програмний засіб, що надає користувачу можливість переміщати ігрових персонажів по згенерованій мапі з урахуванням поставлених обмежень, врахування переміщення інших персонажів. Також, для спрощення оцінки результату реалізована програма візуально відображає результати кожного виконання алгоритму.

Область застосування – програмний продукт використовується для створення гри у жанрі «покрокова стратегія».

РЕФЕРАТ

Пояснительная записка: 103 с., 14 рис., 2 доп., 23 источника

ИНКРЕМЕНТНЫЙ ПОИСК, МУЛЬТИАГЕНТНЫЙ ПОИСК, НЕПРЕРЫВНОЕ ПЛАНИРОВАНИЕ, ПЕРЕПЛАНИРОВКА, ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ ПЛАНА, ПОЖИЗНЕННОЕ ПЛАНИРОВАНИЕ, ПОИСК НА ОСНОВЕ ЭВРИСТИЧЕСКОГО ПЛАНИРОВАНИЯ, ЭВРИСТИЧЕСКИЙ ПОИСК, A*, CBS, D* LITE, MAPF.

Объект исследования – шаблон игры в жанре пошаговой стратегии с реализованным мультиагентные алгоритмом поиска путей для игровых персонажей. Цель работы – разработка программного средства, по команде пользователя на перемещение персонажей выполняется поиск оптимального пути к новой плитки игровой карты, учитывая перемещения других персонажей на карте.

Методы исследования – методы использования мультиагентных алгоритмов, методы проектирования игровых приложений, методы объектно-ориентированной разработки.

Результаты работы – разработан программное средство, оказывает пользователю возможность перемещать игровых персонажей по сгенерированной карте с учетом поставленных ограничений, учета перемещения других персонажей. Также, для упрощения оценки результата реализована программа визуально отображает результаты каждого выполнения алгоритма.

Область применения – программный продукт используется для создания игры в жанре «пошаговая стратегия».

ABSTRACT

Explanatory note: 103 p., 14 figs., 2 apps., 23 sources

A*, CBS, CONTINUOUS PLANNING, D* LITE, HEURISTIC SEARCH, INCREMENTAL SEARCH, LIFE PLANNING, MAPF, MULTI SEARCH, REDEVELOPMENT, RE-USE PLAN, SEARCH BASED HEURISTIC PLANNING.

The object of research is a game template in the genre of step-by-step strategy with an implemented multi-agent algorithm for finding ways for game characters.

The purpose of the work is to develop a software that, on the user's command to move the characters, searches for the optimal path to the new tile of the game card, taking into account the movement of other characters on the map.

Research methods – methods of using multi-agent algorithms, methods of designing game applications, methods of object-oriented development.

The results of the work – a developed software that allows the user to move game characters on the generated map, taking into account the restrictions, taking into account the movement of other characters. Also, to simplify the evaluation of the result, the implemented program visually displays the results of each execution of the algorithm.

Scope – the software product is used to create a game in the genre of "step-by-step strategy".

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 9
Вступ 10
1 Аналіз предметної галузі та формалізована постановка задачі 11
1.1 Огляд і аналіз сучасного стану проблеми, а також існуючих методів і засобів вирішення стандартної задачі пошуку шляху для одного агента 11
1.2 Опис недоліків існуючих базових алгоритмів вирішення задачі одноагентного пошуку шляху та варіанти їх модифікацій 18
1.3 Опис класичної задачі MAPF 20
1.3.1 Типи конфліктів у класичному MAPF 21
1.3.2 Поведінка агента у цілі в класичному MAPF 23
1.3.3 Цільові функції в класичному MAPF 23
1.3.4 MAPF на зважених графах 24
1.3.5 Завдання та агенти 26
1.4 Опис існуючих алгоритмів вирішення задачі MAPF 27
1.5 Постановка задачі 29
2 Дослідження методів розв’язання задачі 31
2.1 Дослідження алгоритму високого рівня 31
2.1.1 Приклад використання алгоритму CBS 34
2.2 Дослідження алгоритму низького рівня 36
2.2.1 Математичний опис алгоритму LPA* 37
2.2.2 Математичний опис алгоритму D* Lite 42
2.2.3 Оптимізація алгоритму D* Lite 46
2.2.4 Практичне застосування алгоритму D* Lite 49
3 Програмна реалізація 51
3.1 Вибір і обґрунтування мовних і програмних засобів 51
3.1.1 Опис інструментального засобу Unity 3D 51
3.1.2 Опис мовного засобу C# 52
3.1.3 Опис інструментального засобу Rider 52

3.2 Опис структури програми 53
3.3 Опис коду алгоритму пошуку шляху 56
3.4 Опис інтерфейсу розробленої програми 65
3.5 Інтерпретація результатів 68
Висновки 71
Перелік джерел посилання 72
Додаток А Текст програми 74
Додаток Б Відомість кваліфікаційної роботи магістра 103

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Агент пошуку – сутність, у задачі пошуку шляху, для якої треба розробити шлях;

Евристика – в області комп'ютерних наук, штучного інтелекту та математичної оптимізації, евристика – це техніка, розроблена для пришвидшення вирішення проблем, коли класичні методи занадто повільні, або для пошуку наближеного рішення, коли класичні методи не знайшли жодного точного рішення. Це досягається за рахунок зменшення оптимальної, повноти або точності в обмін на швидкість;

Плитка – атомарна частина карти в покрокових стратегіях;

TBS – Покрокова стратегія – Turn based strategy – стратегічна гра, в якій гравці грають по черзі. Це відрізняється від RTS – стратегії реального часу – Real time strategy, в якій всі гравці грають одночасно.

ВСТУП

Комп'ютерні ігри можуть бути відмінним полігоном для дослідження штучного інтелекту (ШІ), будучи середнім між синтетичними, високо абстрагованими академічними тестами, та більш складними проблемами з реального життя. Серед багатьох методик та проблем, пов'язаних з ШІ, таких як навчання, планування та обробка природною мовою, пошук шляху виступає як один з найпоширеніших видів дослідження ШІ в іграх [1].

Незважаючи на значний прогрес, що спостерігався в останні роки, проблема пошуку шляху продовжує постійно привертати увагу дослідників, що частково пояснюється високими вимогами до продуктивності. Рухи повинні обчислюватися в реальному часі, і часто існує багато мобільних юнитів для яких треба обчислити шляхи. Шляхи повинні виглядати досить реалістично для користувача. Крім більш стандартного пошуку шляхів на повністю відомій статичній карті, ігри мають більш складні варіації проблеми, такі як динамічні зміни в середовищі, різнорідні території та ігрові персонажі, відсутність повної інформації та комбінації цих проблем.

Ранні розв'язки проблеми пошуку шляхів у комп'ютерних іграх, такі як глибинний пошук, ітеративне поглиблення, пошук по ширині, алгоритм Дейкстри, алгоритм A*, скоро були перевантажені експоненціальним зростанням складності ігор. Необхідні більш ефективні рішення для того, щоб мати можливість вирішувати проблеми пошуку шляхів на більш складній обстановці з обмеженим часом і ресурсами. Через величезний успіх алгоритму A* у пошуку шляхів багато дослідників сподіваються на прискорення A*, щоб задовольнити мінливі потреби ігор [2]. Протягом останніх десятиліть було зроблено значні зусилля для оптимізації цього алгоритму і успішно впроваджено десятки перероблених алгоритмів. Прикладами таких оптимізацій є вдосконалення евристичних методів, оптимізація відображення карт, впровадження нових структур даних і скорочення вимог до пам'яті.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ФОРМАЛІЗОВАНА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд і аналіз сучасного стану проблеми, а також існуючих методів і засобів вирішення стандартної задачі пошуку шляху для одного агента

Дослідження шляхів зазвичай стосується пошуку найкоротшого шляху між двома кінцевими точками. Прикладами таких проблем є планування транзиту, маршрутизація телефонного трафіку, навігація по лабіринту і планування шляху для роботів. Із ростом важливості ігрової індустрії, пошук шляхів став популярною проблемою в ігровій індустрії. Ігри таких жанрів як рольові ігри та стратегії, часто мають персонажів, відправлених на місії з їхнього поточного місця розташування до зумовленого або визначеного гравцем місця призначення. Найбільш поширеною проблемою пошуку шляхів у відеоіграх є те, як розумно уникнути перешкод і шукати найефективніший шлях через різну місцевість [5].

У контексті відеоігор оптимальним шляхом між будь-якими двома місцями є шлях найменшої вартості, а не найкоротший шлях . Загальне рішення полягає в накладанні іншої карти з інформацією про вартість на реальний ігровий світ і прийняття на ньому алгоритму пошуку, щоб знайти шлях найменшої вартості. Поточні рішення для пошуку шляхів, у контексті відеоігор, або забезпечують високу швидкість пошуку, жертвуючи точністю або створюючи оптимальний шлях, але використовуючи більше часу та ресурсів [3]. Ефективний пошук оптимального шляху, все ще залишається областю досліджень [4].

Три ключові елементи базового підходу до пошуку шляхів – представлення карти у вигляді графа, алгоритм пошуку та евристична функція, яка спрямовує пошук. Мережеві карти є популярним способом перетворення ігрової карти в пошуковий граф. Як частина процесу, карта розділена на атомарні клітини, іноді звані плитками. Залежно від початкової топології карти, плитка бути позначена

прохідною або заблокованою. Ігровий персонаж може займати одну прохідну плитку одночасно. Прохідні плитки стають вузлами в пошуковому графі. Ребра графів з'єднують сусідні прохідні плитки.

За своєю суттю, метод пошуку шляхів проводить пошук у графі, починаючи з однієї вершини і досліджуючи сусідні вузли, поки не буде досягнуто вузол призначення, як правило, з метою пошуку «найдешевшого» маршруту. Хоча методи пошуку графа, такі як пошук у ширину, знайдуть маршрут, якщо надається достатньо часу, інші методи, які «досліджують» графік, мають тенденцію досягати місця призначення раніше. Аналогією буде людина, що йде по кімнаті; замість того, щоб вивчати всі можливі маршрути заздалегідь, людина звичайно ходить у напрямку до місця призначення і лише відхиляється від шляху, щоб уникнути перешкод, і робити відхилення якнайменше.

Двома основними проблемами пошуку шляхів є:

– пошук шляху між двома вузлами в графі;

– пошук оптимального найкоротшого шляху.

Прості алгоритми, такі як пошук у ширину та глибинну, вирішують першу проблему, вичерпуючи всі можливості; починаючи з даного вузла, вони перебирають всі потенційні шляхи, поки вони не досягнуть цільового вузла. Ці алгоритми виконуються в лінійний час:

$$O(|V| + |E|), \tag{1.1}$$

де V – кількість вершин,

E – кількість ребер між вершинами.

Більш складною проблемою є пошук оптимального шляху. Вичерпний підхід у цьому випадку називається алгоритмом Беллмана–Форда, який дає часову складність або квадратичного часу. Проте, не потрібно вивчати всі можливі шляхи пошуку оптимального. Такі алгоритми, як алгоритм A* і

Дейкстри, стратегічно усувають шляхи, або через евристику, або через динамічне програмування. Усуваючи неможливі шляхи, ці алгоритми можуть досягти тимчасової складності

$$O(|E| * log(|V|)) \text{ [6].} \tag{1.2}$$

Відомим прикладом алгоритму пошуку на основі графів є алгоритм Дейкстри. Цей алгоритм, поданий у лістингу 1.1, починається з початкового вузла і «відкритого набору» вузлів кандидатів. На кожному кроці розглядається вузол відкритого набору з найменшою відстанню від початку. Вузол позначений як «закритий», і всі вузли, прилеглі до нього, додаються до відкритого набору, якщо вони ще не були досліджені. Цей процес повторюється, поки не буде знайдено шлях до пункту призначення. Оскільки спочатку розглядаються вузли найнижчих відстаней, перший раз, коли знайдено призначення, шлях до нього буде найкоротшим.

Лістинг 1.1 – Псевдокод алгоритму Дейкстри

```
procedure UniformCostSearch(Graph, start, goal)
  node := start
  cost := 0
  frontier := priority queue containing node only
  explored := empty set
  do
    if frontier is empty return failure
    node := frontier.pop()
    if node is goal return solution
    explored.add(node)
    for each of node's neighbors n
      if n is not in explored
        frontier.add(n)
```

Алгоритм Дейкстри не працює, якщо є вершина з негативною вагою. У гіпотетичній ситуації, коли вузли A, B і C утворюють пов'язаний неорієнтований граф з ребрами AB = 3, AC = 4 і BC = −2, оптимальний шлях до C коштує 1, а оптимальний шлях від до B коштує 2. Алгоритм Дейкстри, починаючи з A, спочатку розгляне B, оскільки він є найбільш близьким. Вона призначить їй вартість 3 і позначить її закритою, тобто її вартість ніколи не буде переоцінена. Таким чином, Дейкстри не може оцінити негативні граничні ваги.

A* – варіант алгоритму Дейкстри, який часто використовується в іграх. A* Призначає вагу кожному відкритому вузлу, рівному вазі краю до цього вузла, плюс приблизну відстань між цим вузлом і фінішем. Ця приблизна відстань виявляється евристикою і являє собою мінімально можливу відстань між цим вузлом і кінцем. Це дозволяє виключити більш довгі шляхи після того, як буде знайдено початковий шлях. Якщо між стартовим і кінцевим контуром існує шлях довжини x, а мінімальна відстань між вузлом і фінішною точкою перевищує x, цей вузол не потрібно перевіряти [8].

A* використовує цю евристику для поліпшення поведінки порівняно з алгоритмом Дейкстри. Коли евристика дорівнює нулю, A* еквівалентний алгоритму Дейкстри. Оскільки евристична оцінка зростає і наближається до істинної відстані, A* продовжує знаходити оптимальні шляхи, але працює швидше (завдяки вивченню меншої кількості вузлів). A* з допустимою евристикою повертає оптимальні рішення. Відстань Манхеттена – добре відома допустима евристика. Дано два вузли з координатами (x1, y1) і (x2, y2), відстань Манхеттена визначається як

$$\Delta x + \Delta y, \tag{1.3}$$

де $\Delta x = |x1 - x2|$,

$\Delta y = |y1 - y2|$.

На карті без перешкод, відстань Манхеттена збігається з істинною відстанню. Коли значенням евристики є рівно істинна відстань, A* розглядає найменшу кількість вузлів. (Однак, як правило, недоцільно писати евристичну функцію, яка завжди обчислює справжню відстань.) Оскільки значення евристики зростає, A* досліджує менше вузлів, але більше не гарантує оптимального шляху. У багатьох додатках (таких як відеоігри) це прийнято і навіть бажано, щоб алгоритм працював швидко. Евристики, які не переоцінюють справжню відстань до мети, називають допустимими. Псевдокод алгоритму A* поданий у лістингу 1.2.

Лістинг 1.2 – Псевдокод алгоритму A*

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path

function A_Star(start, goal)
    closedSet := {}
    openSet := {start}
    cameFrom := an empty map
    gScore := map with default value of Infinity
    gScore[start] := 0
    fScore := map with default value of Infinity
    fScore[start] := heuristic_cost_estimate(start, goal)
    while openSet is not empty
        current := the node in openSet having the lowest
fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
```

Продовження лістингу 1.2

```
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue
            tentative_gScore := gScore[current] +
dist_between(current, neighbor)
            if neighbor not in openSet
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] +
heuristic_cost_estimate(neighbor, goal)
```

Алгоритм А* є де-факто стандартом в пошуку шляхів. Він прийшов на заміну алгоритму Дейкстри і є по суті його модифікацією.

Як популярний алгоритм пошуку шляхів в ігровій індустрії, алгоритм A* був застосований до великого числа комп'ютерних ігор. Хоча сам алгоритм легко зрозуміти, реалізація в реальній комп'ютерній грі нетривіальна.

Серія ігор Civilization є класикою покрокових стратегій. Починаючи із п’ятої частини у ній використовується алгоритм A* для пошуку шляху на карті, представленої у вигляді гексагональних плиток. Алгоритм пошуку шляхів застосовується для керування переміщенням ігрових персонажів до потрібного місця через групу прохідних плиток [5]. Інтерфейс пошуку шляху гри Civilization VI показано на рисунку 1.1.

Ще одним прикладом використання A* є «A* Pathfinding Project» – пакет для ігрового движка Unity 3D що являє собою фреймворк для пошуку шляхів, використовуваний у іграх різного типу. Інтерфейс налаштування A* Pathfinding Project показано на рисунку 1.2.

Рисунок 1.1 – Інтерфейс пошуку шляху у Civilization VI

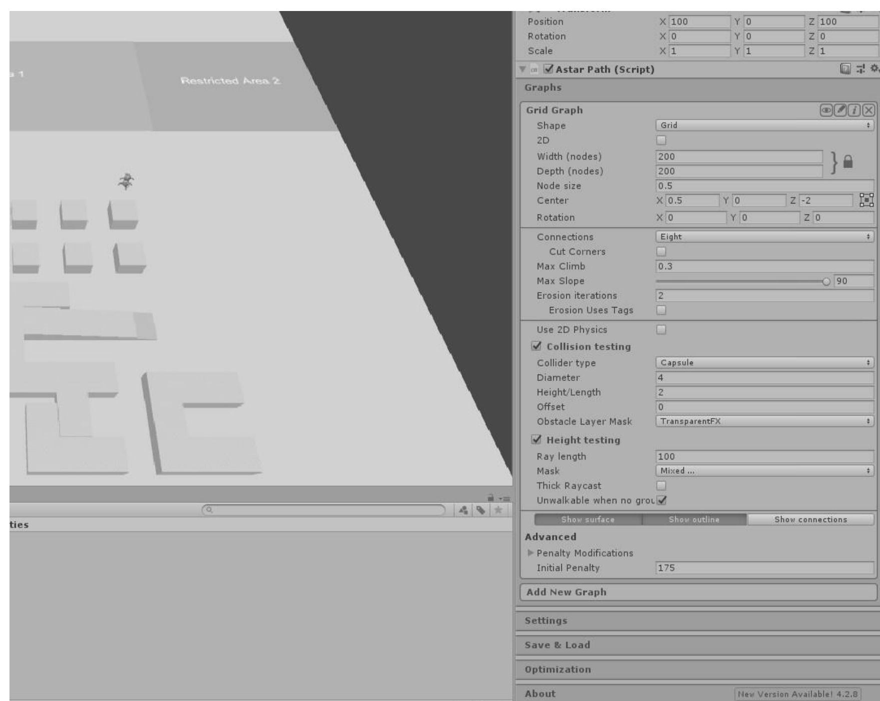

Рисунок 1.2 – Інтерфейс налаштування A* Pathfinding Project

1.2 Опис недоліків існуючих базових алгоритмів вирішення задачі одноагентного пошуку шляху та варіанти їх модифікацій

Незважаючи на те що алгоритм A* є швидшим за такі алгоритми як пошук в ширину чи алгоритм Дейкстри, його швидкості може не вистачати для графів великого розміру. Це гарно видно на прикладі останніх Civilization – де заради швидкості пошук шляху не завжди вибирає оптимальний маршрут.

Одним із рішень проблеми швидкості, яке використовується в тому числі у A* Pathfinding Project є попередня генерація карти шляхів. Це дозволяє за рахунок збільшення часу попереднього завантаження карти мати більшу швидкість пошуку. Недоліком цього методу є те, що він працює лише на статичних, або майже статичних картах, через те що кожна зміна у карті призводить до перегенерації карти шляхів, що є витратною операцією. Через цей недолік даний метод не підходить для покрокових стратегій, адже їх карти постійно змінюються.

Іншим підходом до оптимізації є ієрархічний пошук. Карта розподіляється на кластери. На слої високого рівня планується шлях між кластерами. Після того, як план був знайдений, другий шлях планується в межах кластера на нижньому рівні [9]. Перевага полягає в тому, що кількість вузлів зменшується і алгоритм виконується швидко. Недоліком є те, що ієрархічний планувальник шляхів важко реалізувати [10]. Для ієрархічного пошуку шляхів можна використовувати квадратичні дерева. Ця ідея була вперше описала в індустрії відеоігор, яка мала потребу в плануванні на великих картах з ефективним використанням процесорного часу. Подібною методикою є навігаційні сітки (navmesh), які використовуються для геометричного планування в іграх і планування мультимодальних перевезень, які використовуються в задачах комівояжера з більш ніж одним транспортним засобом. Ієрархічний пошук може бути гарним рішенням для ігор, мапи яких складаються з кількох з’єднаних частин, але не

підходять для покрокових стратегій, адже їх карти майже неможливо розділити на кластери.

Рішення які більш підходять для покрокових стратегій, використовують ітеративний пошук шляху. Більшість досліджень з пошуку вирішують проблеми одноразового пошуку. Проте часто доводиться вирішувати ряд подібних пошукових завдань, оскільки стан об’єктів або знання про їх стан змінюються. Одним із прикладів ітеративних алгоритмів пошуку є Adaptive A*, який відрізняється від оригінального A* тим, що при повторних пошуках використовує евристику, основану на результаті попереднього пошуку.

Adaptive A* дає можливість використовувати інформацію з попередніх пошуків, але він все ще передбачає повторний аналіз усіх елементів графу, що не дає можливості отримати значний приріст швидкості виконання. Цю проблему вирішують алгоритми сімейства D*:

– Оригінальний D*, автор Ентоні Стенц, є інформованим інкрементним алгоритмом пошуку;

– Орієнтований D* – інформований інкрементний алгоритм евристичного пошуку Ентоні Стенца, який поєднує ідеї A* і оригіналу D*. Фокусований D* був результатом подальшого розвитку оригінального D*;

– D* Lite – це алгоритм почергового евристичного пошуку Свена Кеніга і Максима Ліхачова, який базується на LPA*, інкрементному алгоритмі евристичного пошуку, який поєднує ідеї A* і Dynamic SWSF-FP (strict weak superior function – fixed-point).

Всі три алгоритми пошуку вирішують одні й ті ж проблеми планування шляхів на основі допущення, включаючи планування з припущенням вільного простору [13], де агент пошуку повинен перейти до заданих координат цілі в невідомому рельєфі. Він робить припущення щодо невідомої частини місцевості (наприклад, що вона не містить перешкод) і знаходить найкоротший шлях від його поточних координат до координат цілі під цими припущеннями. Потім агент йде по шляху. При спостереженні нової картографічної

інформації (наприклад, раніше невідомих перешкод) він додає інформацію до своєї карти і, при необхідності, поновлює новий найкоротший шлях від поточних координат до заданих координат мети. Він повторює процес, поки не досягне мети або не визначить, що координати цілі не можуть бути досягнуті. При перетині невідомої місцевості часто можна виявити нові перешкоди, так що це перепланування має бути швидким. Інкрементні (евристичні) алгоритми пошуку прискорюють пошук послідовностей подібних пошукових завдань, використовуючи досвід попередніх завдань для прискорення пошуку поточного. Припускаючи, що координати цілі не змінюються, всі три алгоритми пошуку є більш ефективними, ніж повторні пошуки A* [11].

1.3 Опис класичної задачі MAPF

MAPF – важливий тип мультиагентної проблеми планування, в якій завдання полягає в плануванні шляхів для кількох агентів, де ключовим обмеженням є те, що агенти зможуть одночасно йти цими шляхами, не стикаючись між собою. MAPF має ряд відповідних сучасних застосувань, включаючи автоматизовані склади, автономні транспортні засоби та робототехніку. Отже, за останні роки на цю проблему звернули увагу різні дослідницькі групи та академічні спільноти [18].

Спочатку опишемо те, що називається класичною проблемою MAPF. Вхідними даними класичної задачі MAPF з k агентами є ненаправлений граф та набір агентів з начальними на цільовими вершинами.

Час вважається дискретизованим, і на кожному часовому кроці кожен агент знаходиться в одній з вершин графа і може виконувати одну дію. Дія в класичному MAPF – це функція a: V → V, така що a (v) = v’ означає, що якщо агент знаходиться у вершині v і виконує a, то він буде у вершині v’ на наступному кроці часу.

Кожен агент має два типи дій: чекай і рухайся. Дія очікування означає, що агент залишається в поточній вершині ще один крок часу. Дія переміщення означає, що агент рухається від поточної вершини v до сусідньої вершини v’ на графіку.

Для послідовності дій π та агента i ми позначаємо $\pi_i[x]$ місце розташування агента після виконання перших x дій у π, починаючи з джерела агента s(i).

Послідовність дій π є одноагентним планом для агента i, якщо виконання цієї послідовності дій Sy s(i) призводить до знаходження на t(i). Рішення – це набір k одноагентних планів, по одному для кожного агента.

1.3.1 Типи конфліктів у класичному MAPF

Основною метою вирішувачів MAPF є пошук рішення, тобто плану з одним агентом для кожного агента, який може бути виконаний без зіткнень. Для досягнення цього вирішувачі MAPF використовують поняття конфліктів під час планування, де рішення MAPF називається дійсним, якщо між двома одноагентними планами не існує конфлікту. Визначення того, що являє собою конфлікт, залежить від навколишнього середовища, і відповідно література про класичний MAPF включає кілька різних визначень того, що становить конфлікт між планами [18].

Нижче перечислені загальні визначення конфліктів. Нехай π_i та π_j – пара одноагентних планів:

– конфлікт вершин – конфлікт між π_i та π_j виникає тоді, коли згідно з цими планами агенти планують зайняти одну і ту ж вершину одночасно.

– конфлікт ребер – конфлікт меж між π_i та π_j виникає, якщо згідно з цими планами агенти планують пройти одне і те саме ребро одночасно з кроком у тому ж напрямку.

– конфлікт переслідування – конфлікт між π_i та π_j виникає, якщо один агент планується зайняти вершину, яка була зайнята іншим агентом на попередньому кроці часу.

– конфлікт циклу – конфлікт між набором одноагентних планів π_i, π_{i+1}, ..., π_j виникає, якщо на тому ж кроці часу кожен агент рухається до вершини, яка раніше була зайнята іншим агентом, утворюючи шаблон «обертального циклу».

– конфлікт обміну – конфлікт між π_i та π_j виникає, якщо агенти планують міняти місцями за один часовий крок.

Рисунок 1.1 ілюструє різні типи конфліктів. Зверніть увагу, що наведений набір визначень конфліктів, безумовно, не є повним набором усіх можливих конфліктів.

Беручи до уваги формальне визначення цих конфліктів, очевидно, що між ними існують відносини домінування: заборона конфліктів вершин означає, що також заборонені конфлікти країв, заборона конфліктів ребер означає що конфлікти циклів і конфлікти обміну також заборонені, заборона конфліктів циклів означає, що конфлікти обміну також заборонені.

Навпаки, дозвіл конфліктів країв означає, що дозволені також конфлікти вершин; дозвіл конфліктів обміну, означає, що конфлікти циклів також допускаються, і дозвіл конфліктів циклів, передбачає також конфлікти переслідування [18].

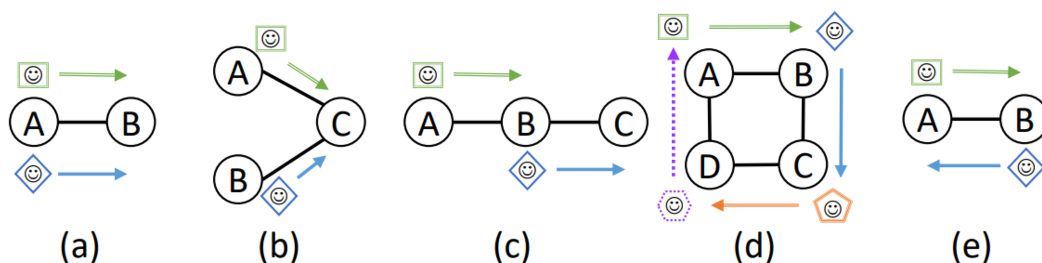

Рисунок 1.3 – Ілюстрація поширених типів конфліктів.

1.3.2 Поведінка агента у цілі в класичному MAPF

У вирішенні класичної проблеми MAPF агенти можуть досягти своїх цілей на різних етапах часу. Отже, при визначенні класичної проблеми MAPF потрібно визначити, як поводиться агент на етапах часу після досягнення цілі та до того, як останній агент досяг цілі.

Існує два загальних припущення щодо того, як агенти поводяться у своїх цілях:

– залишаються в цілі – за цим припущенням агент чекає у своїй меті, поки всі агенти не досягнуть своїх цілей. Цей агент очікування спричинить конфлікт вершин з будь-яким планом, який проходить через його ціль після того, як він її досяг.

– зникають в цілі – за цим припущенням, коли агент досягає цілі, він негайно зникає. Це означає, що план цього агента не матиме жодних конфліктів після того періоду часу, за який відповідний агент досяг своєї мети.

1.3.3 Цільові функції в класичному MAPF

Можна з упевненістю сказати, що в більшості реальних додатків MAPF деякі рішення MAPF кращі за інші. Щоб зрозуміти це, робота в класичному MAPF розглядає цільову функцію, яка використовується для оцінки рішень MAPF. Дві найпоширеніші функції, що використовуються для оцінки рішення в класичному MAPF – це сума часу та сума витрат:

– сума часу. Кількість часових кроків, необхідних всім агентам для досягнення цілі.

– сума витрат. Сума часових кроків, необхідних кожному агенту для досягнення своєї мети.

Якщо поведінка агента-цілі залишається на меті, а цільова функція – це сума витрат, тоді потрібно вказати, як перебування на цілі впливає на суму

витрат. Наприклад, можна визначити, що якщо агент чекає своєї мети, це не збільшує суми витрат.

Типовим припущенням у більшості попередніх робіт є те, що агент, що перебуває у своїй цілі, вважається дією очікування, якщо він не планує знову відходити від своєї цілі.

Наприклад, припустимо, що агент i досягає цілі на кроці часу t, залишає ціль на кроці часу t', повертається до цілі на кроці часу t'', а потім залишається на цілі, доки всі агенти не досягнуть своєї мети. Тоді цей одноагентний план внесе t'' у суму витрат на відповідне рішення.

Звичайно це не єдино можливі цільові функції для класичного MAPF. Можна визначити інші цільові функції, такі як загальна кількість непередбачуваних дій, необхідних для досягнення цілі (деякі називають це сумою палива), і загальний час, витрачений агентом, що не знаходиться в цілі. Однак, наскільки нам відомо, вищезазначені цільові функції є єдиними, що використовуються в попередній роботі над класичним MAPF.

Сума часу широко використовується алгоритмами MAPF на основі компіляції, тоді як сума витрат використовується більшістю алгоритмів MAPF на основі пошуку. Але також була проведена робота над обома цільовими функціями за допомогою обох типів алгоритмів MAPF [19].

Також була проведена робота щодо максимізації кількості агентів, що досягають своїх цілей протягом заданого періоду часу (тобто терміну) [20].

1.3.4 MAPF на зважених графах

Припущення, що кожна дія – переміщення або очікування – займає рівно один часовий крок, неявно передбачає дещо спрощену модель руху для агентів. Більш складні моделі руху були вивчені в літературі MAPF, в яких різні дії можуть мати різну тривалість.

Це означає, що базовий граф, що представляє можливі місця розташування агентів, які можуть зайняти (позначено раніше як G), тепер є зваженим графом, де вага кожного ребра відображає тривалість, необхідну агенту для проходження цього краю.

Типи зважених графів, які використовуються у MAPF, включають:

– MAPF у 2^k-сусідніх сітках. Такі карти є обмеженою формою зважених графіків, в яких кожна вершина представляє клітинку у двовимірній сітці. Дії переміщення агента в комірці – це всі його 2^k сусідні комірки, де k – параметр. Витрати засновані на евклідовій відстані, тому, коли $k > 2$, це вводить дії з різними витратами. Наприклад, у 8-сусідній сітці діагональний хід коштує $\sqrt{2}$, тоді як переїзд в одному з основних напрямків коштує 1. Рисунок 1.4 показує можливі дії переміщення в 2^k – сусідні сітки для k = 2, 3, 4 і 5;

– MAPF в евклідовому просторі. MAPF в евклідовому просторі – це узагальнення MAPF, в якому кожен вузол в G представляє евклідову точку (x, y), а ребра – дозволені дії переміщення. Такі налаштування виникають, наприклад, коли базовий графік є дорожньою картою, сформованою для безперервного евклідового середовища [21];

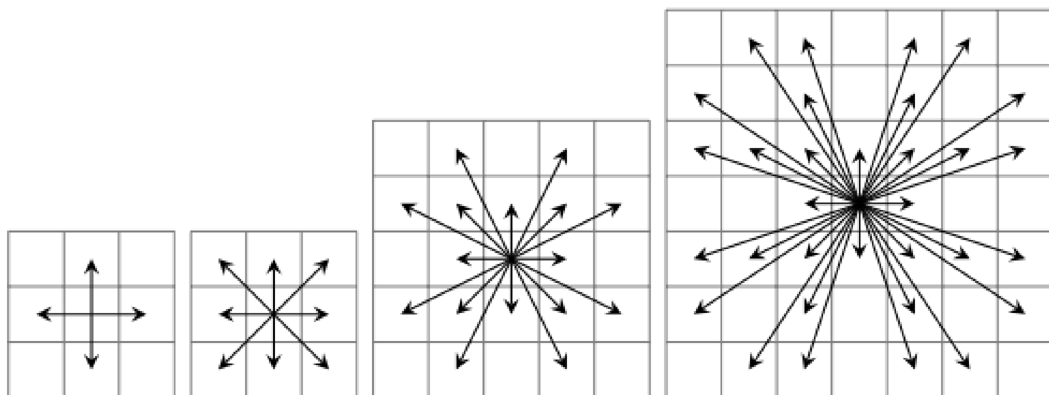

Рисунок 1.4 – 2^k моделі руху сусідства для k = 2,3,4 та 5.

1.3.5 Завдання та агенти

У класичному MAPF кожен агент має одне завдання – дістати його до своєї мети. У літературі MAPF було зроблено кілька розширень, в яких агентам може бути призначено більше однієї цілі:

а) анонімний MAPF. У цьому варіанті MAPF метою є переміщення агентів до набору цільових вершин, але не має значення, який агент досягає якої цілі. Інший спосіб розглянути цей варіант MAPF – це проблема MAPF, при якій кожен агент може бути призначений будь-якій цілі, але це має бути індивідуальне відображення між агентами та цілями.

б) кольоровий MAPF. Цей варіант MAPF є узагальненням анонімного MAPF, в якому агенти згруповані в команди, і кожна команда має набір цілей. Мета полягає в тому, щоб перемістити агентів у кожній команді до їх цілей [20]. Інший спосіб розглянути цей варіант MAPF – це проблема MAPF, при якій кожен агент може бути призначений цілям лише з набору цілей, призначених для його команди. Можна ще більше узагальнити кольорові MAPF, призначаючи ціль та агента декільком командам.

в) онлайн MAPF. В онлайн-MAPF послідовність задач MAPF вирішується на одному графіку. Цей параметр також називали «MAPF протягом усього життя» [20]. Інтернет-проблеми MAPF можна класифікувати наступним чином:

- складська модель. Це параметр, коли фіксований набір агентів вирішує проблему MAPF, але після того, як агент знаходить ціль, може бути поставлено завдання перейти до іншої цілі [20]. Цей параметр натхненний MAPF для автономних складів;
- модель перетину. Це установка, де можуть з’являтися нові агенти, і кожен агент має одне завдання – досягти своєї мети. Цей параметр натхненний автономними транспортними засобами, які в’їжджають і виїжджають з перехрестя.

Звичайно, можливі також гібридні моделі, в яких агент може отримати нове завдання, коли досягне своєї мети, а нові агенти можуть з’явитися з часом.

1.4 Опис існуючих алгоритмів вирішення задачі MAPF

Оптимальні розв'язувачі MAPF зазвичай шукають у глобальному просторі пошуку, який поєднує окремі стани всіх k-агентів. Цей простір станів позначається як простір станів k-агента. Стани в просторі станів k-агента – це різні способи розміщення k агентів у |V| вершини, по одному агенту на вершину. У стартах старту та цілі агент a_i знаходиться відповідно у вершинах $start_i$ та $goal_i$. Оператори між станами – це всі неконфліктні дії (включаючи очікування), які мають усі агенти. З огляду на цей загальний простір станів, будь-який алгоритм на основі A* може бути використаний для оптимального вирішення проблеми MAPF.

Введемо термін b_{base} для позначення фактора розгалуження окремого агента, тобто кількості місць, куди може переміститися один агент за один часовий крок. У цій роботі основна увага приділяється 4 підключеним сіткам, де $b_{base} = 5$, оскільки кожен агент може рухатися до чотирьох основних напрямків або чекати у своєму поточному місці.

При розширенні стану в просторі станів k-агента можуть розглядатися всі комбінації b-потенціалів, але законними сусідами є лише ті, які не мають конфліктів (з іншими агентами чи перешкодами). Кількість законних сусідів позначається як b_{legal}.

Для найліпшого аналізу можна розглядати b_{legal} в тому ж порядку, що і $b_{potential}$, тобто експоненціальний за кількістю агентів (k). З іншого боку, на щільних графіках (з великою кількістю агентів і з невеликою кількістю порожніх станів) b_{legal} може бути набагато меншим, ніж $b_{potential}$.

Загалом, ідентифікація нелегальних сусідів від можливих $b_{potential}$ сусідів є проблемою задоволення обмежень (CSP), де змінні є агентами, значення – це дії,

які вони вживають, і обмеження – щоб уникнути конфліктів. Надалі ми просто позначаємо b_{legal} через b.

Щоб ефективніше розв’язувати MAPF за допомогою A*, потрібна нетривіальна допустима евристика.

Проста допустима евристика полягає в підсумовуванні індивідуальних евристик окремих агентів, таких як відстань Манхеттена для 4-з'єднаних сіток або евклідова відстань для евклідових графіків. HCA* покращує це шляхом обчислення оптимальної відстані до мети для кожного агента, ігноруючи інші агенти [22].

Оскільки це завдання суто простіше, ніж пошук за допомогою додаткових агентів, воно може бути використано як допустима евристика. HCA* виконував це обчислення поступово для кожного агента, тоді як Стенлі виконував обчислення апріорі вичерпно до вирішення проблеми MAPF [23].

A* завжди починається з розширення стану та вставки його наступників у відкритий список (позначається OPEN).

Усі розширені штати зберігаються у закритому списку (позначається CLOSED). Через це A* для MAPF страждає від двох недоліків. По-перше, розмір простору станів експоненціальний за кількістю агентів (k), що означає, що CLOSED не може підтримуватися в пам'яті для великих проблем.

По-друге, коефіцієнт розгалуження даного стану може бути експоненціальним у k. Розглянемо стан із 20 агентами на 4-з'єднаній сітці. Кожен агент може мати до 5 можливих ходів (4 основні вказівки та очікування). Повна генерація всіх $5^{20} = 9{,}53 \times 10^{14}$ сусідів навіть стартового стану може бути обчислювально нездійсненною. Для подолання цих недоліків було запропоновано наступні вдосконалення.

Оскільки простір станів MAPF є експоненціальним за кількістю агентів, експоненціальний пришвидшення можна отримати, зменшивши кількість агентів у задачі. З цією метою Стенді запровадив систему виявлення незалежності Independence Detection framework (ID) [23]

Дві групи агентів незалежні, якщо для кожної групи існує оптимальне рішення, яке не має конфліктів. ID намагається виявити незалежні групи агентів. По-перше, кожен агент потрапляє у свою групу.

Кожна група вирішується окремо за допомогою A*. Рішення, повернене A*, є оптимальним щодо даної групи агентів. Потім шляхи всіх груп перевіряються на валідність щодо один одного. Якщо виявлено конфлікт, конфліктні групи об’єднуються в одну групу та оптимально вирішуються за допомогою A*.

Цей процес перепланування та об’єднання груп повторюється доти, доки не буде конфліктів між планами всіх груп. Отримані групи є незалежними один від одного. Зверніть увагу, що ID не є ідеальним, в тому сенсі, що більш незалежні підгрупи можуть лежати непоміченими в групах, повернутих за допомогою ID.

Тепер звернемося до опису алгоритму пошуку на основі конфлікту (CBS). Нагадаємо, що простір станів, охоплений A* у MAPF, є експоненціальним у k (кількість агентів). Навпаки, в задачі пошуку одного шляху агента k = 1, а простір станів лише лінійний за розміром графіка.

CBS вирішує MAPF, розкладаючи його на велику кількість обмежених одноагентних задач пошуку шляхів. Кожна з цих проблем може бути вирішена в часі, пропорційно розміру карти та довжині рішення, але таких одноагентних проблем може бути експоненціальна кількість.

1.5 Постановка задачі

Метою даної роботи є розробка шаблону гри у жанрі покрокової стратегії з реалізованим мультиагентним алгоритмом пошуку шляхів для ігрових персонажів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– вивчити предметну область;

– проаналізувати існуючі методи вирішення поставленого завдання;

– вибрати найбільш оптимальний спосіб реалізації програми;

– реалізувати вибраний спосіб для досягнення мети роботи.

Гра складається з наступних сутностей:

– ігрова мапа, що складається з плиток, які можуть буди прохідними чи не прохідними;

– ігрові персонажі, розміщені на мапі, кожен з яких може пересуватися по клітинам мапи по команді гравця.

По команді гравця на переміщення одного з персонажів виконується пошук оптимального путі до нової плитки, враховуючи переміщення інших персонажів на мапі.

Алгоритм пошуку шляху повинен враховувати наступні обмеження:

– ігрові персонажі можуть перейти на одну сусідню плитку за один крок;

– ігровий персонаж може перейти на сусідню плитку якщо вона є прохідною;

– тільки один ігровий персонаж може бути на плитці у кінці ходу.

Для вирішення задачі будуть використовуватися алгоритми CBS та D* Lite, модифіковані таким чином, щоб вони відповідали наведеним обмеженням. Також, для спрощення оцінки результату реалізована програма має візуально відображати результати кожного виконання алгоритму.

2 ДОСЛІДЖЕННЯ МЕТОДІВ РОЗВ’ЯЗАННЯ ЗАДАЧІ

Наступні визначення використовуються в решті роботи:

– термін шлях використовується лише в контексті одного агента, а термін рішення – для позначення набору k шляхів для даного набору k агентів.

– обмеження – це кортеж (a_i, v, t), де агенту a_i заборонено займати вершину v на момент часу t. Під час роботи алгоритму агенти будуть пов'язані з обмеженнями. Послідовний шлях для агента a_i – це шлях, який задовольняє всі його обмеження. Аналогічним чином, послідовне рішення – це рішення, яке складається з шляхів, таким чином, що шлях для будь-якого агента a_i узгоджується з обмеженнями a_i.

– конфлікт – це кортеж (a_i, a_j, v, t), де агент a_i та агент a_j займають вершину v в момент часу t. Рішення (з k шляхів) є дійсним, якщо всі його шляхи не мають конфліктів. Послідовне рішення може бути недійсним, якщо, незважаючи на те, що окремі шляхи узгоджуються з обмеженнями, пов'язаними з їх агентами, ці шляхи все ще мають конфлікти.

Ключова ідея CBS полягає у створенні набору обмежень та пошуку шляхів, які відповідають цим обмеженням. Якщо ці шляхи мають конфлікти і, таким чином, недійсні, конфлікти вирішуються додаванням нових обмежень. CBS працює на двох рівнях. На високому рівні виявляються конфлікти та додаються обмеження. Низький рівень знаходить шляхи для окремих агентів, які відповідають новим обмеженням. Далі кожна частина цього процесу описана більш докладно [23].

2.1 Дослідження алгоритму високого рівня

На високому рівні CBS здійснює пошук у дереві, яке називається деревом обмежень (CT). CT – це бінарне дерево. Кожен вузол N в CT складається з:

– набір обмежень (N.constraints). Кожне з цих обмежень належить одному агенту. Корінь CT містить порожній набір обмежень. Дочірній елемент вузла в CT успадковує обмеження батьківського і додає одне нове обмеження для одного агента.

– рішення (N.solution). Набір з k шляхів, по одному шляху для кожного агента. Шлях для агента a_i повинен узгоджуватися з обмеженнями a_i. Такі шляхи знаходять пошук низького рівня.

– загальна вартість (N.cost) поточного рішення (підсумована за витратами на одноагентний шлях). Ця вартість позначається як f-значення вузла N.

Вузол N у CT є цільовим вузлом, коли N.solution є дійсним, тобто набір шляхів для всіх агентів не має конфліктів. Високий рівень виконує найкращий перший пошук на КТ, де вузли впорядковуються за вартістю. У нашому впровадженні зв'язки розриваються на користь вузлів КТ, пов'язане рішення яких містить менше конфліктів. Подальші зв'язки були розірвані в порядку FIFO.

Враховуючи список обмежень для вузла N CT, викликається пошук на низькому рівні. Низькорівневий пошук (докладно описаний нижче) повертає один найкоротший шлях для кожного агента, a_i, що узгоджується з усіма обмеженнями, пов’язаними з a_i у вузлі N. Після того, як для кожного агента знайдений послідовний шлях (щодо власні обмеження) ці шляхи потім перевіряються щодо інших агентів. Перевірка виконується шляхом ітерації всіх часових кроків та узгодження розташувань, зарезервованих усіма агентами. Якщо два агенти не планують перебувати в одному і тому ж місці одночасно, цей вузол N CT оголошується цільовим вузлом і повертається поточне рішення (N.solution), що містить цей набір шляхів. Однак, якщо під час виконання перевірки виявляється конфлікт для двох або більше агентів a_i і a_j, перевірка зупиняється, а вузол оголошується нецільовим вузлом.

З огляду на нецільовий вузол CT N, рішення якого N.solution включає конфлікт C_n, ми знаємо, що в будь-якому допустимому рішенні щонайбільше один із конфліктуючих агентів (a_i та a_j) може займати вершину v в момент часу

t. Отже, принаймні одне з обмежень (a_i, v, t) або (a_j, v, t) повинно бути додано до набору обмежень у N.constraints. Для гарантії оптимальності розглядаються обидві можливості, і вузол N ділиться на двох дочірніх організацій. Обидва діти успадковують набір обмежень від N. Ліва дочірня вирішує конфлікт, додаючи обмеження (a_i, v, t), а права дочірнє додає обмеження (a_j, v, t).

Зверніть увагу, що для даного вузла CT N не потрібно зберігати всі його сукупні обмеження. Натомість він може зберегти лише своє останнє обмеження та витягти інші обмеження, пройшовши шлях від N до кореня через своїх предків. Подібним чином, за винятком кореневого вузла, пошук на низькому рівні повинен виконуватися лише для агента a_i, який пов'язаний із нещодавно доданим обмеженням. Шляхи інших агентів залишаються незмінними, оскільки для них не додаються нові обмеження. Псевдокод алгоритму CBS поданий у лістингу 2.1.

Лістинг 2.1 – Псевдокод алгоритму CBS

```
Input: MAPF instance
Root.constraints = Ø
Root.solution = find individual paths by the low level()
Root.cost = SIC(Root.solution)
insert Root to OPEN
while OPEN not empty do
  P ← best node from OPEN // lowest solution cost
  Validate the paths in P until a conflict occurs.
  if P has no conflict then
    return P.solution // P is goal
  C ← first conflict (ai,a j, v,t) in P
  if shouldMerge(ai,a j) // Optional, MA-CBS only then
    a{i,j} = merge(ai,a j, v,t)
    Update P.constraints(external constraints).
    Update P.solution by invoking low level(a{i,j})
    Update P.cost
```

Продовження лістингу 2.1

```
if P.cost < ∞ // A solution was found then
      Insert P to OPEN
    continue // go back to the while statement
  foreach agent ai in C do
    A ← new node
    A.constraints ← P.constraints + (ai, v,t)
    A.solution ← P.solution
    Update A.solution by invoking low level(ai)
    A.cost = SIC(A.solution)
    if A.cost < ∞ // A solution was found then
           Insert A to OPEN
```

2.1.1 Приклад використання алгоритму CBS

Опишемо CBS, використовуючи приклад з рисунку 2.1, де мишам потрібно дістатися до відповідних шматків сиру. Відповідна CT показана на рисунку 2.2. Корінь містить порожній набір обмежень. У рядку 2 низький рівень повертає оптимальне рішення для кожного агента, <S_1, A_1, D, G_1> для a_1 та <S_2, B_1, D, G_2> для a_2. Таким чином, загальна вартість цього вузла дорівнює 6. Потім корінь вставляється у OPEN і буде розширений далі.

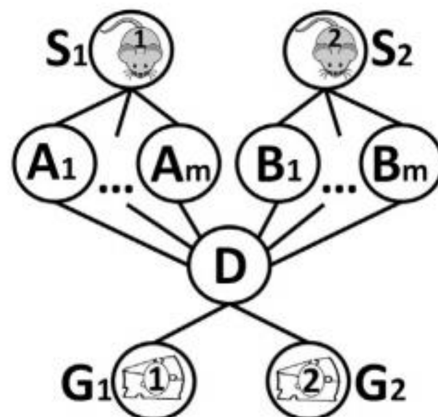

Рисунок 2.1 – Приклад екземпляра MAPF з 2 агентами

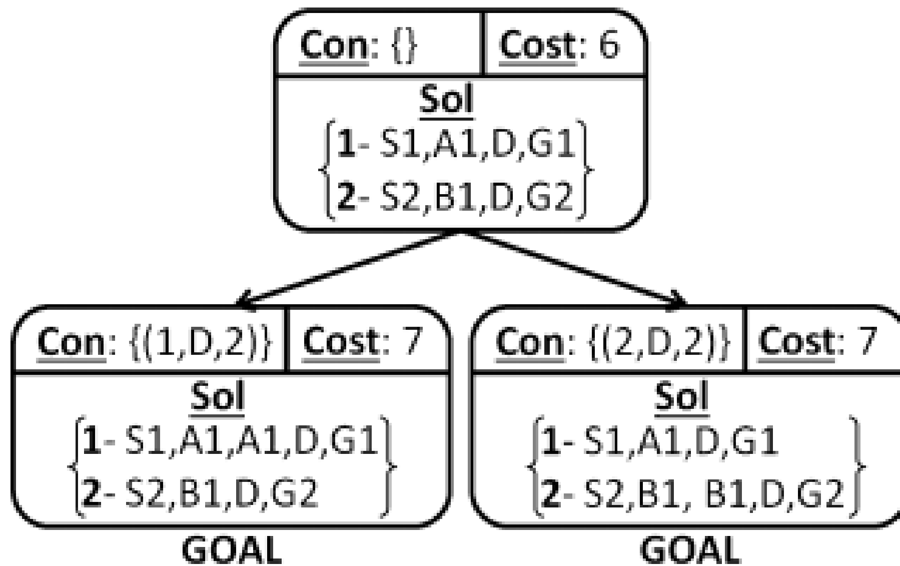

Рисунок 2.2 – Приклад дерева обмежень (CT)

Під час перевірки рішення двох агентів, заданого двома окремими шляхами, виявляється конфлікт, коли обидва агенти приходять до вершини D на етапі часу 2. Це створює конфлікт (a_1, a_2, D, 2). В результаті корінь оголошується не ціллю, а для вирішення конфлікту генерується двоє дочірніх елементів.

Лівий дочірній додає обмеження (a_1, D, 2), тоді як правий дочірній додає обмеження (a_2, D, 2). Тепер викликається пошук на низькому рівні для лівої дочірньої дитини, який також задовольняє новому обмеженню. Для цього a1 повинен зачекати один крок або на A_1, або на S_1, і шлях $<S_1, A_1, A_1, D, G_{1>}$ повертається для a1.

Шлях для $<a_2, S_2, B_1, D, G_2>$ залишається незмінним у лівої дочірньої дитини. Обидві дитини вставляються в OPEN. У наступній ітерації циклу while лівий дочірній елемент вибирається для розширення та перевіряються основні шляхи. Оскільки конфліктів не існує, ліва дочірня особа оголошується вузлом цілі і її рішення повертається як оптимальне рішення.

2.2 Дослідження алгоритму низького рівня

Розглянемо цілеспрямоване завдання навігації в невідомому рельєфі, де агент завжди спостерігає, які з його сусідніх кліток прохідні і потім рухається до одного з них.

Агент починає з початкової клітки і повинен перейти до цільової клітки. Він завжди обчислює найкоротший шлях від своєї поточної клітки до цільової клітки за умови, що клітки з невідомим статусом блокування є прохідними. Потім він рухається по цьому шляху, поки не досягне цільової клітки, і в цьому випадку він успішно зупиняється, або спостерігає непрохідну клітинку, і в цьому випадку він перераховує найкоротший шлях від своєї поточної клітки до цільової клітки.

Рисунок 2.1 показує відстані цілей всіх прохідних клітин і найкоротших шляхів від його поточної клітки до цільової клітки як до, так і після того, як агент перемістився по шляху і виявив першу заблоковану клітку, про яку він не знав. Клітки, чиї відстані до цілі змінилися, затінені сірим. Відстані до цілі важливі, тому що можна легко визначити найкоротший шлях від поточної клітки агента до клітки, зменшуючи відстані до цілі після того, як будуть обчислені відстані до цілі.

Зверніть увагу на те, що кількість кліток із зміненими відстанями до цілі невелика, і більшість змінених відстаней до цілі не мають значення для перерахунку найкоротшого шляху від поточної клітки до цільової клітки. Таким чином, можна ефективно перерахувати найкоротший шлях від своєї поточної клітки до цільової клітки, перераховуючи тільки ті відстані, які змінилися (або не були розраховані раніше), і є релевантними для перерахунку найкоротшого шляху [14].

Приклад використання алгоритму D* Lite показаний на рисунку 2.3.

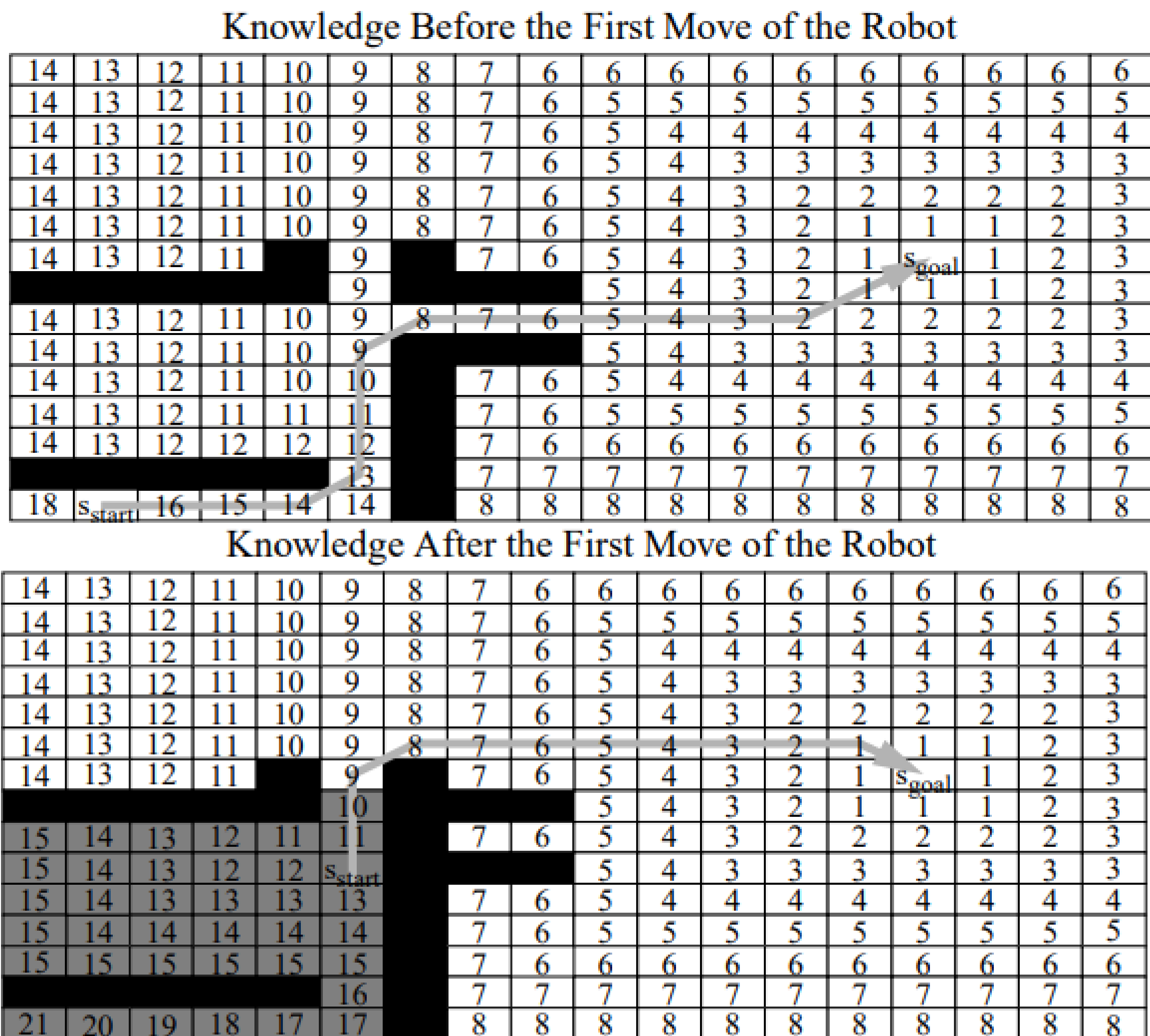

Рисунок 2.3 – Приклад використання D* Lite

2.2.1 Математичний опис алгоритму LPA*

Lifelong Planning A* (LPA*) LPA* є інкрементною версією A*. Вона застосовна до задач пошуку на кінцевих та відомих графах, вартість ребер яких збільшується або зменшується з плином часу (що також можна використовувати для моделювання ребер або вершин, які додаються або видаляються). позначає кінцевий набір вершин графа.

Succ(S) позначає множину наступників вершини s ∈ S. Аналогічно, Pred(s) позначає множину попередників вершини s ∈ S. LPA* завжди визначає

найкоротший шлях від заданої початкової вершини s_{start} ∈ S до заданої цільової вершини s_{goal} ∈ S, знаючи як топологію графа, так і поточну вартість ребер. Ми використовуємо g*(s) для позначення стартової відстані вершини s ∈ S, тобто довжини найкоротшого шляху від s_{start} до s. Подібно до A*, LPA* використовує евристики, які наближають цілі відстані кліток s, евристики повинні бути невід'ємними та послідовними, тобто підкорятися нерівності трикутника для всіх вершин при $s \neq s_{goal}$ [12]. Псевдокод алгоритму LPA* поданий у лістингу 2.2.

Лістинг 2.2 – Псевдокод алгоритму LPA*

```
procedure CalculateKey(s)
return [min(g(s), rhs(s)) + h(s,sgoal); min(g(s), rhs(s))];

procedure Initialize()
U = Ø;
for all s ∈ Srhs(s) = g(s) = ∞;
  rhs(sstart) = 0;
  U.Insert(sstart, CalculateKey(sstart));

procedure UpdateVertex(u)
if (u ≠ sstart) rhs(u) = mins' ∈ Pred(u)(g(s') + c(s', u));
if (u ∈ U) U.Remove(u);
if(g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
while (U.TopKey() < CalculateKey(sgoal) OR rhs(Sgoal) ≠ g(sgoal))
  u = U.Pop();
  if (g(u) > rhs(u))
    (u) = rhs(u);
    for all s ∈ Succ(u) UpdateVertex(s);
  else
    g(u) = 0;
    for all s ∈ Succ(u) ∪ {u} UpdateVertex(s);
```

Продовження лістингу 2.2

```
procedure Main()
Initialize();
forever
  ComputeShortestPath();
  Wait for changes in edge costs;
  for all directed edges (u, v) with changed edge costs
    Update the edge cost c(u, v);
    Update Vertex(v);
```

Псевдокод використовує наступні функції для керування приоретизованою чергою: U.Top() повертає вершину з найменшим пріоритетом усіх вершин в приоретизованій черзі U. U.TopKey() повертає найменший пріоритет всіх вершин в приоретизованій черзі U. (якщо U пуста, то U.TopKey() повертає [∞; ∞]) U.Pop() видаляє вершину з найменшим пріоритетом в черзі U і повертає вершину. U.Insert(s, k) вставляє вершини s в чергу U з пріоритетом k. U.Update(s, k) змінює пріоритет вершин s в черзі пріоритету U до k. (Він нічого не робить, якщо поточний пріоритет вершин s вже дорівнює k). Нарешті, U.Remove(s) видаляє вершину s з черги U.

LPA* підтримує оцінку g(s) початкової відстані g* (s) кожної вершини s. Ці значення безпосередньо відповідають g-значенням пошуку A*. LPA* переносить їх з пошуку до пошуку. LPA* також підтримує другу оцінку стартових відстаней.

Вершина називається локально консистентною, якщо її g-значення дорівнює його rhs-значенню, інакше вона називається локально неконсистентною. Якщо всі вершини локально узгоджені, то g-значення всіх вершин дорівнюють їх відповідним стартовим відстаням. У цьому випадку можна простежити найкоротший шлях від s_{start} до будь-якої вершини u, переходячи з поточної вершини s, починаючи з u, до будь-якого попередника s', який мінімізує g(s') + c(s', s) (нічиї можна вирішувати довільно) до тих пір, поки

не буде досягнутий s_{start}. Проте LPA* не робить всі вершини локально консистентними після зміни вартостей деяких ребер графу. Замість цього він використовує евристику для фокусування пошуку та оновлення лише значень g, які є важливими для обчислення найкоротшого шляху.

З цією метою LPA* підтримує приоретизовану чергу. Черга завжди містить тільки локально неконсистентні вершини (інваріант 2). Це вершини, для яких g-значення LPA* потенційно потребують оновлення, щоб зробити їх локально консистентними.

Пріоритет вершини в черзі завжди збігається з його ключем (інваріант 3), який є вектором з двома компонентами:

$$k(s) = [k1(s); k2(s)], \tag{2.1}$$

де $k_1(s) = \min(g(s),$

$rhs(s)) + h(s, s_{goal})$,

$k_2(s) = \min(g(s), rhs(s))$.

Перша складова ключів $k_1(s)$ безпосередньо відповідає значенням f(s), що використовується A*, оскільки і g-значення, і rhs-значення LPA* відповідають g-значенням A*, а h-значення LPA* відповідають h-значенням A*. Друга складова ключів $k_2(s)$ відповідає g-значенням A*. Ключі порівнюються відповідно до лексикографічного впорядкування.

LPA* завжди розширює вершину в приоретизованій черзі з найменшим ключем. Це подібно до A*, який завжди розширює вершину в черзі пріоритету з найменшим значенням f, якщо він розриває зв'язки з найменшим значенням g. Результат поведінки LPA* і A* також схожий. Ключі вершин, розширених за допомогою LPA*, з часом не зменшуються так само, як f-значення вершин, розширених A* (оскільки евристики є послідовними).

Основна функція Main() у LPA* спочатку викликає Initialize() для ініціалізації задачі пошуку. Initialize() встановлює g-значення всіх вершин до

нескінченності і встановлює їх значення rhs. Таким чином, спочатку s_{start} є єдиною локально неконсистентною вершиною і вставляється в інакше порожню приоретизовану чергу. Ця ініціалізація гарантує, що перший виклик ComputeShortestPath() виконує точно пошук A*, тобто розширює точно ті ж вершини, що і A*, в точно такому ж порядку.

Зауважимо, що в реальній реалізації Initialize() потрібно ініціалізувати вершину, тільки коли вона зустрічається з нею під час пошуку, і тому не потрібно ініціалізувати всі вершини спереду. Це важливо, оскільки кількість вершин може бути великим і лише деякі з них можуть бути досягнуті під час пошуку.

Потім LPA* чекає змін у вартості ребер. Для збереження інваріантів 1-3, якщо деякі вартості ребер змінені, він викликає UpdateVertex() для оновлення rhs-значень і ключів вершин, які потенційно впливають на зміну вартостей ребер, а також на їхнє членство в черзі пріоритетів, якщо вони стають локально послідовними або непослідовний, і, нарешті, перераховує найкоротший шлях, викликаючи ComputeShortestPath(), що неодноразово розширює локально непослідовні вершини в порядку їх пріоритетів.

Локально неконсистентні вершини s називаються локально надконсистентними, якщо $g(s) > rhs(s)$. Коли ComputeShortestPath() розширює локально надконсистентну вершину, то вона встановлює g-значення вершини до її значення rhs, що робить вершину локально консистентною.

Локально неконсистентні вершини s називаються локально недоконсистентними якщо $g(s) < rhs(s)$. Коли ComputeShortestPath() розширює локально недоконсистентну вершину, то вона просто встановлює g-значення вершини до нескінченності. Це робить вершину локально послідовною або надконсистентною.

Якщо розширена вершина була локально надконсистентною, то зміна її g-значення може вплинути на локальну консистентність її наступників. Аналогічно, якщо розширена вершина була локально недоконсистентною, це може вплинути на неї та її наступників.

Для підтримки інваріантів 1-3, ComputeShortestPath() оновлює rhs-значення цих вершин, перевіряє їх локальну консистентність, і додає їх до або видаляє їх з приоретизованої черги відповідно. ComputeShortestPath() розширює вершини до тих пір, поки s_{goal} не буде локально консистентною, а ключ вершини для наступного розширення не менше, ніж ключ s_{goal}.

Це схоже на A*, що розширює вершини, поки він не розширить s_{goal}, в який момент часу g-значення s_{goal} дорівнює його початковій відстані, а f-значення вершини, що розширюється далі, не менше, ніж f-значення s_{goal}. Якщо $g(s_{goal}) = \infty$ після пошуку, то не існує кінцевого шляху від s_{start} до s_{goal}. В іншому випадку можна простежити найкоротший шлях від s_{start} до s_{goal}, переходячи з поточної вершини s, починаючи з s_{goal}, до будь-якого попередника s', який мінімізує g(s) + c(s', s) (нічиї можуть бути вирішені довільно) до початку s.

2.2.2 Математичний опис алгоритму D* Lite

D* Lite майже повністю копіює LPA*, але, на відміну від LPA* він враховує переміщення агента з плином часу.

Спочатку необхідно змінити напрямок пошуку LPA* Версія LPA*, представлена на раніше, шукає від початкової вершини до цільової вершини, і, отже, її g-значення є оцінками відстаней від старту. D* Lite виконує пошук від цільової вершини до стартової вершини і, таким чином, його g-значення є оцінкою відстаней до цілі.

Він відрізняється з LPA* шляхом обміну початкової і цільової вершини і реверсування всіх ребер псевдокоду. Таким чином, D* Lite працює на початковому графіку і на графіку немає обмежень, за винятком того, що він повинен вміти визначати наступників і попередників вершин, як LPA*.

Після повернення з ComputeShortestPath() можна слідувати за найкоротшим шляхом від s_{start} до s_{goal}, завжди переходячи з поточної вершини s,

починаючи з s_{start}, до будь-якого наступника s', який мінімізує c(s, s') + g(s ') аж до досягнення s_{goal} (нічиї можуть бути вирішені довільно).

Щоб вирішити проблеми навігації в невідомому рельєфі, Main() тепер повинна перемістити агента по шляху, визначеному CalculatePath(). Main() може перераховувати пріоритети вершин в приоретизованій черзі кожен раз, коли агент помічає зміну вартості ребер після переміщення. Якщо пріоритети не будуть перераховані, вони не задовольняють інваріант 3, оскільки вони базуються на евристиках, обчислених по відношенню до старої вершини агента. Проте повторне пере-упорядкування черги може бути дорогою операцією, оскільки черга часто містить велику кількість вершин.

Внаслідок цього, D* Lite використовує метод, отриманий з D*, щоб уникнути необхідності пере упорядкувати чергу пріоритетів, а саме пріоритети, які є нижчими межами пріоритетів, які LPA* використовує для відповідних вершин.

Ця вимога не є обмежуючою, оскільки обидві властивості гарантовано виконуються, якщо евристики отримані шляхом розслаблення задачі пошуку, яка майже завжди буде доречною і справедливою для евристики, використовуваної в даній роботі.

Після переміщення агента з вершин s до деяких вершин s', де він виявляє зміни у вартості ребер, перший елемент пріоритетів (ключів) черги може зменшитися не більше ніж на h(s, s'). (Друга складова не залежить від евристики і, таким чином, залишається незмінною.) Таким чином, для підтримки нижньої межі, потрібно відняти h(s, s') з першого елемента пріоритетів усіх вершин в черзі пріоритету.

Однак, оскільки h (s, s') однакова для всіх вершин в черзі пріоритету, порядок вершин в черзі пріоритету не змінюється, якщо віднімання не виконується. Потім, коли обчислюються нові пріоритети, їх перші компоненти є на h(s, s') меншими щодо пріоритетів у черзі. Таким чином, h(s, s') потрібно додавати до їх перших компонентів кожен раз, коли змінюються деякі вартості

ребер. Якщо агент знову рухається, а потім знову виявляє зміни у вартості, то константи потрібно додати. Ми робимо це в змінній k_m. Таким чином, коли обчислюються нові пріоритети, змінна k_m повинна бути додана до їх перших компонентів.

Таким чином, порядок вершин в черзі пріоритету не змінюється після переміщення агента, а приоретизовану чергу не потрібно пере упорядковувати. Пріоритети, з іншого боку, завжди є нижчими межами відповідних пріоритетів LPA* після того, як перша складова пріоритетів LPA* була збільшена на поточну вартість k_m. Ми використовуємо цю властивість, змінюючи ComputeShortestPath() наступним чином.

Після того, як ComputeShortestPath() видалила вершину u з найменшим пріоритетом k_{old} = U.TopKey() з черги пріоритету, тепер вона використовує CalculateKey() для обчислення пріоритету, який він повинен був мати.

Якщо k_{old} < CalculateKey(u), то він повторно вставляє видалену вершину з пріоритетом, обчисленим CalculateKey() в чергу пріоритету. Таким чином, залишається вірним, що пріоритети всіх вершин в черзі є нижчими межами відповідних пріоритетів LPA* після того, як перші компоненти пріоритетів LPA* були збільшені на поточне значення k_m. Якщо $k_{old} \geq$ CalculateKey(u), то він вважає, що k_{old} = CalculateKey(u), оскільки k_{old} був нижньою межею значення, що повертається CalculateKey(). У цьому випадку ComputeShortestPath() розширює вершину u так само, як LPA*. Псевдокод алгоритму D* Lite поданий у лістингу 2.3.

Лістинг 2.3 – Псевдокод алгоритму D* Lite

```
procedure CalculateKey(s)
  return (min(g(s), rhs(s)) + h(sstart,s) + km; min(g(s),
rhs(s))];

procedure Initialize()
  U = Ø;
```

Продовження лістингу 2.3

```
  km = Ø;
  for all s ∈S rhs(s) = g(s) = ∞;
  rhs (sgoal) = 0;
  U.Insert(sgoal, CalculateKey(sgoal));

procedure UpdateVertex(u)
  if (u ≠ sgoal) rhs(u) = mins'∈Succ(u)(c(u, s') + g(s'));
  if (u ∈ U) U.Remove(u);
  if (g(u) ≠ rhs(u)) U.Insert(u , CalculateKey(u));

procedure ComputeShortestPath()
  while (U.TopKey()<CalculateKey(sstart) OR rhs(sstart) +
g(sstart))
    kold = U.TopKey();
    u = U.Pop();
    if (kold < CalculateKey(u))
      U.Insert(u, CalculateKey(u);
    else if(g(u) > rhs(u))
      g(u) = rhs(u);
      for all s ∈ Pred(u) Update Vertex(s);
    else
      g(u) = ∞;
      for all s ∈ Pred(u) ∪ {u} Update Vertex(s);

procedure Main()
  slast = sstart;
  Initialize();
  ComputeShortestPath();
  while (start ≠ sgoal)
    if (g(sstart)) = ∞) then there is no known path
    sstart = arg mins' ∈ Succ(s-start)(c(sstart, s’) + g(s’))
    Move to sstart;
    Scan graph for changed edge costs;
```

Продовження лістингу 2.3

```
if (any edge costs changed)
 km = km + h(slast, sstart);
 slast = sstart;
 for all directed edges (u, v) with changed edge costs
   Update the edge cost c(u, v);
   UpdateVertex (u);
 ComputeShortestPath();
```

2.2.3 Оптимізація алгоритму D* Lite

Прикладом оптимізації може бути умова завершення ComputeShortestPath(), яка може бути змінена, щоб зробити ComputeShortestPath() більш ефективною. Як зазначено, ComputeShortestPath() припиняється, коли початкова вершина локально консистентна і її ключ менше або дорівнює U.TopKey (). Тим не менш, ComputeShortestPath() вже може завершитися, коли початкова вершина не є локально консистентною і її ключ менше або дорівнює U.TopKey().

Щоб зрозуміти, чому це так, припустимо, що початкова вершина локально несумісна і її ключ менше або дорівнює U.TopKey(). Потім його ключ повинен бути рівним U.TopKey(), оскільки U.TopKey() є найменшим ключем будь-якої локально несумісної вершини. Таким чином, ComputeShortestPath() може розширити початкову вершину далі, у цьому випадку вона встановить своє значення g на своє значення rhs. Початкова вершина стає локально послідовною, її ключ менше або дорівнює U.TopKey(), і ComputeShortestPath() при цьому припиняється.

У цей момент часу g-значення початкової вершини дорівнює його відстані до цілі. Таким чином, ComputeShortestPath() вже може завершитися, коли початкова вершина не локально недоконсістентна, а її ключ менше або дорівнює U.TopKey(). У цьому випадку початкова вершина може залишатися локально

неконсистентною після завершення ComputeShortestPath() і, таким чином, її g-значення не може бути рівним відстані від мети (але його значення rhs є). Це не є проблемою, оскільки значення g не використовується для визначення того, як робот повинен рухатися. Псевдокод оптимізованого алгоритму D* Lite поданий у лістингу 2.4.

Лістинг 2.4 – Псевдокод оптимізованого алгоритму D* Lite

```
procedure CalculateKey(s)
  return [min (g(s), rhs(s)) + (sstart, s) + km; min(g(s),
rhs(s))];
procedure Initialize()
  U = ∅;
  km = 0;
  for all s ∈ S rhs(s) = g(s) = ∞;
  rhs(sgoal) = 0;
  U.Insert(sgoal, [h(sstart, sgoal); 0]);
procedure UpdateVertex(u)
  if(g(u) ≠ rhs(u) AND u ∈U) U.Update(u, CalculateKey(u));
  else if(g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, Calculate
Key(u));
  else if(g(u) = rhs() AND u ∈U) U.Remove(u);
procedure ComputeShortestPath()
  while (U.TopKey() < CalculateKey(sstart) OR rhs(sstart) >
g(sstart))
  u = U.Top();
  kold = U.TopKey();
  knew = CalculateKey());
  if(kold < knew)
    U.Update(u, knew);
  else if(g(u) > rhs(u))
    g(u) = rhs(u);
    U.Remove(u);
```

Продовження лістингу 2.4

```
    for all s ∈ Pred(u)
      if (s ≠ sgoal) rhs(s) = min(rhs(s), c(s, u) + g());
      UpdateVertex(s);
  else
    gold = g(u);
    g(u) = ∞;
    for all s ∈ Pred(u) ∪ u}
    if (rhs(s) = c(s, u) + gold)
      if(s ≠ sgoal) rhs(s) = mins' Succ(s)(c(s, s') + g(s'));
    Update Vertex(s);
procedure Main()
  Slast = sstart;
  Initialize();
  ComputeShortestPath();
  while (sstart ≠ sgoal)
  if (rhs (sstart) = ∞) then there is no known path
  sstart = arg mins'∈ Succ(s-start)(c(sstart, s') + g(s'));
  Move to sstart
  Scan graph for changed edge costs;
  if any edge costs changed
    km = km + h(slast, sstart);
    slast = sstart;
    for all directed edges (u, v) with changed edge costs
      cold = c(u, v);
      Update the edge cost c(u, v);
      if (cold > c(u, v))
        if (u + sgoal) rhs(u) = min(rhs(u), c(u, v) + g(v));
      else if (rhs(u) = cold + g(v))
        if (u ≠ sgoal) rhs(u) = mins' ∈ Succ(u)(c(u,s') + g(s'));
      UpdateVertex(u);
    ComputeShortestPath();
```

2.2.4 Практичне застосування алгоритму D* Lite

Інкрементні методи пошуку, такі як DynamicSWSF-FP, D* та D* Lite, в даний час використовуються в штучному інтелекті. Вони повторно використовують інформацію з попередніх пошуків, щоб знайти рішення для подібних пошукових завдань набагато швидше, ніж це можливо, вирішуючи кожне завдання пошуку з нуля [13].

Алгоритми пошуку шляху мають велике число можливих засобів використання. Одним із них є покрокові стратегії, які розглядаються у даній роботі, де алгоритми пошуку шляху використовуються для визначення оптимального шляху для ігрових персонажів між двома плитками на карті. Одна із найперспективніших сфер використання алгоритмів пошуку шляху – це робототехніка.

Прикладом є автономна навігація транспортних засобів – процес переміщення транспортних засобів з однієї позиції в іншу без нагляду оператора. Планування шляхів, як підхід штучного інтелекту до автономної навігації транспортним засобом дозволяє включити апріорне знання навколишнього середовища. Це дозволяє безпілотним транспортним засобам діяти розумно, підкоряючись обмеженням, а не просто реагувати на перешкоди на своєму шляху.

D* широко використовувався на реальних роботах, включаючи HMMWV. В даний час він також інтегрован в прототипи Марс Ровер – Джозеф Карстен і Арт Ранкін з лабораторії реактивного руху НАСА встановили версію D*, використовуючи елементи D* Lite на марсоходах «Spirit» і «Opportunity», і вперше дозволили керувати ровером на Марсі в лютому 2007 року після його тестування на марсі в листопаді 2006 року [19].

D* Lite коротше D*, при порівнянні пріоритетів використовує лише один критерій, що спрощує підтримку пріоритетів і не потребує вкладених if-операцій зі складними умовами, які займають до трьох рядків, що спрощує аналіз потоку

програми. Ці властивості також дозволяють легко розширювати його, наприклад, використовувати нестандартні евристики і різні критерії розрішення нічиїх для отримання більшої ефективності.

D* Lite можна також використовувати для реалізації жадібного картування, проста, але потужна стратегія картування, що неодноразово використовувалася на мобільних роботах різними дослідними групами. Жадібне картування дискретизує рельєф в клітинах з рівномірною роздільною здатністю, а потім завжди переміщує робота від його поточної клітки до найближчої клітки з невідомою прохідністю, доки не буде відображено рельєф. У цьому випадку граф є пов'язаною сіткою. Витрати на її краї спочатку дорівнюють одиниці. Вони змінюються до нескінченності, коли робот виявляє, що їх не можна пройти. Існує ще одна вершина, яка з'єднана з усіма вершинами сітки. Вартості ребер від неї спочатку дорівнює одиниці. Вони переходять у нескінченність після того, як була відвідана відповідна вершина сітки. Можна реалізувати жадібне картування, застосовуючи D* Lite до цього графа з поточною вершиною s_{start} та додатковою вершиною s_{goal}.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Виходячи з мети кваліфікаційної роботи, шаблону гри у жанрі покрокової стратегії, який розробляється має реалізовувати алгоритмом пошуку шляхів для ігрових персонажів.

Для розробки програмного програмного засобу треба скласти його проект, а також підібрати програмне забезпечення, яке задовольнить потребам розробки програмного засобу. Для реалізації шаблону гри потрібно використати ігровий движок, а також розробити скрипти на одній з мов програмування. Для написання скриптів потрібна середа розробки (IDE).

3.1 Вибір і обґрунтування мовних і програмних засобів

3.1.1 Опис інструментального засобу Unity 3D

Unity – це крос-платформний ігровий движок, розроблений компанією Unity Technologies, вперше оголошений і випущений у червні 2005 року на Apple Inc. Починаючи з 2018 року, движок був розширений для підтримки більш ніж двадцять п’ять платформ. Движок можна використовувати для створення тривимірних, двовимірних, віртуальних ігор та ігор у доповненій реальності, а також моделювання та інших задач. Движок використовується поза галуззю відеоігор, наприклад у кіно, автомобільній промисловості, архітектурі, інженерії та будівництві [16].

Unity надає користувачам можливість створювати ігри як в 2D, так і в 3D, і движок пропонує основний API створення скриптів у C#, як для редактора Unity у вигляді плагінів, так і для самих ігор, а також функцію перетягування. До того, як C# був основною мовою програмування, яка використовувалася для движка, Unity раніше підтримував Boo, якиц був видален з випуском Unity 5, і версію JavaScript під назвою UnityScript, яка була застаріла у серпні 2017 року після

виходу Unity 2017.1, на користь C#. У 2D-іграх Unity дозволяє імпортувати спрайти а також має розширений 2D-рендерінг світу. Для 3D-ігор Unity дозволяє специфікувати стискання текстур, mipmaps і роздільну здатність для кожної платформи, яку підтримує ігровий движок, і надає підтримку для відображення рельєфу, відображення паралакса, екранування простору навколишнього середовища (SSAO), динамічного тіні, які використовують тіньові карти, ефекти візуалізації текстури та повноекранні.

Станом на 2021 рік, Unity був використаний для створення приблизно половини нових мобільних ігор на ринку і 60% контенту для розширеної та віртуальної реальності.

3.1.2 Опис мовного засобу C#

C# – це мова програмування загального призначення, яка охоплює сильну типізацію, імперативні, декларативні, функціональні, загальні, об'єктно-орієнтовані (на основі класів) та компонентно-орієнтовані дисципліни програмування. Він був розроблений в 2000 році корпорацією Майкрософт в рамках своєї ініціативи .NET і пізніше схвалений як стандарт Ecma (ECMA-334) і ISO (ISO / IEC 23270: 2018). C# є однією з мов програмування, призначених для спільної мовної інфраструктури CLI (Common Language Infrastructure) [17].

3.1.3 Опис інструментального засобу Rider

Rider є крос-платформенним .NET IDE: ви можете використовувати його на Windows, Mac і Linux. Rider дозволяє розробляти програми ASP.NET, .NET Core, .NET Framework, Xamarin і Unity.

Rider – це комбінація декількох технологій JetBrains: вона ставить потужну .NET підтримку ReSharper в оболонці IntelliJ IDEA, а також включає

функціональність WebStorm (front-end) і DataGrip (SQL і управління базами даних).

Мови, що підтримуються Rider, включають C #, VB.NET, F #, синтаксис Razor, JavaScript, TypeScript, XAML, HTML, CSS, SCSS, LESS, JSON і SQL.

Набір функцій Rider включає в себе 2000+ перевірок коду в реальному часі на підтримуваних мовах, 500+ рефакторингів, різноманітних допоміжних засобів кодової навігації, відладчик, багату допомога в кодуванні та багато більш просунутих функцій IDE.

3.2 Опис структури програми

Проект із кодом програмного засобу знаходиться у проекті ігрового движка Unity 3D. Структура проекту представлена у ієрархічному вигляді на панелі файлів проекту у середовищі розробки на рисунку 3.1.

Клас, безпосередньо відповіщаючий за реалізацію алгоритму пошуку шляху – це DStarLitePathFinder. Цей класс має один публічний метод – FindNextStepInPath, який приймає початкову та кінцеву плитки (клас Tile) та рухомого персонажа, для якого потрібно створити шлях (клас Unit). Метод FindNextStepInPath проводить пошук оптимального шляху та повертає наступну плитку на яку повинен переміститися персонаж та скільки в нього залишиться очок руху після цього переміщення. При повторному визові цього метода буде використана інформація з попереднього визову, відповідно лозиці алгоритму D* Lite.

Ініціалізація ігрових об’єктів проходить у класі MapManager, який кріпиться до об’єкту у сцені Unity 3D, що дозволяє описати логику ініціалізації у методі Start. Також це дозволяє винести деякі настройки до інтерфейсу Unity 3D, що дозволяє спростити повторне тестування програми.

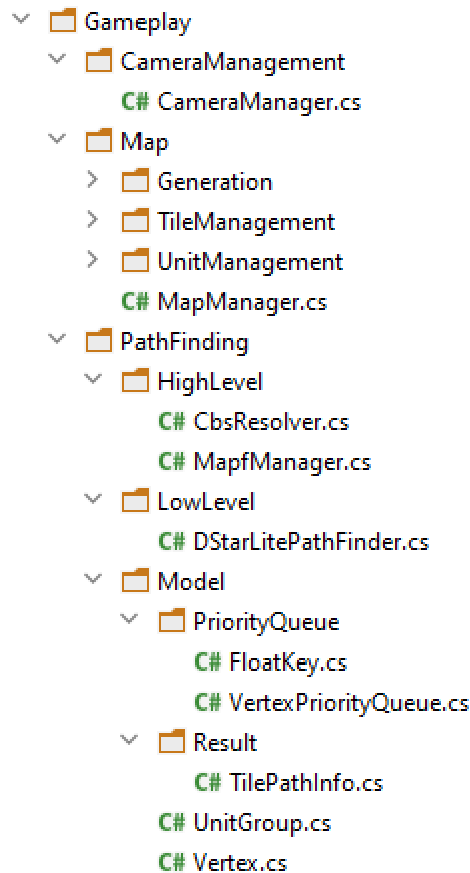

Рисунок 3.1 – Структура проекту

Під час ініціалізації відбувається створення карти заданого розміру (параметр Size), яка представлена класом Map, який містить двомірний масив плиток, представлених класом Hexagon та допоміжну інформацію, таку як словник сусідніх плиток, який містить всі сусідні плитки для кожної плитки на карті, це зроблено для того щоб зменшити навантаження на програму, адже пошук сусідніх плиток виконується дуже часто. На кожній плитці карти виконується генерація ігрових об’єктів, відповідно до конфігурації GameLoader.

Ігрові об’єкти, розміщені на карті, представлені класами Hexagon, Feature, Landscape та юніт. Hexagon представляє собою плитку на карті та є базою для розміщення інших об’єктів. Feature та Landscape – статичні сутності, які представляють ландшафт ігрової карти та впливають на вартість переміщення по плиткам карти. Клас Unit представляє рухомого персонажа, який може

переміщуватись по карті, витрачаючи очки руху. Клас Unit має методи MoveTo, який відповідає за використання алгоритму пошуку шляху і переміщення по знайденому їм шляху, а також HandleTurnEnd, який виконується у кінці кожного ходу і потрібен для оновлення очок руху персонажа.

Для створення ігрових об’єктів використовується шаблон «фабрика» – за це відповідають класи HexagonFactory, FeatureFactory, LandscapeFactory, UnitFactory. Ці класи мають методи, що створюють нові об’єкти відповідних класів, з заданими характеристиками, так клас LandscapeFactory має методи NewPlain, NewHill, NewMountain, які створюють об’єкти класу Landscape з вартістю руху 1, 2 та 8 відповідно. Також класи фабрики проводять загрузку префабів (prefabs). Префаб – це механізм Unity 3D, який дозволяє створювати шаблони об’єктів для подальшого пере-використання. Зроблені префаби знаходяться у проекті Unity 3D та загружаються за допомогою допоміжного класу ResourceLoader, який має метод LoadPrefab, що загружає префаб по заданому відносному путі до файлу префабу. Відносні путі до файлів знаходяться у класі Constants.

Класи Vertex, VertexPriorityQueue, FloatKey та PathFindingResult потрібні для виконання алгоритму пошуку шляху. Клас Vertex містить інформацію про g та rhs значення одної з плиток графу, для одного з персонажів а також використовується для обчислення евристики. Через те що на одну плитку карти може приходитись декілька об’єктів класу Vertex – у класі Hexagon є словник Dictionary<Unit, Vertex>, який дозволяє знайти об’єкт класу Vertex для необхідного персонажа. Клас VertexPriorityQueue поширює SimplePriorityQueue з OptimizedPriorityQueue методом TopKey, який повертає ключ з найменшим пріоритетом. Клас FloatKey представляє собою ключ/пріоритетність для приоретизованої черги, яка використовується алгоритмом пошуку шляху D* Lite. Він містить основний та вторинний ключі, а також реалізує інтерфейс IComparable і його метод CompareTo, для зрівняння двох ключів. PathFindingResult потрібен для того, щоб повернути два об’єкти (об’єкт класу

Hexagon – NextStep, що представляє собою наступну плитку у маршруті персонажа, а також об'єкт класу float – MovePointsLeft, що представляє собою число очок руху, що залишиться у персонажа після переміщення) з методу FindNextStepInPath інтерфейсу IPathFinder.

За взаємодію з інтерфейсом користувача відповідають класи CameraMovement, EndTurnHandler, MouseInputHandler та MouseInputListener. Клас CameraMovement обробляє натискання клавіш WSAD, для пересування камери, а також прокрутку колеса миші для регулювання приближення. Клас EndTurnHandler відповідає за оброблення натиску на кнопку кінця ходу для визиває метод HandleTurnEnd для всіх об'єктів класу Unit. MouseInputHandler та MouseInputListener потрібні для переміщення ігрових персонажів – вони оброблюють натискання лівої клавіші миші і, якщо перший клік був по плитці з ігровим персонажем, а другий по пустій плитці, то персонаж почне рухатися до вказаної пустої плитки.

3.3 Опис коду алгоритму пошуку шляху

Клас DStarLitePathFinder, який відповідає за виконання алгоритму пошуку шляху має метод FindNextStepInPath, що приймає стартову та кінцеву плитки та ігрового персонажа, для якого потрібно розробити маршрут руху між цими плитками. Цей метод повертає об'єкт класу PathFindingResult, що містить плитку з найденого шляху, на яку персонаж може перейти на даному кроці, а також число очок руху, яке залишиться у персонажа після цього переходу.

У класі DStarLitePathFinder метод FindNextStepInPath, код якого поданий у лістингу 3.1, перевіряє, чи рухається персонаж по вже знайденому шляху, або ні, використовуючи умову goalVertex == null || !goalVertex.Equals(goal.GetVertex(unit)), якщо умова повертає істину то потрібно проводити пошук з нуля, у зворотньому випадку можна лише оновити дані для тих плиток, для яких змінилась вартість руху. Після того, як виконався

пошук шляху (або оновлення шляху) перевіряється rhs значення для стартової вершини, якщо воно дорівнює нескінченності, то знайти путь між заданими точками неможливо і метод повертає null.

Потім, для всіх об’єктів класу Vertex у приоретизованій черзі виконується присвоєння OldMovementCost, яке потрібне для того, щоб на при наступних викликах алгоритму можна було визначити плитки, вартість переміщення яких змінилась. За цим слідує визначення плитки з найденого шляху, на яку персонаж може перейти на даному кроці, а також числа очок руху, яке залишиться у персонажа після цього переходу, які повертаються методом у вигляді об’єкту PathFindingResult.

Лістинг 3.1 – Код методу FindNextStepInPath в класі DStarLitePathFinder

```
public Tile FindNextStepInPath(Tile start, Tile goal) {
    if (sGoal != null && sGoal.Tile.Equals(goal)) {
        UpdatePathIfFoundChanges();
    } else {
        StartMovement(start, goal);
    }

    if (float.IsPositiveInfinity(sStart.Rhs)) {
        LogManager.Warn($"No path exists between points
{sStart} and {sGoal} for unit ", unit);
        return null;
    }

    sStart = ComputeNextMove(sStart);

    return sStart.Tile;
}
```

Метод StartMovement класу DStarLitePathFinder, код якого поданий у лістингу 3.2, проводить встановлення значень полів startVertex, goalVertex та

lastVertex, які потрібні для розрахування шляху та визиває методи Initialize, який ініціалізує дані пошуку та ComputeShortestPath, який проводить пошук шляху.

Лістинг 3.2 – Код методу StartMovement в класі DStarLitePathFinder

```
private void StartMovement(Tile start, Tile goal) {
    sStart = start.GetVertex(unit);
    sGoal = goal.GetVertex(unit);

    sLast = sStart;
    Initialize();
    ComputeShortestPath();
}
```

Метод ComputeShortestPath класу DStarLitePathFinder, код якого поданий у лістингу 3.3, проводить оновлення приоретизованої черги, до тих пір поки виконується умова priorityQueue.TopKey() < CalculateKey(startVertex) || startVertex.rhs > startVertex.g. Він бере найменший ключ (kOld) та вершину з найменшим ключем (topVertex) з черги та перевіряє, чи не збільшився ключ для цієї плитки, якщо так, то він оновлює вибрану вершину.

Якщо ні, то перевіряється умова topVertex.g > topVertex.rhs, якщо вона є істиною, то g значення topVertex прирівнюється rhs значенню, topVertex видаляється з черги та проводиться оновлення інформації про всіх попередників topVertex. Якщо і ця умова не виконалася, g значення topVertex присвоюється нескінченності і проводиться оновлення інформації про topVertex та всіх її попередників.

Лістинг 3.3 – Код методу StartMovement в класі DStarLitePathFinder

```
private void ComputeShortestPath() {
    while (priorityQueue.TopKey() < CalculateKey(sStart) ||
sStart.Rhs > sStart.G) {
        Vertex topVertex = priorityQueue.First;
```

Продовження лістингу 3.3

```
        FloatKey kOld = priorityQueue.TopKey();
        FloatKey kNew = CalculateKey(topVertex);
        if (kOld < kNew) {
            priorityQueue.UpdatePriority(topVertex, kNew);
        } else if (topVertex.G > topVertex.Rhs) {
            topVertex.G = topVertex.Rhs;
            priorityQueue.Remove(topVertex);
            foreach (Vertex predecessor in
GetPredecessors(topVertex)) {
                if (!predecessor.Equals(sGoal)) {
                    predecessor.Rhs = CalculateRhs(predecessor,
topVertex);
                }
                UpdateVertex(predecessor);
            }
        } else {
            float gOld = topVertex.G;
            topVertex.G = float.PositiveInfinity;
            foreach (Vertex predecessor in
GetPredecessorsWithSelf(topVertex)) {
                if (predecessor.Rhs ==
CalculateEdgeCost(topVertex) + gOld && !predecessor.Equals(sGoal))
{
                    predecessor.Rhs =
CalculateUpdatedRhs(predecessor);
                }
                UpdateVertex(predecessor);
            }
        }
    }
}
```

Метод UpdatePathIfFoundChanges класу DStarLitePathFinder, код якого поданий у лістингу 3.4, проводить пошук вершин, вартість яких змінилась з моменту попереднього пошуку, і, якщо знаходить хочь одну таку вершину, визиває метод UpdateShortestPath, який оновлює маршрут.

Лістинг 3.4 – Код методу UpdatePathIfFoundChanges в класі DStarLitePathFinder

```
private void UpdatePathIfFoundChanges() {
    List<Vertex> changed = FindChangedVertices();
    if (changed.Count != 0) {
        LogManager.Debug("Performing shortest path update for
changed vertices:", changed);
        UpdateChangedVertices(changed);
        unit.ChangedVertices.AddRange(changed);

        ComputeShortestPath();
    }
}
```

Метод UpdateShortestPath класу DStarLitePathFinder, код якого поданий у лістингу 3.5, оновлює інформацію про всі вершини, вартість яких змінилась з моменту попереднього пошуку та визиває метод ComputeShortestPath для оновлення оптимального маршруту.

Лістинг 3.5 – Код методу UpdateShortestPath в класі DStarLitePathFinder

```
private void UpdateChangedVertices(IEnumerable<Vertex>
changed) {
    additionalHeuristic +=
sLast.CalculateHeuristicCost(sStart);
    sLast = sStart;
    foreach (Vertex changedVertex in changed) {
        if (!changedVertex.Equals(sGoal)) {
```

Продовження лістингу 3.5

```
            float oldMovementCost =
changedVertex.GetOldMovementCost();
            float newMovementCost =
CalculateEdgeCost(changedVertex);
            float oldG = changedVertex.G;
            changedVertex.Rhs =
CalculateUpdatedRhs(changedVertex);
            changedVertex.G = changedVertex.Rhs;
            UpdateVertex(changedVertex);
            foreach (Vertex neighbour in
GetNeighbours(changedVertex)) {
                if (oldMovementCost > newMovementCost) { //
Cost decreased
                    neighbour.Rhs = CalculateRhs(neighbour,
changedVertex);
                } else if (neighbour.Rhs == oldMovementCost +
oldG) { // Cost increased and current Rhs equals
                    neighbour.Rhs =
CalculateUpdatedRhs(neighbour);
                }
                UpdateVertex(neighbour);
            }
        }
    }
}
```

Для того щоб адаптувати роботу алгоритму пошуку низького рівня для мультиагентного пошуку, необхідно зробити декілька дороботок. По перше треба розробити лічильник кроків, який буде збільшуватися кожен раз коли закінчується хід. Ця інформація використовується при передачі найдених шляхів між низьким та високим рівнями пошуку, для ідентифікації конфліктів.

Також, при генерації нового шляху агента формується шлях на низькому рівні, він передається на високий рівень. Після цього на високому рівні виконується перевірка: якщо шлях має конфлікт з іншими шляхами, необхідно створити комбіновану групу одиниць і використовувати CBS для вирішення конфліктів. У іншому випадку – необхідно створити нову групу одиниць і зберегти шлях.

При додаванні одиниці до групи агентів (коли новий шлях агента має конфлікти з одним із членів групи), або при видаленні підрозділу з групи підрозділів (коли один із учасників змінює ціль) необхідно:

– видалити обмеження для членів групи;

– перерахувати шляхи для членів групи;

– перерозподілити групи, якщо це необхідно (необхідно перевірити, чи всі учасники все ще повинні бути в одній групі, або її можна розділити);

– використати алгоритм CBS для вирішення конфліктів.

Реалізація обмежень для низького рівня виглядає наступним чином:

– для кожного потенційного рішення CBS створює копію даних пошуку шляхів (DStarLitePathFinder та Vertices) на високому рівні;

– коли DStarLitePathFinder отримує обмеження: "плитка (a, b) на кроці n" необхідно скинути відповідну "Вершину" визвав метод Vertex.Reset(), код якого поданий у лістингу 3.6;

– коли всі конфлікти вирішені, оптимальне рішення (те, яке надає шлях без конфліктів) встановлюється як дані про пошук шляху агента.

Лістинг 3.6 – Код методу Reset в класі Vertex

```
public void Reset() {
    G = float.PositiveInfinity;
    Rhs = float.PositiveInfinity;
}
```

Для контролю роботи високого та низького рівнів пошуку був розроблений класс MapfManager. Він відповідає за координацію груп агентів, та пошук конфліктів. Його основним методом є AddPath, код якого поданий у лістингу 3.7, який визивається коли агент знаходить путь до нової цілі.

Лістинг 3.7 – Код методу AddPath в класі MapfManager

```
public void AddPath(Unit unit, List<TilePathInfo> path) {
    HashSet<UnitGroup> affectedGroups =
GetAffectedGroups(unit, path);
    if (affectedGroups.Count == 0) {
        if (unit.PathfindingGroup == null) {
            CreateGroup(unit);
        }
        pathMap[unit] = path;
    } else {
        RecalculatePaths(affectedGroups.SelectMany(value =>
value).Append(unit));
    }
}
```

Пошук конфліктів відбувається у методі GetAffectedGroups, код якого поданий у лістингу 3.8, який приймає найдений низьким рівнем шлях і повертає усі групи, які маєть конфлікти з цим шляхом.

Лістинг 3.8 – Код методу GetAffectedGroups в класі MapfManager

```
private HashSet<UnitGroup> GetAffectedGroups(Unit unit,
List<TilePathInfo> path) {
    HashSet<UnitGroup> affectedGroups = new
HashSet<UnitGroup>();
    foreach (UnitGroup unitGroup in unitGroups) {
        bool isAffected = false;
        List<Unit>.Enumerator enumerator =
```

Продовження лістингу 3.8

```
unitGroup.GetEnumerator();
        while (!isAffected && enumerator.MoveNext()) {
            isAffected =
pathMap[enumerator.Current].Any(path.Contains);
        }
        enumerator.Dispose();
        if (isAffected) {
            affectedGroups.Add(unitGroup);
        }
    }
    return affectedGroups;
}
```

Для вирішення конфліктів алгоритмом CBS був розроблений клас CbsResolver. Його основним методом є ResolveConflicts, код якого поданий у лістингу 3.9, який приймає словник агентів та їх шляхів, і повертає такий самий словник, але з виправленими шляхами.

Лістинг 3.9 – Код методу ResolveConflicts в класі CbsResolver

```
public Dictionary<Unit, List<TilePathInfo>>
ResolveConflicts(Dictionary<Unit, List<TilePathInfo>>
unitPathMap) {
    List<HashSet<TilePathInfo>> constraints = new
List<HashSet<TilePathInfo>>();
    int cost = GetTotalCost(unitPathMap);
    Dictionary<Unit, List<TilePathInfo>> open = new
Dictionary<Unit, List<TilePathInfo>>();
    while (open.Count != 0) {
        P = FindBestNode(open);
        ValidatePaths(open);
        if (ChackConflicts(P)) {
            return P.solution;
```

Продовження лістингу 3.9

```
        }
        C = FindConflict(open);
        foreach (Unit ai in C) {
            A = new Node();
            A.constraints = P.constraints + C;
            A.solution = P.solution;
            Update(A.solution);
            A.cost = SIC(A.solution);
            if (A.cost < float.PositiveInfinity) {
                open.Add(A);
            }
        }
    }
}
```

3.4 Опис інтерфейсу розробленої програми

Для відображення роботи програми була створена сцена GameScene, яка містить об'єкти Main Camera, Loader, Canvas та EventSystem, об'єкт Canvas. При цьому містить об'єкт EndTurnButton, який у свою чергу містить об'єкт Text. Структура сцени показана на рисунку 3.2

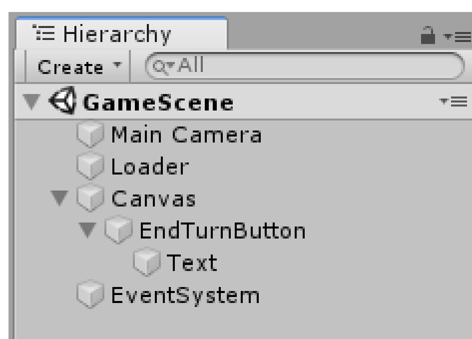

Рисунок 3.2 – Структура створеної сцени у Unity 3D

Об’єкт Main Camera виконує роль точки зору для користувача, за зміну якої відповідає клас CameraMovement. Конфігурація об’єкту Main Camera представлена на рисунку 3.3

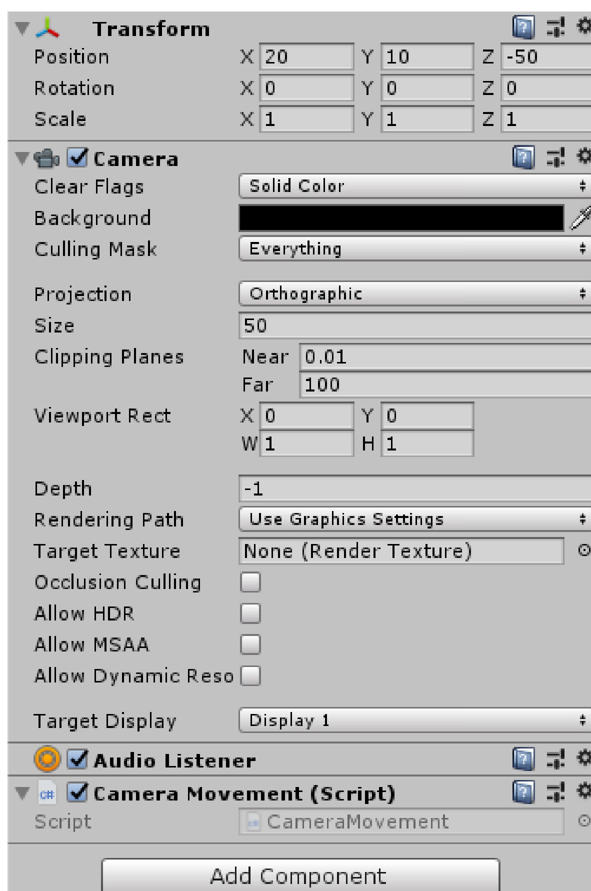

Рисунок 3.3 – Інтерфейс конфігурації об’єкту Main Camera

Об’єкт Loader містить скрипт класу GameLoader, описаний у розділі 3.2. Інтерфейс конфігурації GameLoader представлено на рисунку 3.4.

Об’єкти Canvas та Event System є системними об’єктами Unity 3D. Canvas відповідає за відображення інтерфейсу гри, який у розробленій програмі представлений кнопкою End Turn, натискання якої приводить до оновлення очок руху юнітів. Event System відповідає за обробку дій користувача – він посилає події натискання на кнопки клавіатури та пересування миші, які приходять до класу MouseInputListener.

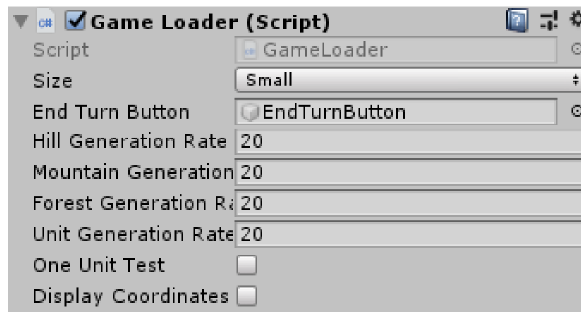

Рисунок 3.4 – Інтерфейс конфігурації GameLoader

Після запуску сцени користувач бачить перед собою згенеровану ігрову карту. На ній зеленими та червоними квадратами будуть відповідно відображені проходимі та не проходимі плитки карти. Білими кругами відображені агенти. Приклад такої карти приведено на рисунку 3.5.

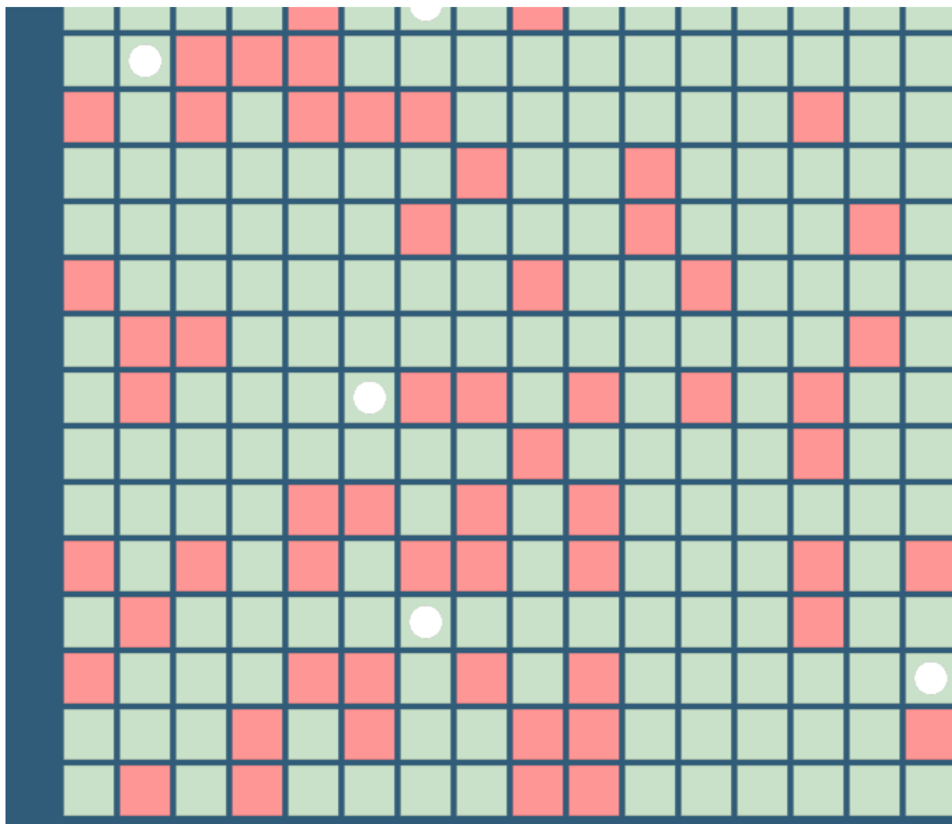

Рисунок 3.5 – Згенерована ігрова карта

При переміщенні одного з рухомих персонажів з’являється інформація про те, які плитки були оброблені на даному кроці, включаючи інформацію про G та Rhs значення з алгоритму D* Lite. Синім кольором помічений наступний крок агента, а жовтим – його ціль. Приклад відображення даної інформації приведено на рисунку 3.6.

Рисунок 3.6 – Відображення інформації про виконання алгоритму пошуку шляху

3.5 Інтерпретація результатів

У результаті виконання кваліфікаційної роботи було розроблено шаблон гри у жанрі покрокової стратегії з реалізованим двох рівневим алгоритмом пошуку шляхів для ігрових персонажів на основі алгоритмів CBS та D* Lite.

Програмний засіб повинен візуально відображати результати кожного виконання алгоритму для спрощення оцінки результату. Також для оцінки результату був проведений аналіз часу, витраченого на роботу алгоритму пошуку шляху. Тестування проводилося на карті розміром 20x20, спочатку 4 ігрових персонажів пройшли карту по діагоналям, використовуючи розроблений алгоритм пошуку шляху. Після цього алгоритм пошуку був змінений таким чином, щоб він не використовував дані попередніх пошуків і ігровий персонаж знову пройшов той самий путь. В обох запусках програми, проводилися заміри витрат часу на виконання алгоритму пошуку шляху на кожному кроці. Результат можна побачити на рисунку 3.7, де красним відділено час виконання алгоритму в нормальному стані, а синім – без використання даних попередніх пошуків.

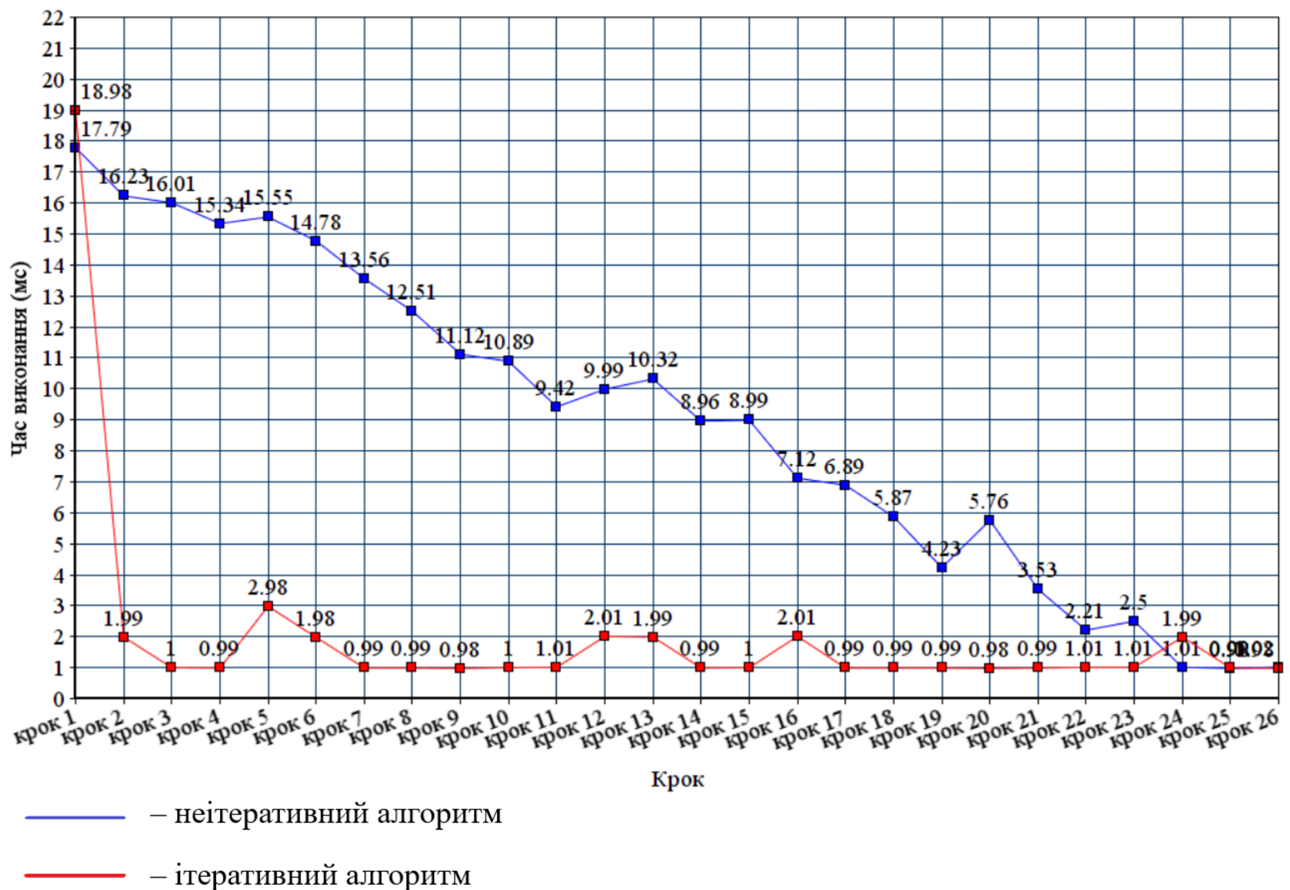

Рисунок 3.7 – Графік часу виконання алгоритму пошуку

На графіку видно, що, навіть при тому що виконання обох версій алгоритму – ітеративної та не ітеративної, потребує однакового часу на першому кроці, на наступних кроках ітеративний алгоритм має перевагу у часі виконання до 16 разів.

Якщо ці дані застосувати для одночасного переміщення більшої кількості персонажів, можна получити приріст у швидкості роботи програми на декілька порядків, навіть на карті що змінюється.

Результати аналізу показали що розроблений ітеративний алгоритм має значно більшу швидкість за стандартні алгоритми на основі A*, що часто використовуються для вирішення задач пошуку шляху.

Результати реалізації програми були проаналізовані, за допомогою порівняння часу, витраченого на роботу алгоритму пошуку шляху при ітеративній та не ітеративній реалізаціях.

Розроблену програму можна адаптувати під більш складні завдання, такі як пошук шляху в частково невідомій місцевості (Задача канадського мандрівника), а також «розумного» пошуку, коли рухомий агент може уникати, або навпаки пріоритезувати частки шляху по наперед заданим правилам.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи був проведений аналіз предметної області і був розроблений шаблон гри у жанрі покрокової стратегії з реалізованим мультиагентним алгоритмом пошуку шляхів для ігрових персонажів.

В результаті проведеного аналізу особливостей різних типі алгоритмів пошуку шляху було вирішено використовувати алгоритми CBS та D* Lite як базу для алгоритму пошуку що буде розроблятись.

Інкрементні методи пошуку, в даний час використовуються в різних сферах штучного інтелекту. Вони повторно використовують інформацію з попередніх пошуків, щоб знайти рішення для подібних пошукових завдань набагато швидше, ніж це можливо, вирішуючи кожне завдання пошуку з нуля.

Розроблений програмний засіб надає користувачу можливість переміщати ігрових персонажів по згенерованій карті з урахуванням поставлених обмежень, таких як переміщення інших персонажів на мапі. Також, для спрощення оцінки результату реалізована програма візуально відображає результати кожного виконання алгоритму.

Результати реалізації програми були проаналізовані, за допомогою порівняння часу, витраченого на роботу алгоритму пошуку шляху при ітеративній та не ітеративній реалізаціях. Результати аналізу показали що розроблений ітеративний алгоритм має значно більшу швидкість за стандартні алгоритми на основі A*, що часто використовуються для вирішення задач пошуку шляху.

Розроблену програму можна адаптувати під більш складні завдання, такі як пошук шляху в частково невідомій місцевості (Задача канадського мандрівника), а також «розумного» пошуку, коли рухомий агент може уникати, або навпаки пріоритезувати частки шляху по наперед заданим правилам.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Botea A., Buzi B., Buro M. Pathfinding in games. Dublin: IBM Research, 2012. 31 p.

2. Stout B. Smart moves: intelligent path-finding. *Game Development Magazine*. 1996. №5. P. 5–35.

3. Bultiko W., Bjornson J., Sturvenant N. R. Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding. *Journal of Artificial Intelligence Research.* 2010. №7. P. 270–303.

4. Millington I., Fange J. Artificial Intelligence for Games. Florida: CRC Press, 2009. 870 p.

5. Tsui S., Shi H. A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security.* 2011. № 2. P. 33–39.

6. Single Source Shortest Paths Problem: Dijkstra's Algorithm. URL: http://lcm.csa.iisc.ernet.in/dsa/node162.html (Last accessed: 24.04.2021).

7. Sun S., Koenig S., Yeo W. Generalized Adaptive A *. *International Foundation for Autonomous Agents and Multiagent Systems.* 2008. № 3. P. 8–16.

8. Introduction to A* Pathfinding. URL: https://www.raywenderlich.com/3016-introduction-to-a-pathfinding (Last accessed: 24.04.2021).

9. Pelechano N., Fuentes S. Hierarchical path-finding for Navigation Meshes (HNA∗). *Computers & Graphics.* 2016. № 5. P. 10–20.

10. Boteya A., Mueller M., Schaeffer J. Near optimal hierarchical path-finding. *Journal of Game Development*. 2004. № 6. P. 21–42.

11. Nosrati M., Karimi R., Hasanwand H. A. Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches. *World Applied Programming*. 2012. № 1. P. 6–12.

12. Ramalingam G., Reps T. On the computational complexity of dynamic graph problems. Madison: University of Wisconsin−Madison, 1996. 277 p.

13. Koenig S., Tovey S., Smirnov Y. Performance Bounds for Planning in Unknown Terrain. *Artificial Intelligence*. 2003. № 11. P. 43–76.

14. Koenig S., Likhachev M. D* Lite. *Artificial Intelligence*. 2006. № 4. P. 16–32.

15. Koenig S., Likhachev M., Fursey. D. Lifelong Planning A*. *Elsevier Science*, 2004. P. 62–74.

16. Unity at 10: For better – or worse – game development. URL: https://arstechnica.com/gaming/2016/09/unity-at-10 (Last accessed: 24.04.2021).

17. C# language specification. Ecma International, 2012. 457 p.

18. Stern R., Sturtevant N., Fellner A. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search*. 2019. P. 8–14.

19. Surinek P., Fellner A., Stern R. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. *Symposium on Combinatorial Search*. 2016. P. 14-16.

20. Ma H., Wagner G., Fellner A. Multi-agent path finding with deadlines. *International Joint Conference on Artificial Intelligence*. 2018. P. 12-26.

21. Wagner G., Kang M., Choset H. Probabilistic path planning for multiple robots with subdimensional expansion. *International Conference on Robotics and Automation*. 2012. P. 7–21.

22. Ryan M. Exploiting subgraph structure in multi-robot path planning. *Journal of Artificial Intelligence*. 2008. № 5. P. 33–52.

23. Sharon G., Stern R., Fellner A. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*. 2015. № 4. P. 73–82.

24. Filatov V., Semenets V. Multiagent approach for managing the information space of corporate systems. *Innovative Technologies and Scientific Solutions for Industries*. 2018. № 4. P. 71–76

25. Filatov V. About One Approach to the Classification of Program Agents. *Modern Problems of Radio Engineering, Telecommunications, and Computer Science*. 2006. P. 410-411.