ВСТУП

Ігрова індустрія знаходиться в стані постійного розвитку, зумовленого не лише технологічними досягненнями, але й швидким розширенням інтересу споживачів до віртуальних розваг. У цьому контексті великою мірою визначається якість ігрового процесу, в якому ключову роль відіграє ігровий штучний інтелект. Створення ефективного ігрового ШІ важливо для забезпечення реалістичності та цікавості гри, а також для досягнення балансу між викликами та задоволенням гравців.

Ця робота спрямована на вивчення та дослідження методів створення ігрового штучного інтелекту для різних сценаріїв гри, зокрема, в контексті популярної платформи розробки Unity. Unity визнана своєю гнучкістю та можливостями для створення ігор різного жанру, а отже, розробка ефективних алгоритмів ШІ для цієї платформи стає ключовим завданням для розробників.

Мета цього дослідження полягає в аналізі та порівнянні різноманітних підходів до створення ігрового ШІ, а також в розгляді їх застосування для різних ігрових сценаріїв. У процесі дослідження будуть вивчені передові технології та методології, що дозволяють реалізовувати високоефективні та адаптивні алгоритми ШІ на базі Unity.

Робота розглядається як спроба внести вклад у розвиток галузі ігрового програмування, а також як практичний посібник для розробників, що прагнуть оптимізувати та удосконалити ігровий Штучний Інтелект у своїх проектах.

У цьому дослідженні будуть розглянуті такі алгоритми пошуку шляху, як Breadth First Search, Dijkstra, A*, Greedy Best First Search, Depth First Search, Bidirectional Search, Lee Algorithm та Dynamic Programming Maze. Окрім того, буде проведено аналіз алгоритмів прийняття рішень, таких як Behaviour Trees, State Machine, Timer Based та Random Select. Кожен із цих алгоритмів буде оцінено з точки зору його ефективності, продуктивності та придатності для різних ігрових сценаріїв.

Об'єктом дослідження є методи створення штучного інтелекту для різних ігрових сценаріїв на платформі Unity, зокрема алгоритми пошуку шляху та прийняття рішень для агентів.

Предметом дослідження є ефективність алгоритмів пошуку шляху та алгоритмів прийняття рішень, їх реалізація, простота та доречність у контексті Unity.

Наукова новизна дослідження полягає у комплексному аналізі та порівнянні різних підходів до створення штучного інтелекту в ігровій індустрії. Проводитиметься порівняння ефективності та продуктивності різноманітних алгоритмів пошуку шляху та прийняття рішень у контексті одного програмного середовища. Дослідження також розкриває нові можливості для адаптації та оптимізації цих алгоритмів у різних ігрових сценаріях.

Практичне значення роботи полягає у створенні рекомендацій для розробників, що працюють з Unity, які прагнуть покращити ігровий процес через ефективне використання штучного інтелекту. Результати дослідження можуть бути застосовані для оптимізації існуючих ігрових проектів, розробки нових ігор, а також для навчання нових спеціалістів у сфері ігрового програмування. Вивчення методів створення штучного інтелекту в іграх допоможе не лише покращити існуючі технології, але й прокласти шлях до нових інновацій у цій динамічній і постійно зростаючій галузі.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Виявлення проблематики

Основні проблеми, що можуть бути розглянуті в ході виконання роботи, включають оптимізацію алгоритмів ігрового ШІ, кількість пам’яті, що може бути використана, швидкодію алгоритму, складність реалізації та доцільність використання саме цього алгоритму в даній ігровій ситуації.

Оптимізація алгоритмів є критичним аспектом розробки ігрового ШІ, оскільки вона впливає на загальну продуктивність гри, використання ресурсів та реалістичність ігрового процесу. Нижче наведено ключові аспекти проблематики оптимізації алгоритмів:

- часова складність алгоритмів;
- використання пам'яті;
- зрозумілість та структура алгоритмів;
- оптимізація циклів;
- вибір структур даних;
- адаптивність до різних обсягів даних;
- баланс витрат і вигод.

Покращення зрозумілості та структури алгоритмів є важливим для забезпечення легкої модифікації та розширення. Це включає створення добре документованих, модульних і зрозумілих кодів, що полегшує їхнє вдосконалення та підтримку.

Уникання непотрібних ітерацій та оптимізація використання циклів допомагає зменшити витрати часу на виконання алгоритмів. Це може включати використання ефективних алгоритмічних конструкцій та уникнення зайвих операцій.

Використання оптимальних структур даних є важливим для забезпечення ефективного доступу та модифікації даних. Вибір відповідних структур даних допомагає підвищити продуктивність алгоритмів та зменшити витрати на їх виконання.

Розробка алгоритмів, які можуть ефективно працювати як з невеликими, так і з великими обсягами вхідних даних, є важливим аспектом для забезпечення їхньої універсальності та масштабованості.

Оцінка витрат часу та ресурсів на оптимізацію алгоритмів порівняно з отриманими вигодами є важливим для прийняття обґрунтованих рішень щодо впровадження оптимізації. Це включає аналіз компромісів між продуктивністю, використанням ресурсів та складністю реалізації.

Зменшення часової складності алгоритмів є важливим завданням для досягнення швидшого виконання та підвищення продуктивності гри. Це включає аналіз і оптимізацію алгоритмів для скорочення часу виконання, що є особливо важливим у реальних ігрових умовах, де затримки можуть негативно вплинути на геймплей.

Мінімізація просторової складності алгоритмів допомагає зменшити вимоги до пам'яті, що є критичним фактором, особливо для мобільних пристроїв та інших обмежених середовищ. Оптимізація використання пам'яті включає вибір ефективних структур даних і методів управління пам'яттю.

Ці аспекти є надзвичайно важливими при розробці програмного забезпечення, особливо в ігровій індустрії, де продуктивність та плавність геймплею є ключовими факторами успіху. Оптимізація алгоритмів ШІ дозволяє створювати більш реалістичні, ефективні та захоплюючі ігрові враження для користувачів.

1.2 Постановка задачі

Основною метою роботи є проведення дослідження щодо методів створення ігрового штучного інтелекту для різних сценаріїв гри на платформі Unity" виглядає так:

- проаналізувати поточний стан ігрового ШІ в індустрії та визначити основні виклики та тенденції;
- вивчити основні можливості та обмеження платформи Unity для реалізації ігрового ШІ [1];

- визначити різні типи ігрових сценаріїв та розглянути їхні особливості;
- визначити критерії, за якими можна порівнювати алгоритми, використані для розробки ігрового ШІ, між собою;
- проаналізувати різні методи створення ігрового ШІ;
- підсумувати результати дослідження, надати висновки.

Ця постановка задачі надає основні напрямки дослідження ігрового ШІ на Unity та визначає ключові аспекти, які будуть розглянуті в роботі.

2 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

2.1 Про ігрову індустрію

Комп'ютерні ігри — це форма розваги, яка виникла разом із розвитком комп'ютерних технологій. Вони включають в себе широкий спектр жанрів та концепцій, забезпечуючи гравцям можливість зануритися у віртуальний світ з різноманітними завданнями та викликами [2].

Основні компоненти комп'ютерних ігор включають графіку, звук, ігровий процес та, все більше, інтелектуальний рівень штучного інтелекту для створення живих та адаптивних персонажів і сценаріїв. Розвиток комп'ютерної графіки, обчислювальних можливостей та програмного забезпечення сприяє постійному удосконаленню геймплею та віртуального світу.

Ігри поділяються на різні жанри, такі як екшн, пригодницькі, стратегії, гонки, рольові ігри, головоломки та інші. Кожен жанр надає унікальний досвід гри та розробляє різні аспекти ігрового досвіду. Наприклад, екшн-ігри зазвичай акцентують увагу на швидкому темпі та реакції гравця, тоді як стратегії вимагають глибокого аналізу та планування.

Однією з ключових характеристик комп'ютерних ігор є їхній вплив на культуру та соціальний вимір. Вони стали не лише формою розваги, а й важливим засобом взаємодії та спілкування між людьми. Онлайн-ігрові спільноти створюють нові способи комунікації та взаємодії, здатні об'єднати гравців з різних куточків світу. Завдяки цим спільнотам, гравці можуть ділитися досвідом, співпрацювати або змагатися, що сприяє формуванню нових соціальних зв'язків та ком'юніті.

Комп'ютерні ігри також використовуються в освіті та навчанні, де вони можуть створювати інтерактивні та змістовні віртуальні середовища для покращення різних навичок та здібностей. Ігри можуть стимулювати розвиток критичного мислення, координації рухів, командної роботи та інших важливих навичок. Освітні ігри можуть включати симуляції історичних подій, наукових експериментів або навіть повсякденних життєвих ситуацій, що робить навчання більш захоплюючим і ефективним.

Зростання популярності комп'ютерних ігор також впливає на економіку. Ігрова індустрія стала багатомільярдним бізнесом, який залучає величезні інвестиції та створює безліч робочих місць у всьому світі. Розробка ігор вимагає участі фахівців різних галузей: програмістів, дизайнерів, художників, музикантів, сценаристів та інших професіоналів.

Сучасні технології, такі як віртуальна реальність (VR) та доповнена реальність (AR), відкривають нові горизонти для ігрової індустрії. Вони дозволяють створювати ще більш занурюючи та реалістичні ігрові світи, що підвищує інтерес гравців і розширює можливості для творчості розробників.

Водночас зростає потреба у створенні ефективного та реалістичного штучного інтелекту (ШІ) для ігрових персонажів. ШІ має забезпечувати реалістичну поведінку агентів, які здатні адаптуватися до дій гравця та змінюваних умов ігрового середовища. Це вимагає використання передових алгоритмів та методів, які можуть забезпечити високий рівень складності та інтерактивності гри.

Ігрова індустрія продовжує розвиватися та змінюватися, відповідаючи на виклики часу та задовольняючи зростаючі потреби гравців. Це динамічне поле, яке постійно еволюціонує, надаючи нові можливості для розробників та гравців у всьому світі.

2.2 Розвиток ігрового штучного інтелекту у історії ігрової індустрії

Історія ігрової індустрії свідчить про неперервний розвиток інновацій у вдосконаленні ігрового штучного інтелекту, який зробив значний шлях від простих алгоритмів до складних систем, здатних адаптуватися до виборів гравців та реагувати на зміни умови гри.

На ранніх етапах розвитку ігор, ігровий ШІ був обмежений базовими правилами системи, де програмувальний код визначав стратегії поведінки персонажів. Проте із зростанням обчислювальних можливостей та розвитком комп'ютерних технологій, з'явилися більш складні методи, такі як алгоритми машинного навчання та глибокого навчання.

Ще в 1951 році, використовуючи комп'ютер Ferranti Mark 1 в університеті міста Манчестер, вчений-програміст написав програму для змагання у шашки, а пізніше Дітріх Принц написав програму, що дозволяє грати у шахи з комп'ютером безкоштовно. У середині 50-х і на початку 60-х, зрештою, навичка вчених у цій галузі стабільно наростав, і поступово штучний інтелект досяг досить високого рівня. Ну, а справжнім апогеєм став 1997 рік, коли шаховий поєдинок між Гарі Каспаровим і комп'ютером завершився не на користь найбільшого гросмейстера.

Початкові етапи розвитку ШІ можна відслідковувати в аркадних іграх 70-80-х років, де використовувались прості алгоритми для керування рухом об'єктів та визначення стратегій ворогів (див. рис 2.1). Проте із введенням комп'ютерних консолей та особистих комп'ютерів, виникла необхідність у більш інтелектуальних ігрових персонажах.

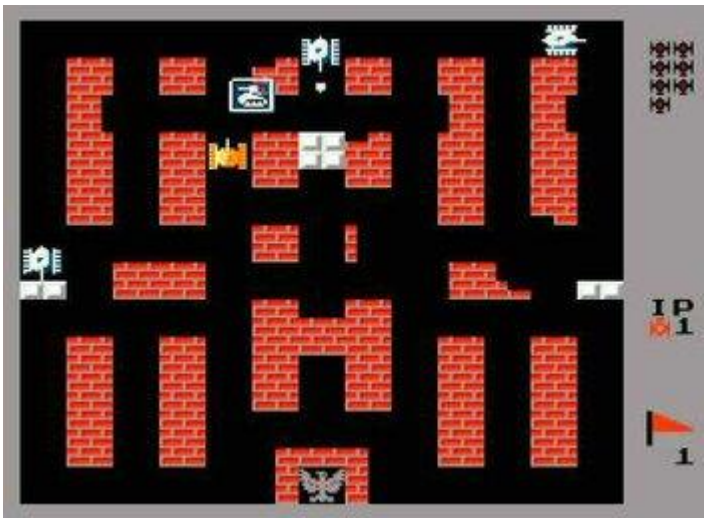


Рисунок 2.1 – Приклад аркадної гри Battle Sity (за даними [3])

Першими помітними прикладами стали аркадне полювання на качок в 1974 (див. рис. 2.2) і симулятор повітряного бою на винищувачі. Дві текстові комп'ютерні ігри, випущені в 1972 році, також мали штучних ворогів, рух яких був заснований на шаблонів, що збереглися. І лише винахід мікропроцесорів дозволило програмістом здійснювати більше обчислень та додавати випадкові елементи та рух у програмні коди.

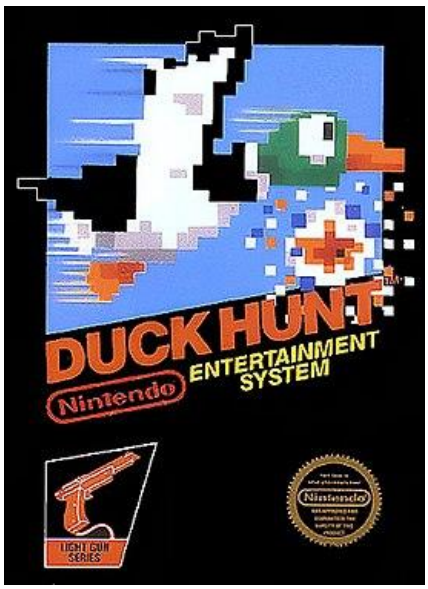


Рисунок 2.2 – Приклад аркадної гри Duck Hunt (за даними [4])

В 90-х роках в іграх з'явились системи прийняття рішень на основі дерева рішень та методи штучного інтелекту, що дозволяли персонажам адаптуватися до дій гравців та пристосовуватися до обставин у грі, що змінюються. Ігри, такі як "F.E.A.R." та "Black & White" показали нові можливості в реалізації ШІ (див. рис 2.3, 2.4).

Рисунок 2.3 – Приклад гри F.E.A.R. (за даними [5])

Рисунок 2.4 – Приклад гри Black & White (за даними [6])

З переходом у нове тисячоліття, розвиток машинного навчання та глибокого навчання значно вплинув на ігровий ШІ. Глибокі нейронні мережі дозволили створювати більш складні та гнучкі системи ШІ, здатні до вивчення та самостійного вдосконалення в ході гри.

Сучасний етап розвитку ігрового ШІ відзначається використанням технологій штучного інтелекту для створення індивідуальних та надзвичайно реалістичних ігрових персонажів, які реагують на гравців та динаміку гри з вражаючою аутентичністю. Розвиток ігрового ШІ і надалі визначатиме напрямок ігрової індустрії, даруючи гравцям неперевершений рівень взаємодії та емоційної насолоди.

2.3 Алгоритми прийняття рішень агентом

Штучний інтелект – це розділ комп’ютерних наук, що зосереджений на створенні пристроїв, що могли б виконувати завдання, які зазвичай потребують людського втручання.

Ігровий штучний інтелект має різноманітні застосування у відеоіграх і може бути реалізований в різних формах.

У процесі розробки ігрового штучного інтелекту використовуються різні алгоритми прийняття рішень, кожен з яких має свої особливості та застосування в залежності від ігрового сценарію. Далі наведено деякі з найбільш поширених алгоритмів прийняття рішень, що використовуються в ігровій сфері:

Дерева рішень – є структурованими діаграмами, що представляють послідовність рішень та їх можливі наслідки. Вони складаються з вузлів, що представляють рішення, та гілок, що представляють можливі результати кожного рішення.

Переваги: прості у візуалізації та реалізації, добре підходять для сценаріїв із визначеними правилами.

Недоліки: можуть стати надто складними та важкими для підтримки в разі великої кількості можливих рішень та результатів.

Машини станів моделюють поведінку агентів через набір станів та переходів між ними на основі певних умов. Кожен стан визначає поведінку агента у відповідний момент часу.

Переваги: легкі для розуміння та реалізації, добре підходять для простих сценаріїв та ігрових персонажів з обмеженим набором поведінкових шаблонів.

Недоліки: мають обмежену гнучкість і можуть бути складними для масштабування у складних сценаріях.

Поведінкові дерева – ці дерева використовуються для моделювання комплексної поведінки агентів через ієрархічну структуру, де вузли представляють поведінку або дії, а гілки визначають умови для переходу між ними.

Переваги: гнучкі та модульні, добре підходять для створення складних та багаторівневих поведінкових моделей.

Недоліки: можуть бути складними у реалізації та налаштуванні, потребують ретельного планування та тестування.

Прийняття рішень на основі часу. Цей метод використовує таймери для контролю частоти виконання певних дій агентами. Наприклад, агенти можуть перевіряти стан навколишнього середовища або виконувати дії через певні інтервали часу.

Переваги: легкі у реалізації та налаштуванні, добре підходять для динамічних сценаріїв з регулярними перевірками стану.

Недоліки: обмежена гнучкість, можуть бути неефективними для сценаріїв, де потрібна миттєва реакція на події.

Нейронні мережі та навчання з підкріпленням використовуються для створення адаптивних моделей прийняття рішень, здатних навчатися на основі досвіду. Метод навчання з підкріпленням дозволяє агентам самостійно вдосконалювати свою поведінку через взаємодію з ігровим середовищем.

Переваги: висока гнучкість та здатність до самонавчання, можливість створення складних та реалістичних моделей поведінки.

Недоліки: вимагають значних обчислювальних ресурсів, складні у реалізації та налаштуванні, потребують великої кількості даних для навчання.

Динамічне планування використовує метод розбиття задачі на підзадачі з метою збереження результатів для уникнення повторних обчислень. Це дозволяє ефективно вирішувати задачі оптимізації, де необхідно знайти найкращий шлях або стратегію.

Переваги: ефективне для задач, які можна розбити на підзадачі з перекриттям, добре підходить для задач оптимізації.

Недоліки: може бути складним у реалізації для великих задач через значні вимоги до пам'яті.

Марковські процеси прийняття рішень є математичною моделлю для прийняття рішень в умовах невизначеності. Вона враховує ймовірності переходу між станами та винагороди, асоційовані з цими переходами.

Переваги: може враховувати ймовірнісні аспекти прийняття рішень, добре підходить для задач, де є невизначеність.

Недоліки: вимагає значних обчислювальних ресурсів для вирішення великих задач, складний у реалізації.

Алгоритм випадкового вибору використовує генерацію випадкових чисел для вибору дій або стратегій. Це дозволяє створити непередбачувану поведінку агентів.

Переваги: простий у реалізації, забезпечує варіативність поведінки, що може підвищити цікавість гри.

Недоліки: може призводити до непослідовної або нерозумної поведінки агентів, що може знизити якість геймплею.

Генетичні алгоритми використовують принципи природного відбору та генетики для пошуку оптимальних рішень. Вони працюють з популяцією можливих рішень, які еволюціонують через процеси відбору, кросовера та мутації.

Переваги: можуть знаходити оптимальні або наближені до оптимальних рішення в складних пошукових просторах, гнучкі та адаптивні.

Недоліки: можуть вимагати значного часу на обчислення, особливо для великих популяцій та складних задач.

Мінімакс є алгоритмом прийняття рішень, який використовується для ігор з нульовою сумою, таких як шахи та шашки. Він визначає найкращий хід, враховуючи всі можливі відповіді супротивника, та намагається мінімізувати максимальний виграш супротивника.

Переваги: гарантує оптимальне рішення в умовах, коли супротивник грає найкраще.

Недоліки: може бути надто обчислювально затратним для ігор з великою кількістю можливих ходів.

Прийняття рішень на основі реального часу. Цей метод забезпечує прийняття рішень у реальному часі, враховуючи постійно змінювані умови та швидкі реакції на події у грі.

Переваги: забезпечує швидкі та адаптивні реакції, добре підходить для екшн-ігор та ігор у реальному часі.

Недоліки: вимагає високої оптимізації та ефективності, може бути складним у реалізації.

2.4 Алгоритми пошуку шляху

Алгоритми пошуку шляху є важливим компонентом у розробці штучного інтелекту для агентів у комп'ютерних іграх. Вони дозволяють агентам знаходити оптимальний шлях від однієї точки до іншої. Алгоритми пошуку шляху є критичними для забезпечення реалістичної та ефективної поведінки агентів у іграх. Вони дозволяють агентам орієнтуватися у складних середовищах, уникати перешкод та досягати цілей, що підвищує загальний рівень залученості та задоволення гравців.

Існує декілька поширених алгоритмів пошуку шляху, які використовуються для забезпечення ефективної та реалістичної навігації агентів.

A* є одним з найпопулярніших алгоритмів пошуку шляху. Він використовує евристичну функцію для оцінки вартості шляху від початкової точки до цільової, дозволяючи швидко знаходити оптимальний маршрут.

Переваги: забезпечує ефективний та оптимальний пошук шляхів, добре підходить для складних середовищ.

Недоліки: може вимагати значних обчислювальних ресурсів для великих карт.

Алгоритм Дейкстри знаходить найкоротший шлях від початкової точки до всіх інших точок графа. Він використовує пріоритетну чергу для поступового розширення найкоротшого шляху.

Переваги: гарантує знаходження найкоротшого шляху для всіх можливих цілей.

Недоліки: менш ефективний у порівнянні з A* у великих середовищах, де відомо місцезнаходження цілі.

Breadth-First Search є алгоритмом пошуку в ширину, який досліджує всі можливі шляхи рівня за рівнем, поки не знайде цільову точку.

Переваги: простий у реалізації, гарантує знаходження найкоротшого шляху в неважених графах.

Недоліки: може бути неефективним для великих графів через високе споживання пам'яті.

Depth-First Search є алгоритмом пошуку в глибину, який досліджує якомога глибше один шлях перед переходом до наступного.

Переваги: простий у реалізації, використовує менше пам'яті порівняно з BFS.

Недоліки: може не знайти оптимальний шлях у великих графах, схильний до зациклення.

Theta* є модифікацією алгоритму A*, який дозволяє прямі переходи між вузлами, а не лише по сітці. Це робить маршрути більш природними та коротшими.

Переваги: забезпечує більш природні та короткі шляхи, ніж традиційний A*.

Недоліки: складніший у реалізації, може вимагати більше обчислювальних ресурсів.

Jump Point Search є оптимізацією алгоритму A*, який зменшує кількість перевірених вузлів, пропускаючи надмірні перевірки.

Переваги: значно підвищує ефективність пошуку шляху на великих картах з сіткою.

Недоліки: працює лише на сітках з регулярними вузлами, складніший у реалізації.

Примітивні алгоритми – ці алгоритми використовують прості геометричні примітиви, такі як пряма лінія, для пошуку шляхів між двома точками в просторі.

Переваги: легкі для розуміння та швидкі у виконанні для простих середовищ.

Недоліки: не завжди знаходять оптимальні шляхи, обмежені у складних середовищах.

Greedy Best-First Search – це алгоритм жадібного пошуку використовує евристику для вибору шляху, який здається найбільш перспективним на кожному кроці.

Переваги: швидкий у пошуку шляху, особливо у великих середовищах.

Недоліки: не гарантує знаходження оптимального шляху.

Bidirectional Search починає пошук одночасно з початкової точки і з цільової точки, доки обидва пошуки не зустрінуться.

Переваги: зменшує кількість необхідних обчислень, що робить його швидшим для великих графів.

Недоліки: складний у реалізації, потребує додаткового зберігання даних для обох пошуків.

Дивергентні шляхи – цей метод дозволяє агентам вибирати різні шляхи до цілі, щоб уникнути скупчення та блокування на одному шляху.

Переваги: покращує продуктивність у багатокористувацьких середовищах, запобігає заторам.

Недоліки: складний у реалізації, може вимагати більше обчислювальних ресурсів для координації агентів.

Ант-Колонія використовує поведінку мурах для знаходження найкоротшого шляху. Віртуальні мурахи досліджують середовище, залишаючи "феромони" на найкращих маршрутах, які інші мурахи можуть слідувати.

Переваги: підходить для розв'язання задач оптимізації в динамічних середовищах.

Недоліки: може бути повільним у випадках великої кількості агентів та складних середовищ.

Lee Algorithm або алгоритм заливання, використовує метод заливання для пошуку найкоротшого шляху в лабіринті.

Переваги: простий у реалізації, гарантує знаходження найкоротшого шляху.

Недоліки: вимагає значних обчислювальних ресурсів для великих середовищ, неефективний у великих графах.

Dynamic Programming Maze використовує метод збереження результатів підзадач для уникнення повторних обчислень.

Переваги: ефективний для розв'язання задач, які можна розбити на підзадачі.

Недоліки: складний у реалізації, вимагає значних обчислювальних ресурсів.

2.5 Ігрові сценарії

Ігрові сценарії можуть варіюватися від простих історій до складних та глибоких сюжетів, залежно від жанру гри та її цільової аудиторії. Нижче наведено декілька прикладів ігрових сценаріїв та їх короткий опис:

Екшн – гравець виконує завдання, що включає інтенсивну стрілянину та боротьбу з ворогами. Швидкі реакції та стратегічне мислення є ключовими для успіху. У грі "Call of Duty" гравець бере участь у військових операціях, використовуючи різноманітну зброю та тактику для перемоги над супротивниками.

Рольова гра – гравець вирушає в подорож, щоб врятувати світ, розв'язує завдання та взаємодіє з персонажами, розвиваючи свого героя та його навички. У грі "The Witcher 3" гравець бере на себе роль мисливця на монстрів, досліджуючи великий світ, взаємодіючи з NPC та виконуючи різноманітні квести.

Стратегія – гравець керує містом або цивілізацією, будує структури, розвиває економіку та планує військові кампанії для досягнення домінування. грі "Civilization VI" гравець керує цивілізацією через століття, приймаючи політичні, військові та економічні рішення для процвітання свого народу.

Пригодницька гра – гравець розслідує та вирішує загадкові події, розкриваючи таємниці та проходячи через різні випробування. У грі "Uncharted" гравець керує шукачем скарбів, досліджуючи древні руїни, розгадуючи головоломки та борючись із супротивниками.

Жахи – гравець повинен виживати в жахливому середовищі, уникати монстрів та розв'язувати головоломки для прогресу. У грі "Resident Evil" гравець

бореться з зомбі та іншими жахливими створіннями, досліджуючи темні локації та шукаючи шляхи для виживання.

Інтерактивний фільм – гравець приймає рішення, які впливають на хід історії та кінцівку, забезпечуючи унікальний досвід кожного проходження. У грі "Detroit: Become Human" гравець керує кількома персонажами, приймаючи важливі рішення, що визначають їхні долі та розвиток сюжету.

Ігри-головоломки – гравець розв'язує головоломки або намагається вибратися з складної лабіринтової структури, використовуючи логіку та кмітливість. У грі "Portal" гравець використовує портал-гармату для розв'язання фізичних головоломок і просування через рівні.

Симулятор – гравець керує персонажем або сім'єю, розвиваючи їхнє життя, забезпечуючи їхні потреби та досягаючи особистих цілей. У грі "The Sims" гравець створює та контролює життя віртуальних персонажів, керуючи їхніми кар'єрами, стосунками та повсякденним життям.

Спортивна гра – гравець бере участь у спортивних змаганнях, намагаючись стати чемпіоном у вибраному виді спорту. У грі "FIFA" гравець керує футбольною командою, змагаючись у турнірах та матчах для досягнення перемог.

Аркада – гравець намагається вижити якнайдовше в безкінечних хвилях ворогів, змагаючись за високі бали та досягнення. У грі "Pac-Man" гравець керує персонажем, який їсть точки в лабіринті, уникаючи привидів та збираючи бонуси.

Гонки – гравець бере участь у гонках, перемагаючи суперників, виконуючи трюки та покращуючи свій транспортний засіб. У грі "Need for Speed" гравець змагається у високошвидкісних перегонах, уникаючи поліції та суперників.

Економічний симулятор – гравець керує власним бізнесом, розвиває його, забезпечує прибуток та приймає стратегічні рішення для процвітання. У грі "SimCity" гравець керує містом, будуючи інфраструктуру, управляючи фінансами та забезпечуючи добробут мешканців.

Фантастичні пригоди – гравець досліджує фантастичний світ, стикається з магією, незвичайними істотами та дивовижними явищами. У грі "The Legend of

Zelda" гравець подорожує через фантастичний світ, бореться з ворогами та розгадує головоломки.

Виживання – гравець повинен вижити в суворих умовах, збираючи ресурси, будуючи укриття та відбиваючись від диких тварин чи ворогів. У грі "Minecraft" гравець добуває ресурси, створює предмети та будівлі, щоб вижити в постійно змінюваному світі.

Платформери – гравець керує персонажем, що стрибає через різноманітні перешкоди та рівні, збираючи предмети та досягаючи цілей. У грі "Super Mario Bros." гравець керує Маріо, який подорожує через різні рівні, збираючи монети та рятуючи принцесу.

Постапокаліптичний сценарій – гравець виживає у світі після глобальної катастрофи, бореться за ресурси та стикається з іншими вижившими або мутантами. У грі "Fallout" гравець досліджує постапокаліптичний світ, виконує квести та взаємодіє з іншими персонажами.

Шпигунські ігри – гравець виконує місії під прикриттям, використовуючи стелс та високотехнологічні пристрої для досягнення своїх цілей. У грі "Metal Gear Solid" гравець керує шпигуном, який використовує стелс та бойові навички для виконання завдань.

Історичні ігри – гравець бере участь у подіях, заснованих на реальних історичних подіях, взаємодіє з історичними персонажами та приймає ключові рішення. У грі "Assassin's Creed" гравець досліджує історичні епохи, виконує завдання та взаємодіє з відомими історичними постатями.

Це лише декілька прикладів, бо кількість ігрових сценаріїв просто безмежна і постійно доповнюється. Різноманіття жанрів та сюжетів дозволяє розробникам створювати унікальні та захоплюючі ігри, які можуть задовольнити будь-які смаки та інтереси гравців.

2.6 Ігрові сценарії, де бере учать ігровий штучний інтелект

Залежно від гри, ігровий штучний інтелект може виконувати різноманітні функції: допомагати гравцю чи навпаки воювати проти нього, супроводжувати і

давати підказки або стати вірним побратимом. Нижче наведено декілька прикладів ігрових ситуацій, де використовується ігровий ШІ.

У іграх бойовиках ШІ може керувати ворожими силами. Його завданням є створення реалістичного та викликаючого опонента, який може адаптувати свою стратегію в залежності від дій гравця. ШІ може керувати ігровими персонажами, які використовують стрільбу, тактику укриття та маневри. Важливо, щоб ШІ адекватно реагував на дії гравця та створював високий рівень виклику.

Наприклад, у грі "Call of Duty" ШІ контролює ворогів, які взаємодіють з гравцем у режимі реального часу, змінюючи свої стратегії та укриття залежно від ситуації.

У кооперативних іграх ШІ може допомагати гравцеві виконувати завдання, подолати перешкоди та вести себе як ефективний член команди. Гравець може створювати власних ігрових агентів, визначаючи їх характеристики, стратегії та реакції на різні ситуації.

Наприклад, у грі "Left 4 Dead" ШІ керує союзниками гравця, які допомагають боротися з зомбі та виконувати завдання, координуючи свої дії з гравцем.

У діалогових іграх ШІ може розпізнавати мову гравця та відповідати на його запитання або команди, що дозволяє зробити більш глибоку інтеграцію з грою. ШІ може аналізувати та розуміти текстові запитання гравця, намагаючись вивести логічно правильні відповіді.

Наприклад, у грі "Mass Effect" діалогова система з ШІ дозволяє гравцям взаємодіяти з персонажами гри, приймати рішення, що впливають на розвиток сюжету.

Ігровий ШІ може адаптувати рівень складності гри в реальному часі в залежності від навичок та досвіду гравця. Це допомагає зберігати баланс гри та забезпечувати гравцям оптимальний рівень виклику.

Наприклад, у грі "Resident Evil 4" ШІ динамічно змінює складність ворогів та ресурси залежно від успіхів гравця, щоб забезпечити збалансований геймплей.

ШІ може генерувати різні ігрові події, завдання чи конфлікти в грі для забезпечення динаміки та цікавості гравців. Це може включати випадкові зустрічі з ворогами, появу нових квестів або зміну умов гри.

Наприклад, у грі "The Elder Scrolls V: Skyrim" ШІ генерує випадкові події та квести, що робить світ гри живим та непередбачуваним.

У симуляційних іграх ШІ може контролювати поведінку NPC, імітуючи реальні соціальні, економічні або екологічні системи. ШІ може моделювати взаємодії між персонажами, їхній розвиток та реакції на дії гравця.

Наприклад, у грі "The Sims" ШІ контролює дії та взаємодії персонажів, моделюючи їхні потреби, бажання та соціальні взаємини.

У стратегічних іграх ШІ може управляти ресурсами, будувати структури, тренувати війська та планувати тактичні операції. ШІ має здатність адаптувати свої стратегії залежно від дій гравця та змінювати тактику в реальному часі.

Наприклад, у грі "StarCraft II" ШІ контролює протиборчі фракції, плануючи атаки, захист та управління ресурсами, створюючи складний і викликаючий геймплей.

У гоночних іграх ШІ може керувати транспортними засобами, адаптуючи їхню швидкість, траєкторії та тактику для створення конкурентного середовища. ШІ також може реагувати на зміни траси та дії гравця.

Наприклад, у грі "Mario Kart" ШІ контролює суперників, які змагаються з гравцем, використовуючи різні тактики та стратегії для перемоги.

У іграх в жанрі Tower Defense ШІ може аналізувати стратегії гравця і відповідно змінювати поведінку ворогів [7]. Це може включати збільшення швидкості, зміцнення броні або використання спеціальних атак, щоб забезпечити додатковий виклик.

Наприклад, у грі "Bloons TD" вороги можуть адаптуватися до типів веж, які використовує гравець, наприклад, з'являтися у великих групах або використовувати спеціальні здібності для прориву оборони.

У іграх з відкритим світом ШІ може контролювати динамічні події, реагувати на дії гравця та забезпечувати живу екосистему. ШІ може моделювати природні явища, поведінку тварин та соціальні взаємодії NPC.

Наприклад, у грі "Red Dead Redemption 2" ШІ моделює життя дикого заходу, контролюючи взаємодії персонажів, тварин та навколишнього середовища.

Ці приклади ілюструють, як різноманітний ігровий штучний інтелект може бути адаптований для різних жанрів та сценаріїв, забезпечуючи захоплюючий та реалістичний досвід для гравців.

3 ІГРОВИЙ РУШІЙ

3.1 Обрання ігрового рушія

Далі наведено чому саме був обраний ігровий рушій Unity3D.

Наразі на ринку існує декілька конкурентних ігрових рушіїв, це: Unity3D Engine, Unreal Engine, Godot Engine.

Далі наведено короткий опис цих технологій:

Unity3D – це ігровий рушій та інтегроване середовище розробки для створення 2D та 3D ігор, віртуальної реальності (VR) та інших інтерактивних додатків. Використовує такі мови програмування: C# та JavaScript. Має безкоштовну версію та платні плани для більш розширених можливостей та комерційного використання. Простий для вивчення та початку роботи. Має велику спільноту розробників та широку підтримку сторонніх активів. Зручна інтеграція з різними платформами (мобільні пристрої, консолі, VR). Гнучка система розширення та активів.

Unreal Engine – це ігровий рушій та середовище розробки для створення великих та високоякісних ігор. Використовує такі мови програмування: C++ та Blueprints (візуальна система програмування). Безкоштовний для особистих проектів та комерційних проектів (з відрахуваннями від прибутку). Велика потужність та графічна деталізація. Розширені засоби програмування з використанням C++. Висока якість графіки та реалістичність. Підтримка VR, AR та інших ігрових пристроїв.

Godot Engine – відкритий ігровий рушій з візуальною системою програмування. Використовує GDScript (схожий на Python) та C#. Повністю безкоштовний. Простий у вивченні, особливо для новачків. Ефективний для 2D та 3D графіки, має низькі системні вимоги. Підтримує різні платформи, але може вимагати додаткових налаштувань для мобільних пристроїв.

Далі позначаємо критерії, за якими будемо порівнювати рушії:

– розширеність функцій та гнучкість – міра розмаїття та складності інструментів, які ігровий рушій може надати для розробників гри;

- легкість використання – простота та зручність використання інтерфейсу та інструментів для розробки гри;
- спільнота та документація – наявність активної спільноти розробників та якість документації для вирішення труднощів;
- інтеграція з іншими ігровими компонентами – легкість інтеграції ігрового рушія з різними компонентами, такими як фізика, звук, графіка тощо;
- швидкість та оптимізація – ефективність рушія у відтворенні гри з високою швидкістю кадрів та здатність оптимізувати ресурси;
- розповсюдження та ліцензування – умови використання рушія та можливість використання його в комерційних проектах.

Тепер обравши критерії вибору, додаймо їх до таблиці критеріїв відносно кожного рушія (див. табл. 3.1).

Таблиця 3.1 – Значення критеріїв вибору ігрового рушія (таблиця виконана самостійно)

	Розширеність функцій та гнучкість	Легкість використання	Спільнота та документація	Інтеграція з іншими ігровими компонентами	Швидкість та оптимізація	Розповсюдження та ліцензування
Unity 3D Engine	5 - Unity має багатофункціональну систему та є дуже гнучким для розробки різноманітних ігор.	5 - Unity вважається одним з найлегших для вивчення та початку роботи ігрових рушіїв.	5 - Unity має велику та активну спільноту розробників та широку базу документації.	5 - Unity підтримує широкий спектр інтеграції з ігровими компонентами, включаючи різні бібліотеки та SDK.	5 - Unity може вимагати деяких налаштувань для досягнення високої оптимізації у великих проектах.	4 - Є безкоштовна версія, але для деяких функцій та комерційного використання може бути необхідна платна ліцензія.

Кінець таблиці 3.1

	Розширеність функцій та гнучкість	Легкість використання	Спільнота та документація	Інтеграція з іншими ігровими компонентами	Швидкість та оптимізація	Розповсюдження та ліцензування
Unreal Engine	5 - Unreal має велику кількість функцій та є дуже гнучким для розробки різноманітних ігор.	3 - Може бути складнішим для вивчення порівняно з Unity, але має потужні можливості для досвідчених розробників.	4 - Хоча менша за Unity, спільнота Unreal все ще велика, а документація є досить повною.	5 - Unreal гарно інтегрується з різними компонентами, включаючи високоякісну підтримку VR та AR.	5 - Unreal вражає швидкістю та високоякісною оптимізацією, особливо для графічно вимогливих проектів.	4 - Безкоштовний для особистих та комерційних проектів, але з обов'язковими відрахуваннями у випадку успіху.
Godot Engine:	4 - Godot надає багато можливостей, але може бути менш гнучким для деяких великих проектів порівняно з конкурентами.	5 - Godot славиться своєю легкістю використання, особливо для новачків та тих, хто знайомий із скриптованою розробкою	3 - Хоча спільнота Godot росте, вона все ще менша за Unity та Unreal, але документація досить добре виписана.	4 - Godot підтримує різні ігрові компоненти, але інтеграція може вимагати деяких зусиль.	3 - Godot може мати певні обмеження в оптимізації для деяких проектів, особливо для графічно вимогливих.	5 - Godot є повністю безкоштовним та відкритим, що дає велику вільність у використанні та модифікації.

Наступний крок – будемо використовувати лінійну адитивну згортку з ваговими коефіцієнтами, тому що деякі з окремих критеріїв є більш важливі за інші. Тому надамо кожному критерію свій коефіцієнт значення для підрахунків.

– розширеність функцій та гнучкість – 0,15;

– легкість використання – 0,05;

– спільнота та документація – 0,1;

– інтеграція з іншими ігровими компонентами – 0,2;

– швидкість та оптимізація – 0,45;

– розповсюдження та ліцензування – 0,05.

Найбільший коефіцієнт має критерій «Швидкість та оптимізація», адже від нього напряму залежить швидкодія нашого штучного інтелекту.

Описавши критерії та їх коефіцієнти, ми можемо знайти який з цих рушіїв нам підходить більше усього. Розрахунки проводимо за формулою 3.1:

$$Z^* = \max_{i=1,m} \sum_{j=1}^{n} \alpha_j \beta_j a_{ij} \qquad 3.1)$$

Де,

α_j – ваговий коефіцієнт для критерію j, що відображає його важливість.

β_j– нормалізоване значення вагового коефіцієнта для критерію j.

a_{ij} – оцінка ігрового рушія і за критерієм j.

Вагові коефіцієнти (α_j) відображають важливість кожного критерію. Наприклад, швидкість та оптимізація отримує найбільший коефіцієнт (0,45), оскільки вона є ключовим фактором для нашого проекту.

Нормалізовані вагові коефіцієнти (β_j) розраховуються для кожного критерію, щоб забезпечити їх відносну важливість. Вони можуть бути розраховані так, щоб сума всіх коефіцієнтів дорівнювала 1.

Оцінки ігрових рушіїв (a_{ij}) показують, наскільки добре кожен рушій відповідає кожному критерію.

Для зручності, проводимо розрахунки у Excel, а результати перенесемо сюди у вигляді таблиці 3.2.

Таблиця 3.2 – Результати розрахунку (таблиця виконана самостійно)

	Розширеність функцій та гнучкість	Легкість використання	Спільнота та документація	Інтеграція з іншими ігровими компонентами	Швидкість та оптимізація	Розповсюдження та ліцензування	Z*
Unity3D Engine	5	5	5	5	5	4	0,374
Unreal Engine	5	3	4	5	5	4	0,358
Godot Engine	4	5	3	4	3	5	0,267
β	0,15	0,05	0,1	0,2	0,45	0,05	
α	0,07142	0,07692	0,083333	0,0714286	0,0769230	0,076923	

Отримано такі результати:

Unity3D Engine – 0,374.

Unreal Engine – 0,35833.

Godot Engine – 0,26731.

Знайдено ігровий рушій, що підходить для проведення дослідження, а саме – Unity3D Engine, що має значення 0,374, що більше за інші.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Алгоритми пошуку шляху

4.1.1 TileGrid

Для розробки та тестування наших алгоритмів пошуку шляху необхідно розробити середовище, де вони могли б працювати. Для цього було вирішено розробити певний код, що відповідав би за генерацію графа, в нашому випадку – сітка певних розмірів.

Основний функціонал полягає в ініціалізації сітки тайлів. Клас TileGrid створює сітку з тайлів, визначаючи кількість рядків та стовпців.

Всі методи, де використовується генерація об’єктів чи визначення позицій за допомогою випадкових чисел, використовує значення seed, аби можна було повторити генерацію в будь-який момент.

Сітка представляється у вигляді двовимірного масиву тайлів, де кожен тайл має свою позицію у сітці. Для кожного тайла визначається його тип, наприклад, звичайний, дорогий або непрохідний. Візуальні представлення тайлів створюються з використанням префабів, визначених у властивості TilePrefab.

Випадковим чином визначаються стартова та кінцева позиції на сітці. Початкова позиція та кінцева позиція не можуть збігатися.

Визначається кількість перешкод на сітці на основі розмірів сітки та генератора випадкових чисел. Перешкоди розміщуються у випадкових позиціях на сітці, крім стартової та кінцевої позицій. Тайли, що містять перешкоди, стають непрохідними для агентів.

Визначається кількість дорогих тайлів на основі розмірів сітки та генератора випадкових чисел. Дорогі тайли мають підвищену вартість проходження і розміщуються у випадкових позиціях на сітці, крім стартової та кінцевої позицій.

Кожен тип тайла має свій унікальний колір для візуального відображення (звичайний, дорогий, непрохідний, стартовий, кінцевий, відвіданий, шлях).

Кольори тайлів змінюються в реальному часі в процесі роботи алгоритмів пошуку шляху.

Далі наведено приклад згенерованого графу з тайлів (див. рис. 4.1).

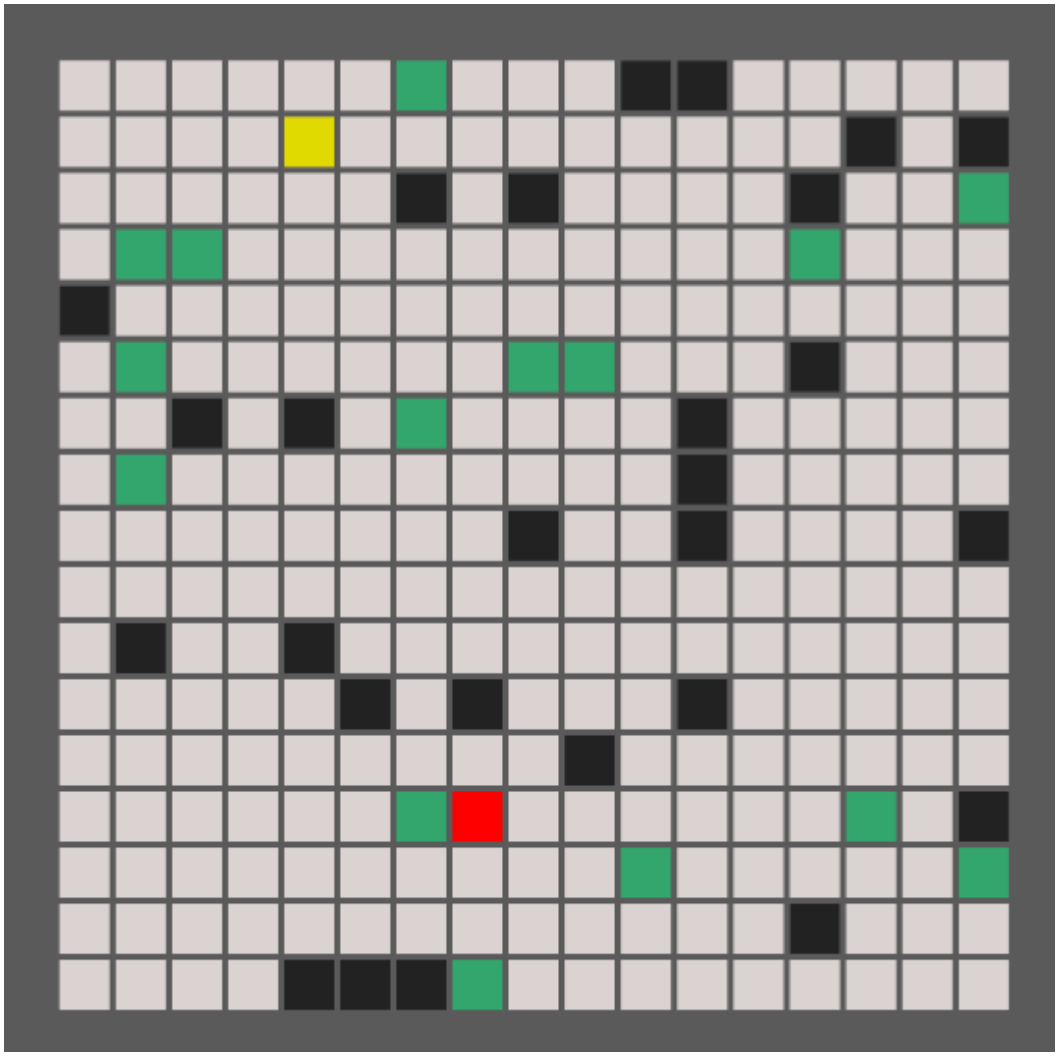


Рисунок 4.1 - Згенерований граф з тайлів (створено самостійно)

На рисунку зображено клітини різного кольору, далі наведено пояснення:

– сірі – це звичайні тайли, вартість проходу по яким коштує 1;
– зелені – це тайли, вартість проходу по яким коштує більше, наприклад 5;
– чорні – це тайли, прохід по яким неможливий;
– жовтий – це тайл стартова точка;
– червоній – це тайл кінцева точка;

Клас TileGrid інтегрує різні алгоритми пошуку шляху, такі як A*, Dijkstra, BFS, DFS, та інші.

Алгоритми приймають на вхід сітку тайлів, стартовий та кінцевий тайли, і повертають список тайлів, що представляє знайдений шлях. Потім, їх робота візуалізується покроково в реальному часі або миттєво, приклад показано но рисунку 4.2.

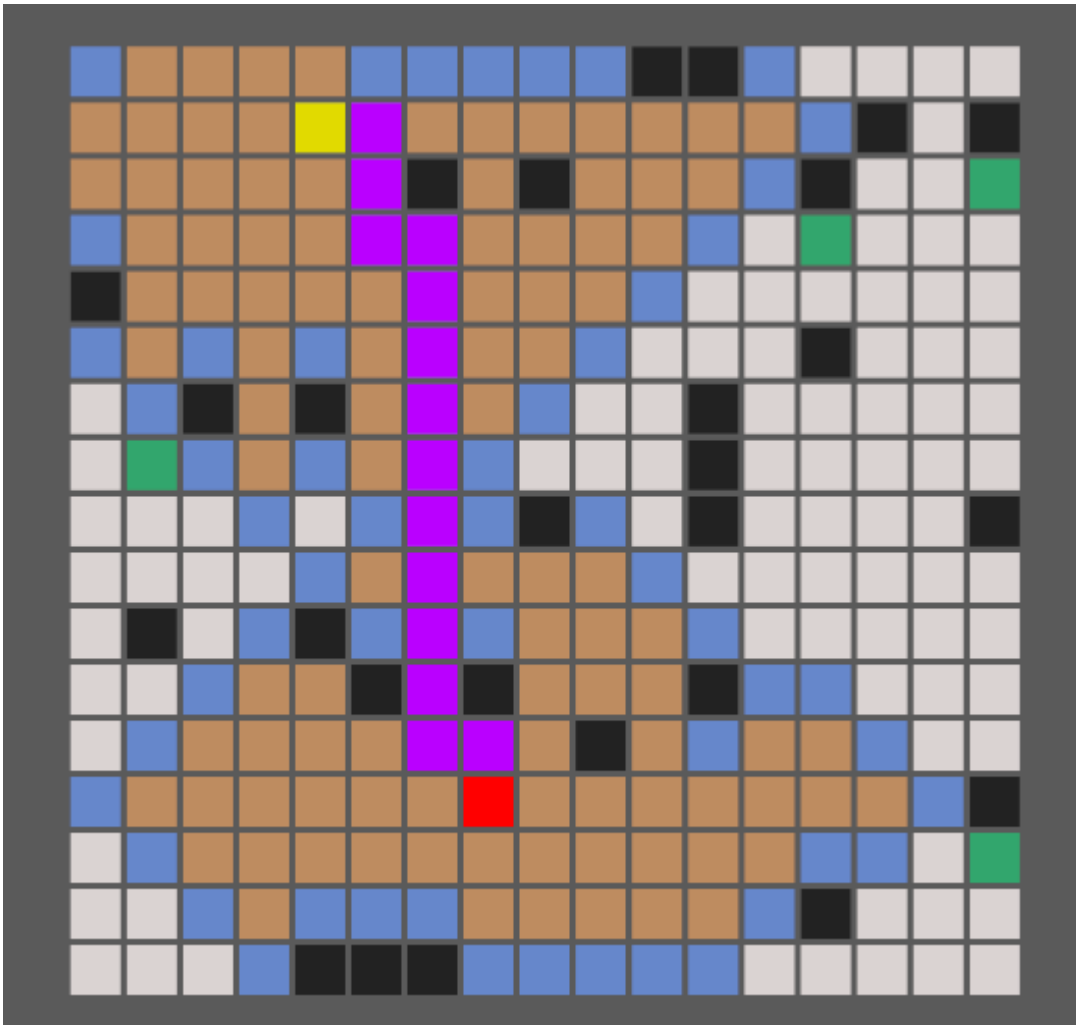


Рисунок 4.2 – Візуалізація робити алгоритму (створено самостійно)

На рисунку зображено клітини різного кольору, далі наведено пояснення:

– фіолетовий – це тайли, що позначають шлях від початкової точки до кінцевої, знайдений алгоритмом;
– коричневі – це переглянуті алгоритмом тайли;

– синій – цей тайл є частиною так званого "фронту пошуку", який представляє собою тайл, який вже був розглянутий алгоритмом, але ще не повністю оброблений;

Параметри, такі як час виконання, кількість відвіданих вузлів, довжина шляху та використання пам'яті, збираються для кожного алгоритму, виводяться на екран та зберігаються у файл.

Результати виконання алгоритмів пошуку шляху виводяться на екран за допомогою текстових полів, які відображають назву алгоритму, час виконання, кількість відвіданих вузлів, довжину шляху та використання пам'яті, це можна побачити на рисунку 4.3.

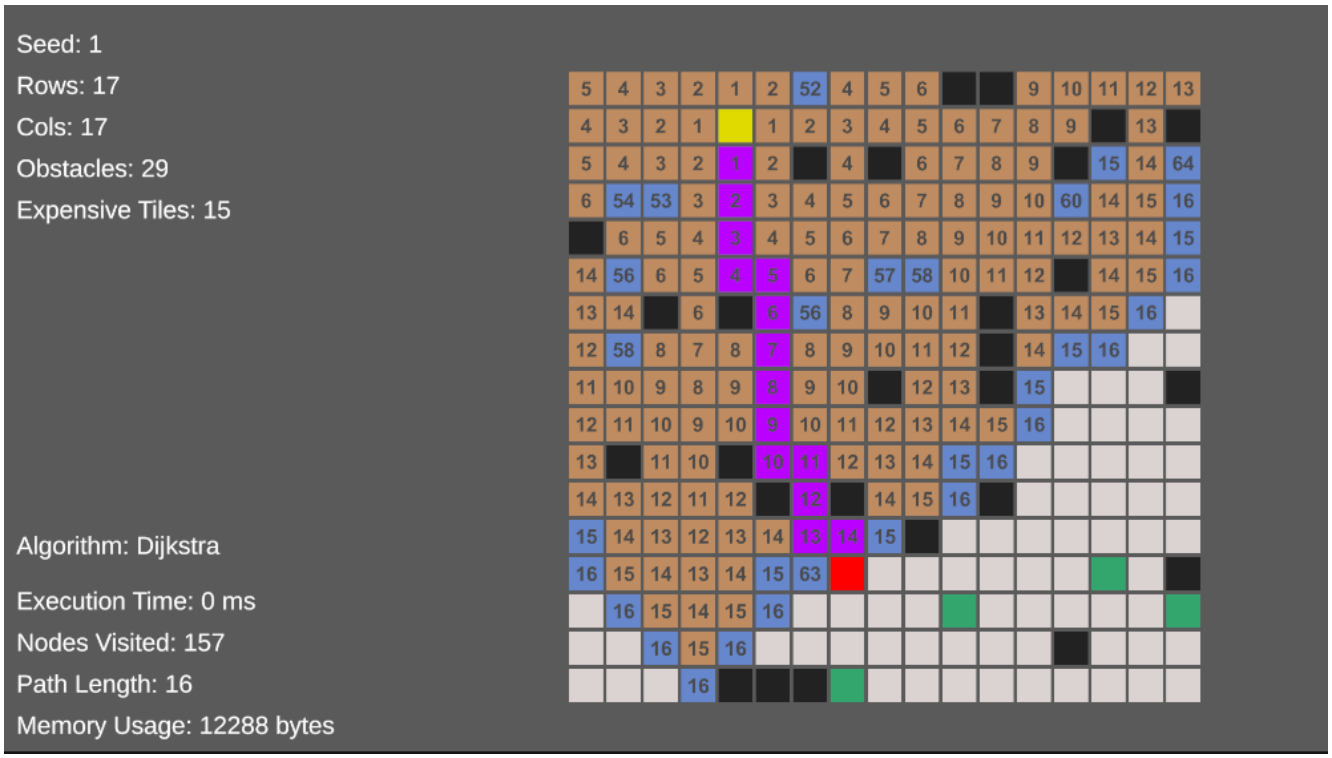


Рисунок 4.3 – Виведення даних на екран (створено самостійно)

Тут показано загальну інформацію про сітку, на якій працює алгоритм, у верхній частині, інформацію про роботу алгоритму у нижній частині та сам результат пошуку шляху справа.

Результати виконання алгоритмів пошуку шляху зберігаються у файл CSV для подальшого аналізу. Записуються такі параметри, як назва алгоритму, час

виконання, кількість відвіданих вузлів, довжина шляху, використання пам'яті, розміри сітки, значення seed.

Користувач може запускати різні алгоритми пошуку шляху за допомогою натискання певних клавіш. При натисканні пробілу запускається тестування всіх алгоритмів на різних розмірах сітки та значеннях seed. При натисканні клавіш від 1 до 8 запускаються відповідні алгоритми пошуку шляху, а при натисканні Escape сітка скидається до початкового стану.

4.1.2 Breadth First Search

Алгоритм пошуку в ширину BFS був одним з перших алгоритмів, розроблених для обходу графів [8]. Він був введений Едсгером Дейкстрою в 1959 році як частина його робіт над алгоритмами пошуку найкоротших шляхів у графах. BFS є фундаментальним алгоритмом в інформатиці і широко використовується в багатьох додатках, включаючи навігацію, пошук інформації в базах даних та вирішення задач штучного інтелекту.

Алгоритм BFS починає зі стартової вершини графа і обробляє всі суміжні вершини на поточному рівні перед переходом до наступного рівня. Це робить його придатним для знаходження найкоротших шляхів у неважених графах. BFS використовує структуру даних "черга" для управління фронтом пошуку.

Кроки алгоритму:

- додати стартову вершину до черги і позначити її як відвідану;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з початку черги;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана, додати її до черги і позначити як відвідану;
- записати попередника для кожної вершини, щоб можна було відновити шлях;
- якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;

– якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Наприклад, обхід графу, зображеного на рисунку 4.4, виглядає так «a -> b -> c -> d -> e -> f -> g -> h»

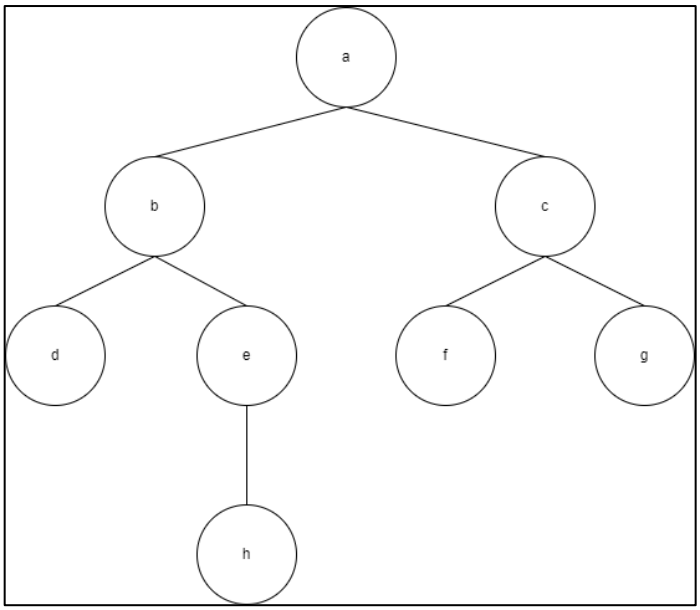


Рисунок 4.4 – Граф для обходу Breadth First Search (створено самостійно)

Далі, на рисунку 4.5 наведено приклад роботи алгоритму Breadth First Search у ігровому рушії на сітці розміром 17 на 17 тайлів.

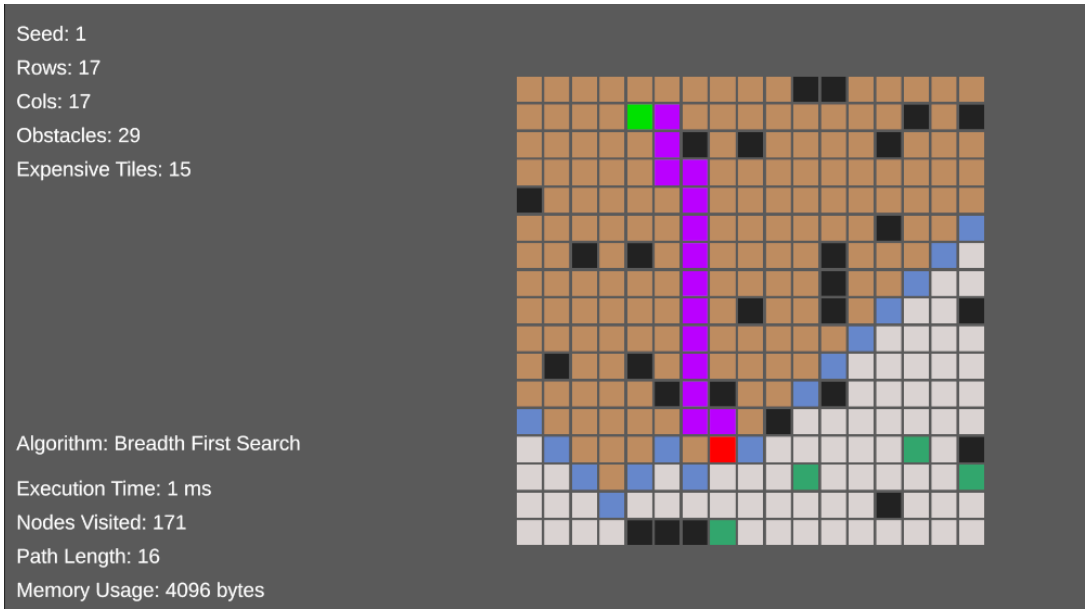


Рисунок 4.5 – Приклад роботи BFS на сітці 17x17 тайлів

BFS гарантує знаходження найкоротшого шляху в неважених графах. Алгоритм може вимагати значного обсягу пам'яті, особливо для великих графів, оскільки всі вершини на поточному рівні зберігаються у черзі.

Часова складність BFS становить O(V + E), де V — кількість вершин, E — кількість ребер графа. Просторова складність алгоритму BFS дорівнює O(V), оскільки всі використані структури даних потребують пам'яті, пропорційної кількості вершин у графі. Всі вершини повинні бути позначені як відвідані, щоб уникнути повторного оброблення та зациклення.

4.1.3 Dijkstra

Алгоритм Dijkstra є одним з найвідоміших алгоритмів для пошуку найкоротших шляхів у графах з невід'ємними вагами ребер [9]. Він був розроблений Едсгером Дейкстрою в 1956 році та опублікований у 1959 році. Алгоритм широко використовується в комп'ютерних мережах, GPS-навігації та багатьох інших застосуваннях, де важливо знайти найкоротший шлях між двома точками.

Алгоритм Дейкстри починає з стартової вершини графа і поступово розширює множину оброблених вершин, завжди вибираючи вершину з найменшою відомою вартістю шляху до неї. Він використовує пріоритетну чергу для управління фронтом пошуку.

Кроки алгоритму:

- встановити вартість для всіх вершин як нескінченність, окрім стартової вершини, для якої вартість дорівнює нулю;
- лодати стартову вершину до пріоритетної черги;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з найменшою вартістю з черги;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана, обчислити нову вартість шляху до неї;
- якщо нова вартість менша за поточну, оновити вартість і додати вершину до черги;

– записати попередника для кожної вершини, щоб можна було відновити шлях;
– якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
– якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Наприклад, обхід графу, зображеного на рисунку 4.6, виглядає так:

– шлях від a до h: a -> b -> e -> h з вартістю 5;
– шлях від a до g: a -> c -> g з вартістю 5;
– шлях від a до d: a -> b -> d з вартістю 5;
– шлях від a до f: a -> c -> f з вартістю 2;
– шлях від a до e: a -> b -> e з вартістю 3;
– шлях від a до b: a -> b з вартістю 2;
– шлях від a до c: a -> c з вартістю 1.

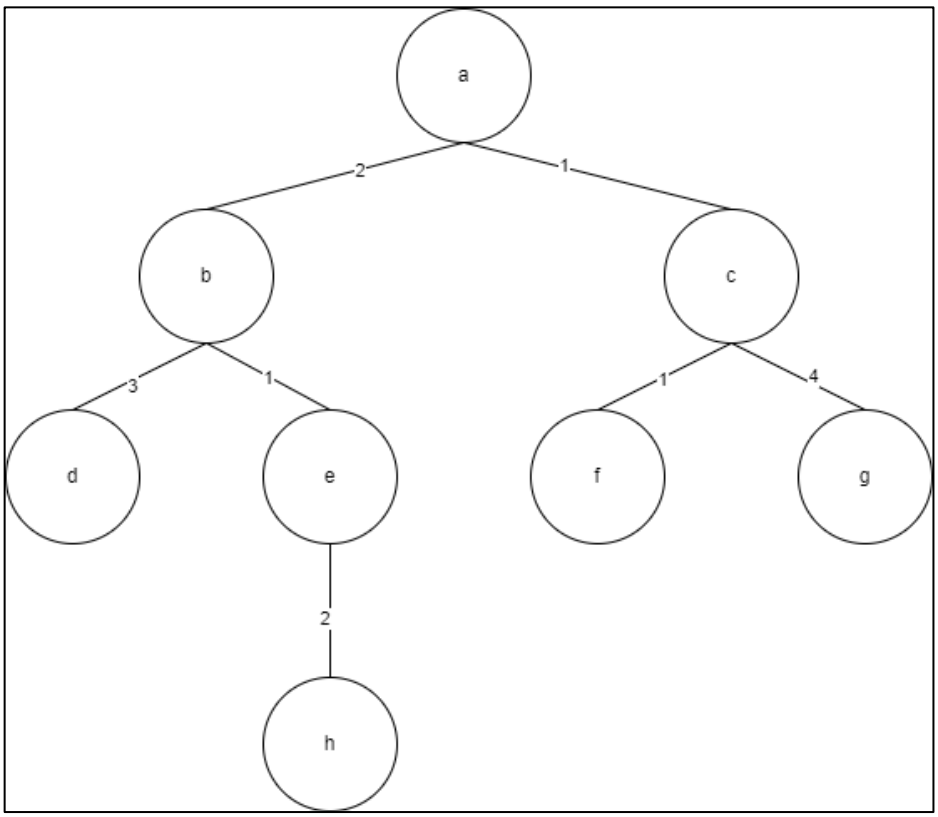


Рисунок 4.6 –Графу для обходу за допомогою алгоритму Dijkstra (створено самостійно)

На рисунку 4.7 наведено приклад роботи алгоритму Dijkstra у ігровому рушії на сітці розміром 17 на 17 тайлів. На тайлах написано вартість шляху від початкової точки до тайлу. Зелені тайли – тайли, вартість руху по яким більша за одиницю.

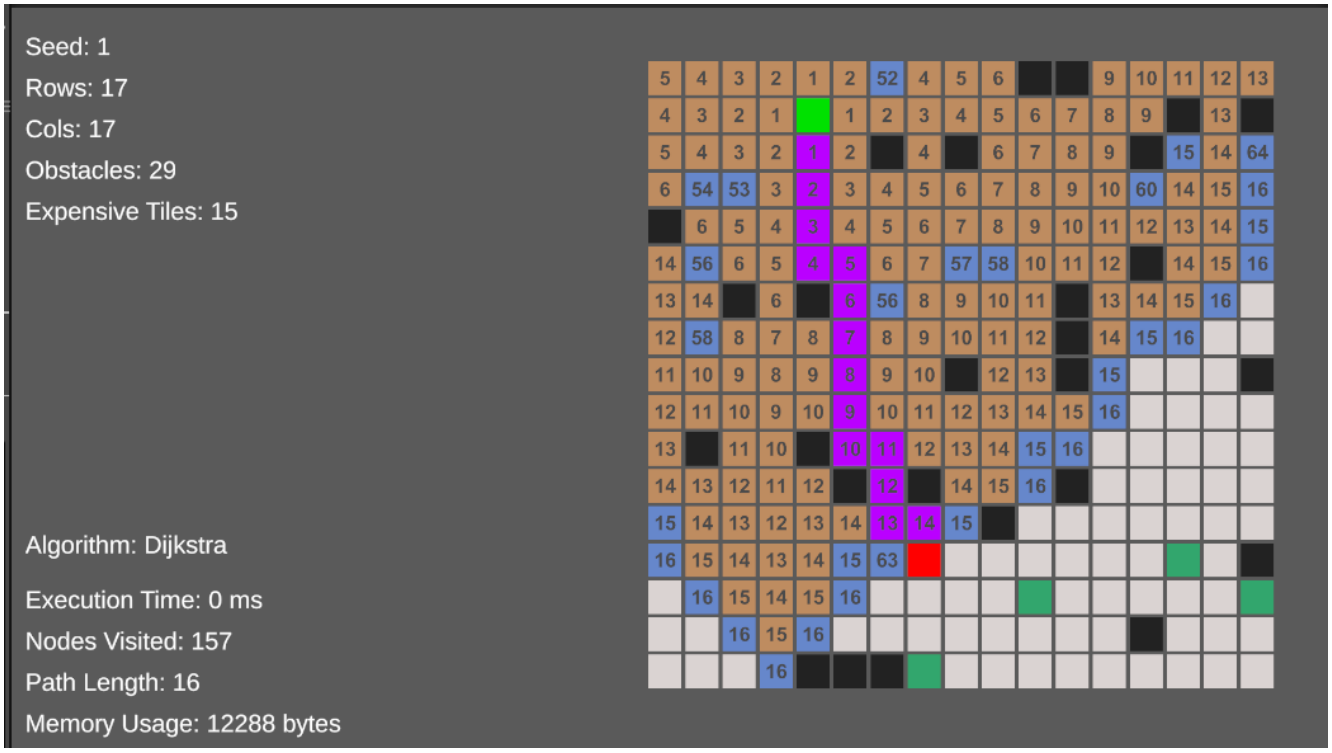


Рисунок 4.7 – Приклад роботи алгоритму Дейкстри на сітці 17x17 тайлів

Алгоритм Дейкстри є ефективним і надійним для знаходження найкоротших шляхів у графах з невід'ємними вагами ребер. Він забезпечує точні результати і може бути застосований до різних типів задач. Алгоритм може вимагати значного обсягу пам'яті, особливо для великих графів, оскільки всі вершини зберігаються у пріоритетній черзі.

У загальному випадку, якщо використовується непріоритетна черга (наприклад, звичайнийсписок), алгоритм Дейкстри має часову складність O(V^2), де V - кількість вершин у графі. Однак, якщо для підтримки операції видалення вершини з найменшою вагою шляхувикористовується пріоритетна, то часова складність алгоритму Дейкстри стає O((V+E)logV), V -кількість вершин, E - кількість ребер у графі, який використовує пріоритетну чергу. Отже,

використання пріоритетної черги дозволяє значно зменшити часову складність алгоритмуДейкстри, що робить його більш ефективним для великих графів з великою кількістю ребер.

Просторова складність алгоритму Dijkstra дорівнює (O(V)), оскільки всі використані структури даних потребують пам'яті, пропорційної кількості вершин у графі. Всі вершини повинні бути позначені як відвідані, щоб уникнути повторного оброблення та зациклення.

4.1.4 AStar

Алгоритм A* є одним з найефективніших алгоритмів для пошуку найкоротших шляхів у графах [10]. Він був розроблений Пітером Хартом, Нільсом Нільсоном та Бертом Рафаелем у 1968 році. A* комбінує властивості пошуку за найменшою вартістю та жадібного найкращого пошуку, що дозволяє йому знаходити оптимальний шлях швидше за інші методи.

Алгоритм A* використовує евристичну функцію для оцінки вартості шляху, яка допомагає направляти пошук у бік цільової вершини, зменшуючи кількість оброблених вершин.

Кроки алгоритму:

- встановити вартість для всіх вершин як нескінченність, окрім стартової вершини, для якої вартість дорівнює нулю;
- додати стартову вершину до пріоритетної черги;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з найменшою вартістю (вартість шляху + евристична вартість) з черги;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана, обчислити нову вартість шляху до неї;
- якщо нова вартість менша за поточну, оновити вартість і додати вершину до черги;
- записати попередника для кожної вершини, щоб можна було відновити шлях;

– якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
– якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Наприклад, обхід графу, зображеного на рисунку 4.8, виглядає так: шлях від a до h: a -> c -> f -> e -> h з вартістю 5.

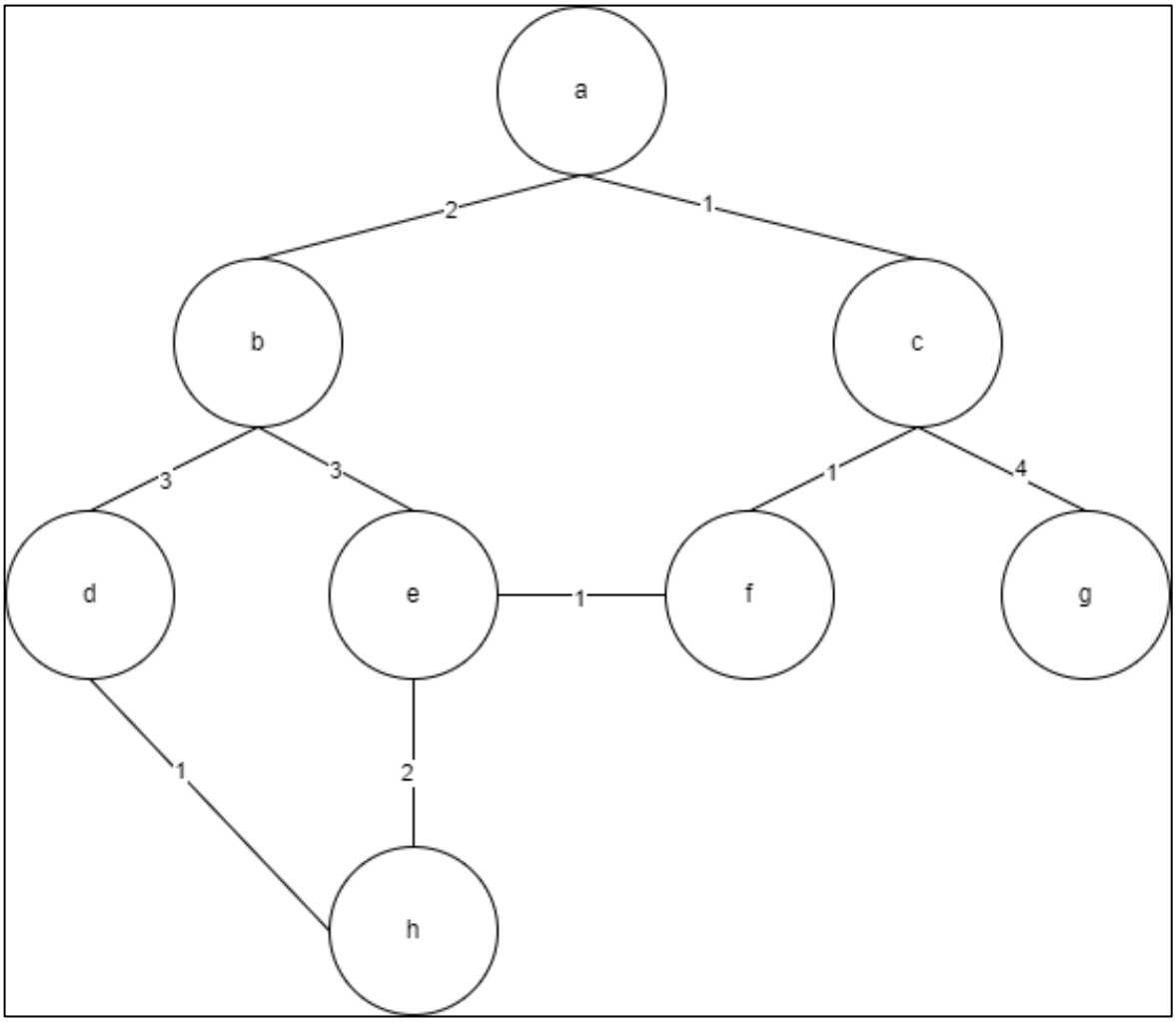


Рисунок 4.8 – Граф для обходу за допомогою алгоритму A* (створено самостійно)

На рисунку 4.9 наведено приклад роботи алгоритму A* у ігровому рушії на сітці розміром 17 на 17 тайлів. На тайлах написано вартість шляху від початкової точки до тайлу. З рисунку видно, що алгоритм прямує до цільової точки.

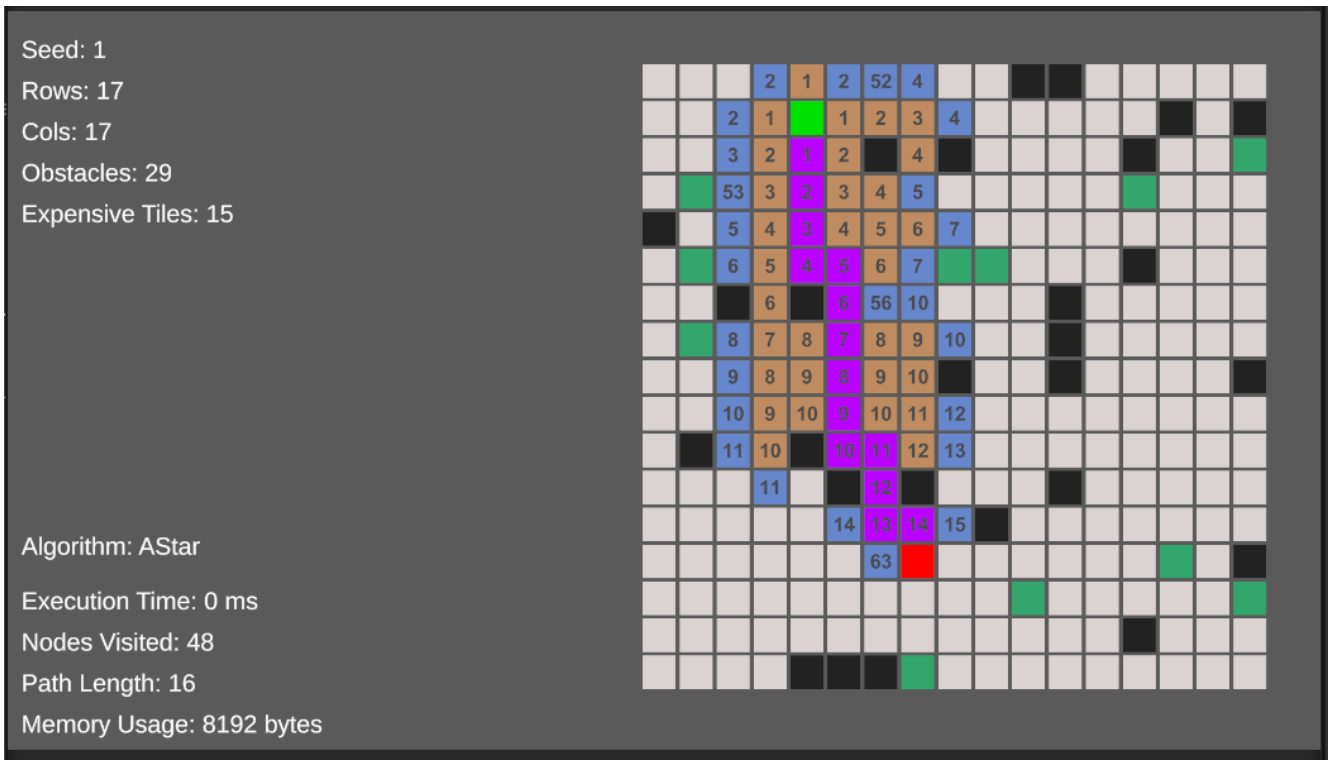


Рисунок 4.9 – Приклад роботи алгоритму A* на сітці 17x17 тайлів (створено самостійно)

Алгоритм A* є ефективним і надійним для знаходження найкоротших шляхів у графах. Він забезпечує точні результати, використовуючи евристичні функції, що направляють пошук до цільової вершини. Алгоритм може вимагати значного обсягу пам'яті, особливо для великих графів, оскільки всі вершини зберігаються у пріоритетній черзі.

Часова складність алгоритму A* залежить від евристики. У гіршому випадку, кількість вершин, досліджуваних алгоритмом, зростає експоненційно порівняно з довжиною оптимального шляху, але складність стає поліноміальною, коли простір пошуку є деревом, а евристика задовольняє таку умову:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

де h^* - оптимальна евристика, тобто точна оцінка відстані з вершини x до мети. Іншими словами, помилка h(x) не повинна зростати швидше, ніж логарифм від оптимальної евристики. Просторова складність алгоритму A* дорівнює (O(V)), оскільки всі використані структури даних потребують пам'яті,

пропорційної кількості вершин у графі. Всі вершини повинні бути позначені як відвідані, щоб уникнути повторного оброблення та зациклення.

4.1.5 Greedy Best First Search

Алгоритм Greedy Best First Search (Жадібний Пошук з Пріоритетом) є евристичним методом пошуку, який використовується для знаходження шляху в графах [11]. Цей алгоритм прагне до швидкого знаходження шляху шляхом вибору вершин, які мають найменшу евристичну вартість, тобто ті, що на його думку, найближчі до цільової вершини. Він не гарантує знаходження оптимального шляху, але часто використовується завдяки своїй простоті та швидкості.

Кроки алгоритму:

- додати стартову вершину до пріоритетної черги;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з найменшою евристичною вартістю з черги;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана, додати її до черги та відзначити як відвідану;
- записати попередника для кожної вершини, щоб можна було відновити шлях;
- якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
- якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Обхід графу від a до h, зображеного на рисунку 4.10, виглядає так: a -> b -> e -> h .

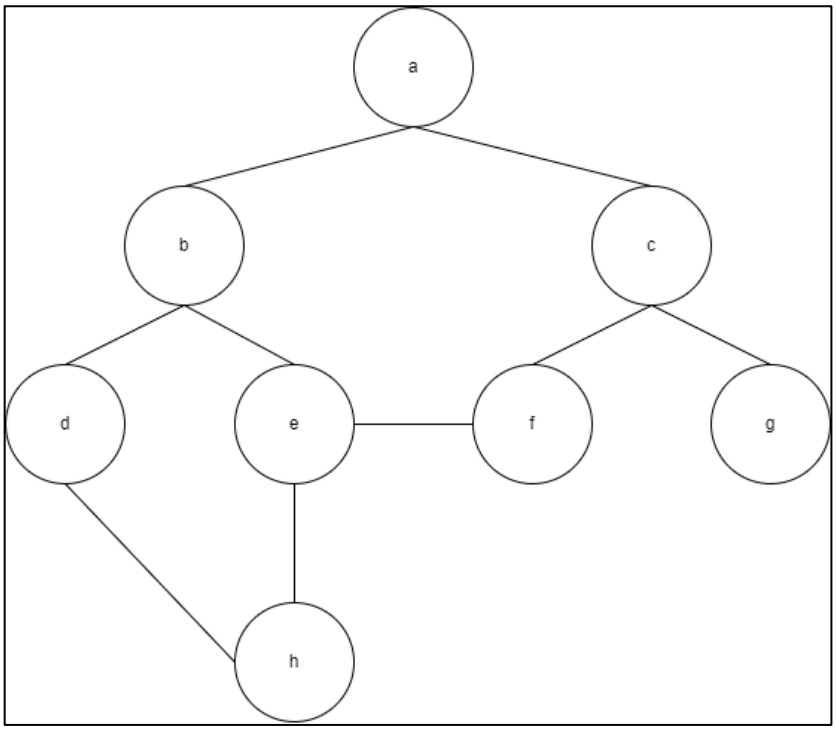


Рисунок 4.10 – Граф для обходу за допомогою алгоритму Greedy Best First Search (створено самостійно)

На рисунку 4.11 наведено приклад роботи алгоритму Greedy Best First Search у ігровому рушії на сітці розміром 17 на 17 тайлів.

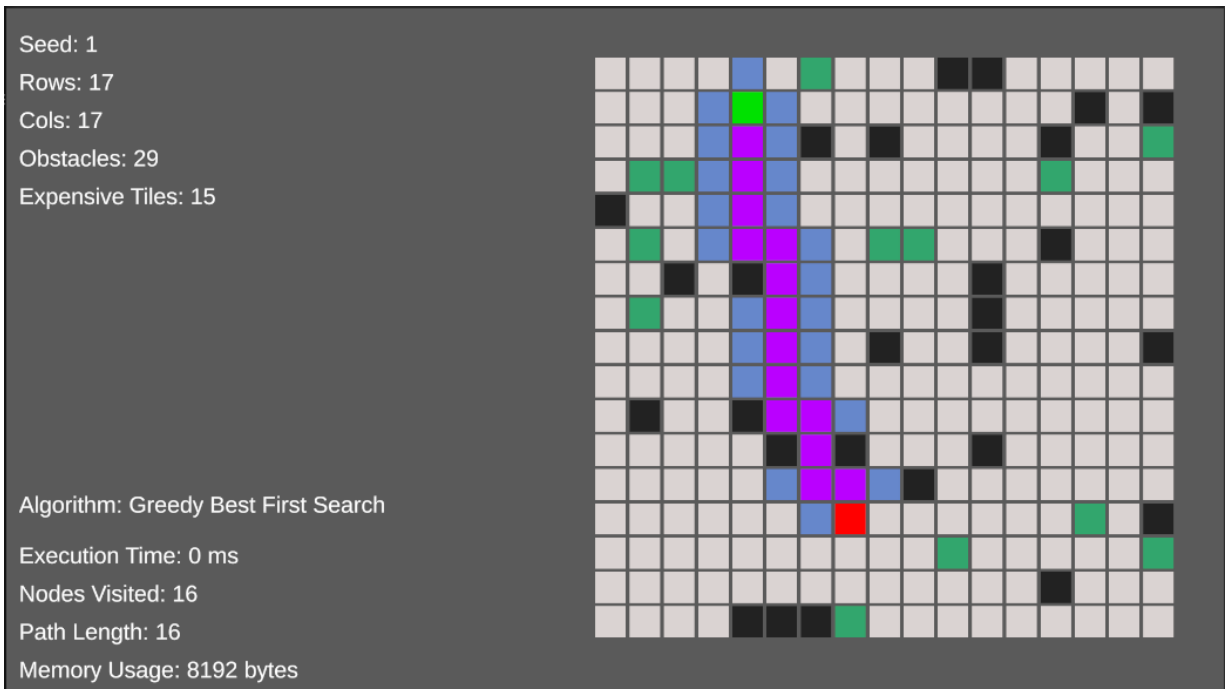


Рисунок 4.11 – Приклад роботи алгоритму Greedy Best First Search на сітці 17x17 тайлів (створено самостійно)

Алгоритм Greedy Best First Search обирає вершини на основі евристичної функції, яка оцінює вартість до цільової вершини. Це робить алгоритм швидким, але він не гарантує знаходження найкоротшого шляху, оскільки не враховує поточну вартість шляху, лише евристику.

Часова складність

Часова складність алгоритму Greedy Best First Search залежить від кількості вершин V та ребер E у графі. У найгіршому випадку алгоритм може відвідати всі вершини графа, і тому його часова складність оцінюється як O(V+E). Оскільки алгоритм використовує пріоритетну чергу для управління фронтом пошуку, операції вставки та видалення з черги мають складність O(logV). Таким чином, загальна часова складність алгоритму з урахуванням роботи з чергою буде O(VlogV+ElogV). Проте, оскільки кількість ребер у графі зазвичай обмежена деякою функцією від кількості вершин, найчастіше часова складність Greedy Best First Search записується як O(VlogV).

Просторова складність алгоритму Greedy Best First Search визначається обсягом пам'яті, необхідним для зберігання даних про вершини та чергу. У найгіршому випадку алгоритм може вимагати зберігання інформації про всі вершини графа та їх вартість у пріоритетній черзі. Тому просторова складність алгоритму Greedy Best First Search оцінюється як: O(V).

Використовується у задачах, де потрібна швидка реакція, наприклад, в ігрових рушіях для навігації персонажів, системах навігації для роботів та інших сферах, де швидкість важливіша за точність.

4.1.6 Depth First Search

Алгоритм Depth First Search є одним із фундаментальних алгоритмів для обходу або пошуку в графах. Він був розроблений Едсгером Дейкстрою в 1959 році [12]. DFS працює за принципом глибини, тобто він проходить від початкової вершини до найглибшої можливості, перш ніж відступати назад.

Алгоритм DFS використовує структуру даних "стек" для управління фронтом пошуку. Він підходить для задач, де важливо досліджувати всі можливі шляхи або де глибина пошуку має значення.

Кроки алгоритму:

- додати стартову вершину до стека і позначити її як відвідану;
- повторювати наступні кроки, поки стек не порожній;
- вийняти вершину з вершини стека;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана, додати її до стека і позначити як відвідану;
- записати попередника для кожної вершини, щоб можна було відновити шлях;
- якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
- якщо стек порожній і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Обхід графу, зображеного на рисунку 4.12, виглядає так:

- a -> b -> e -> l;
- a -> b -> e -> m;
- a -> b -> f;
- a -> c;
- a -> d -> g -> n;
- a -> d -> g -> o;
- a -> d -> h.

Алгоритм завжди прямує в глибину.

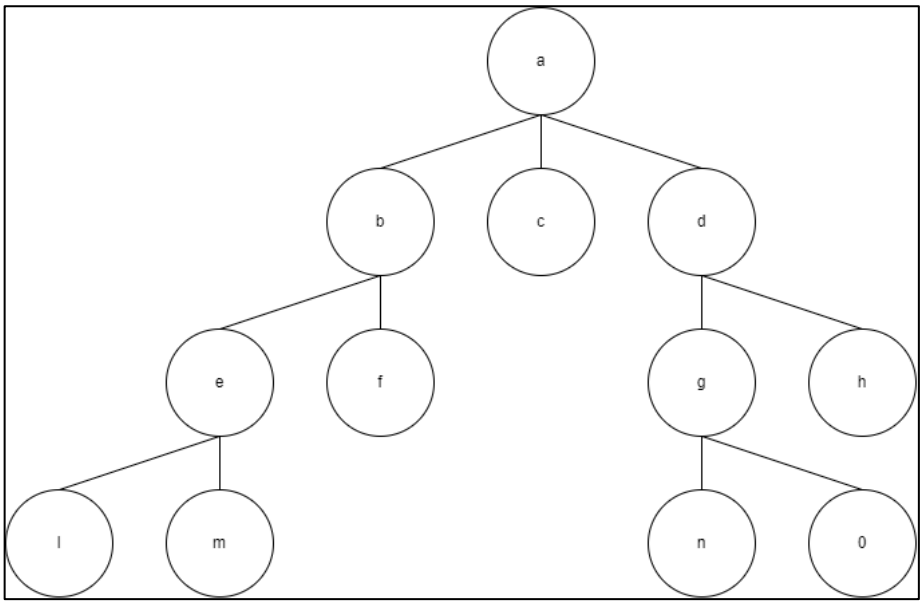


Рисунок 4.12 – Граф для обходу за допомогою алгоритму DFS (створено самостійно)

На рисунку 4.13 наведено приклад роботи алгоритму DFS у ігровому рушії на сітці розміром 17 на 17 тайлів. З рисунку видно, що алгоритм прямує до цільової точки шляхом обходу в глибину, але так як наш граф – це сітка, де кожна вершина пов’язана з іншою, то шлях виходить досить довгим, в порівнянні з іншими алгоритмами.

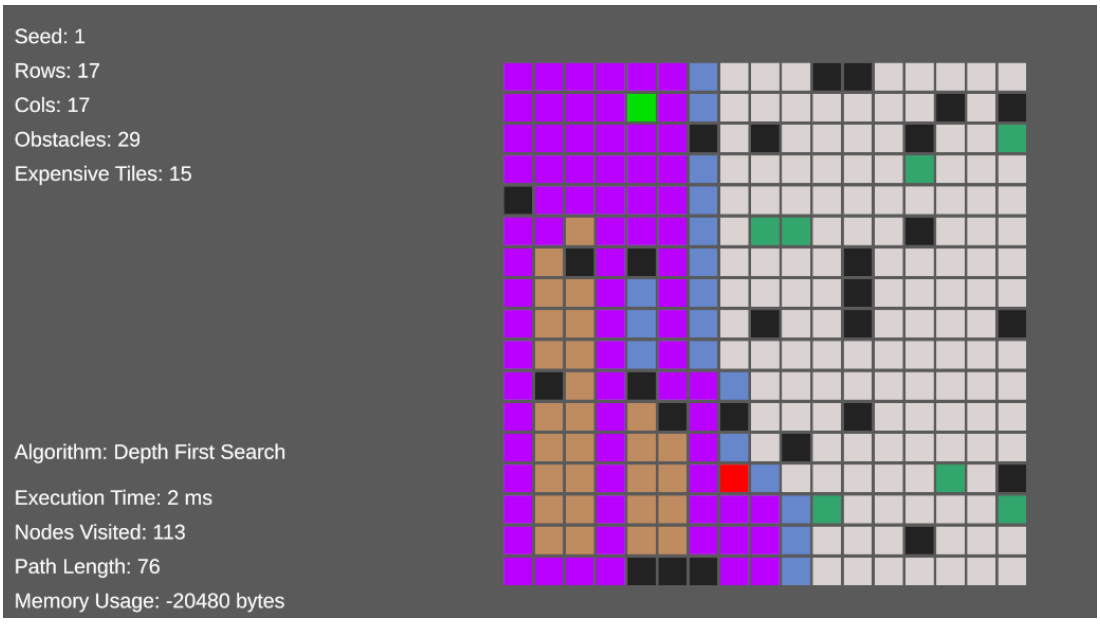


Рисунок 4.13 – Приклад роботи алгоритму DFS на сітці 17x17 тайлів (створено самостійно)

Алгоритм DFS може виявити найглибші шляхи в графі, але він не гарантує знаходження найкоротшого шляху. Це робить його придатним для задач, де глибокі шляхи мають більшу важливість. Очевидним застосуванням цього алгоритму буде проходження лабіринту. На практиці DFS найчастіше використовують для перевірки графа на зв'язність, або узагальнено, пошуку компонентів зв'язності, пошуку циклів і роботи з деревами.

Часова складність (O(V + E), просторова складність (O(V)), оскільки всі використані структури даних потребують пам'яті, пропорційної кількості вершин у графі.

4.1.7 Bidirectional Search

Алгоритм Bidirectional Search є потужним методом для пошуку найкоротших шляхів у графах [13]. Він працює одночасно з двох напрямків — від початкової вершини та від цільової вершини, доки обидва фронти пошуку не зустрінуться. Це дозволяє значно зменшити кількість оброблених вершин, що робить алгоритм дуже ефективним у багатьох випадках.

Bidirectional Search був введений Пітером Хартом, Нільсом Нільсоном та Бертом Рафаелем у 1968 році. Цей алгоритм є ефективним для знаходження найкоротшого шляху у великих графах.

Кроки алгоритму:

- ініціалізувати два набори відвіданих вершин: один для пошуку від початкової вершини, інший для пошуку від цільової вершини;
- ініціалізувати дві черги для фронтів пошуку: одну для початкової вершини, іншу для цільової вершини;
- додати початкову вершину до першої черги і цільову вершину до другої черги;
- повторювати наступні кроки, поки обидві черги не порожні;
- вийняти вершину з черги, що веде від початкової вершини, і перевірити всі суміжні вершини;
- якщо суміжна вершина вже відвідана з іншого фронту, пошук завершено;

- якщо вершина ще не відвідана, додати її до черги і позначити як відвідану;
- вийняти вершину з черги, що веде від цільової вершини, і перевірити всі суміжні вершини;
- якщо суміжна вершина вже відвідана з іншого фронту, пошук завершено;
- якщо вершина ще не відвідана, додати її до черги і позначити як відвідану;
- якщо черги порожні і цільова вершина не знайдена, повернути повідомлення, що шлях не існує;
- побудувати шлях, використовуючи зустрічну вершину як точку з'єднання.

Наприклад, обхід графу, зображеного на рисунку 4.14, від точки a до точки m виглядає так: рухаємося від a до m та від m до a

- дві черги: одна починається з a, інша з m;
- відвідані вершини для кожного фронту пошуку: {a} і {m};
- розширення черги з a, додавання b, c, d;
- розширення черги з m, додавання l, e, f, n;
- розширення черги з b, додавання e, f;
- розширення черги з l, додавання e;
- розширення черги з c, додавання f, g;
- розширення черги з f, зустріч з обома фронтами.

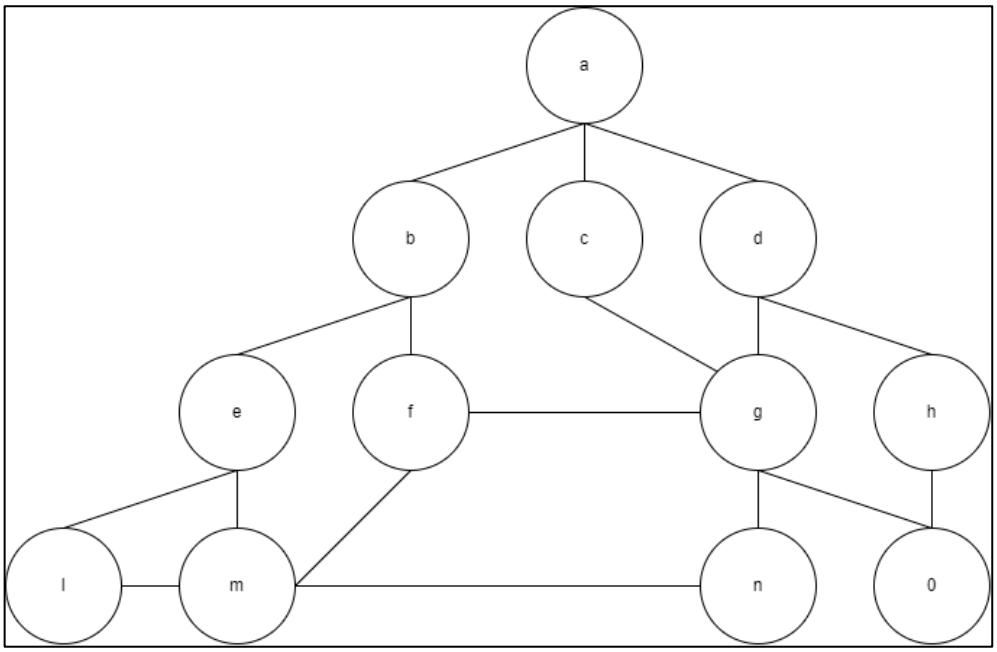


Рисунок 4.14 – Граф для обходу за допомогою алгоритму Bidirectional Search (створено самостійно)

На рисунку 4.15 наведено приклад роботи алгоритму Bidirectional Search у ігровому рушії на сітці розміром 17 на 17 тайлів.. З рисунку видно, що алгоритм прямує до цільової точки з двох сторін.

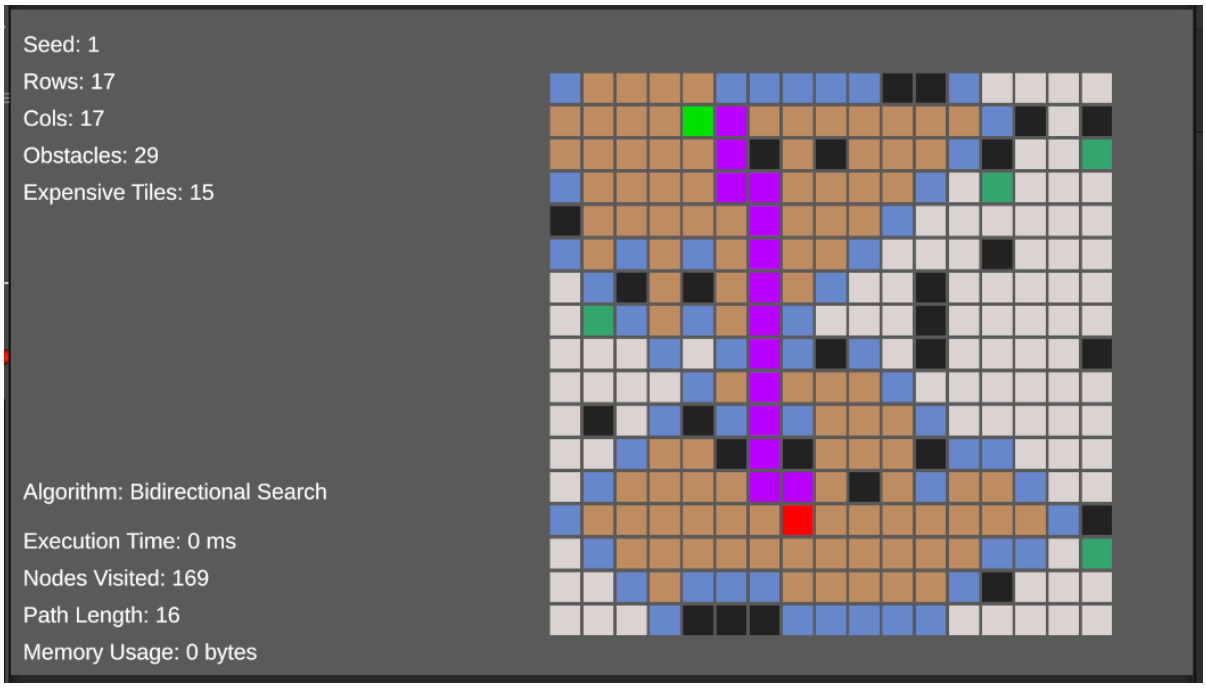


Рисунок 4.15 – Приклад роботи алгоритму Bidirectional Search на сітці 17x17 тайлів (створено самостійно)

Алгоритм Bidirectional Search може значно скоротити кількість оброблених вершин порівняно з односпрямованими методами пошуку.

Часова складність $O(b^{d/2})$ де b – середня кількість суміжних вершин для кожної вершини, d – довжина найкоротшого шляху між початковою та цільовою вершинами. Часова складність обчислюється на основі того, що кожен фронт пошуку охоплює приблизно половину шляху.

У книзі Norvig & Russell зазначено, що просторова складність двонаправленого пошуку, що відповідає найбільшій можливій кількості вузлів, які ви зберігаєте на межі $O(2b^{d/2})= O(b^{d/2})$.

4.1.8 Lee Algorithm

Алгоритм Лі був розроблений американським інженером Кларенсом А. Лі у 1961 році [14]. Він запропонував цей алгоритм для використання в автоматизованій маршрутизації друкованих плат, але згодом алгоритм знайшов широке застосування у задачах пошуку шляху в двовимірних сітках та інших графових структурах. Алгоритм Lee гарантує знаходження найкоротшого шляху в неважених графах, але при цьому може бути повільнішим у порівнянні з іншими методами для великих графів.

Алгоритм Lee працює за принципом пошуку в ширину (Breadth-First Search), розширюючи фронт пошуку рівномірно у всі сторони від початкової точки до цільової точки.

Кроки алгоритму:

- ініціалізувати матрицю відстаней, встановивши всі значення на нескінченність, окрім стартової точки, для якої відстань дорівнює нулю;
- додати стартову точку до черги;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з черги;
- перевірити всі суміжні вершини;
- якщо вершина ще не відвідана і не є перешкодою, оновити її відстань і додати до черги;

– записати попередника для кожної вершини, щоб можна було відновити шлях;
– якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
– якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Наприклад, обхід графу, зображеного на рисунку 4.16, виглядає так: шлях від a до m: a -> b -> e -> m.

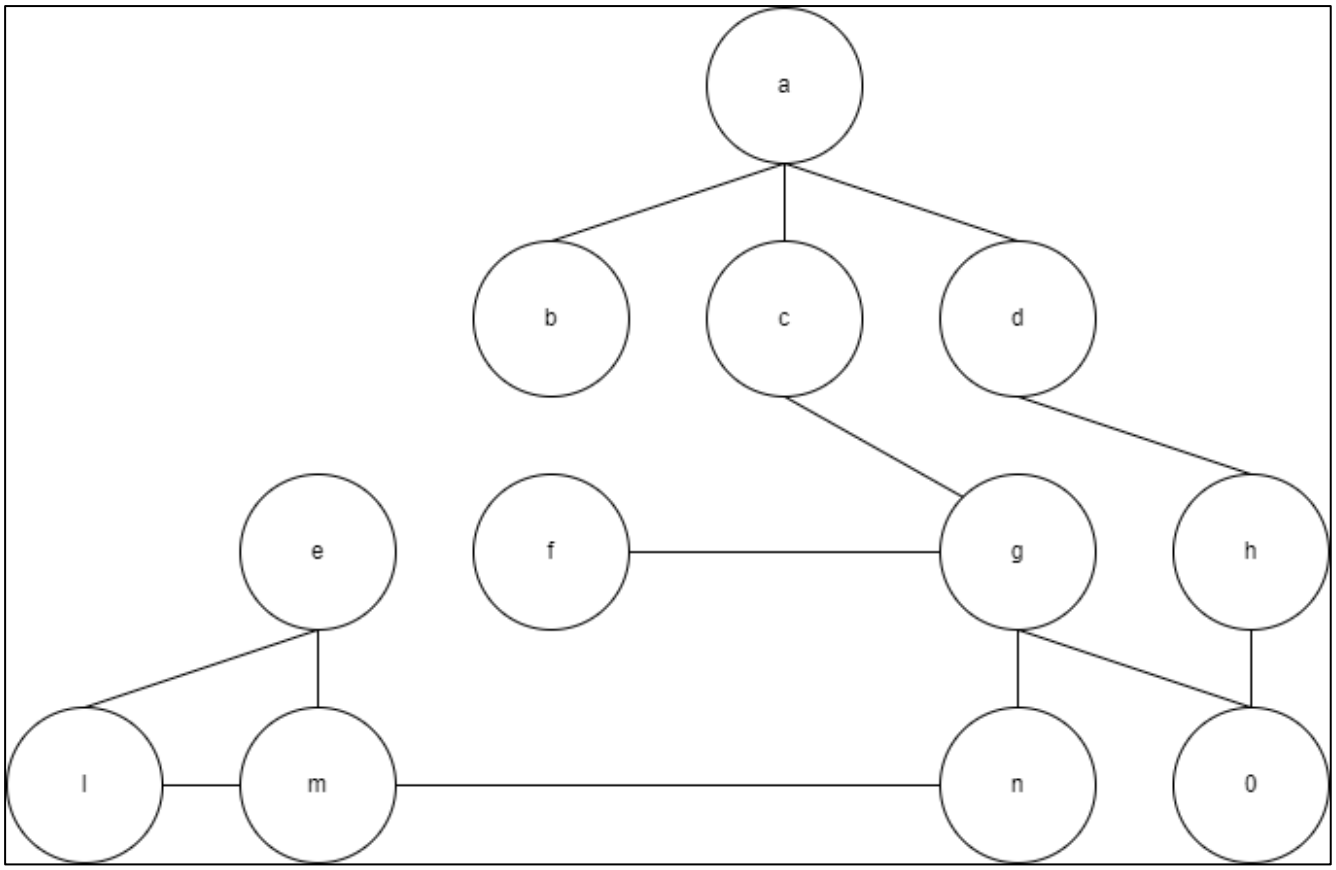


Рисунок 4.16 – Граф для обходу за допомогою алгоритму Lee (створено самостійно)

На рисунку 4.17 наведено приклад роботи алгоритму Lee у ігровому рушії на сітці розміром 17 на 17 тайлів. На тайлах написано вартість шляху від початкової точки до тайлу. З рисунку видно, що алгоритм рівномірно розширює фронт пошуку до цільової точки.

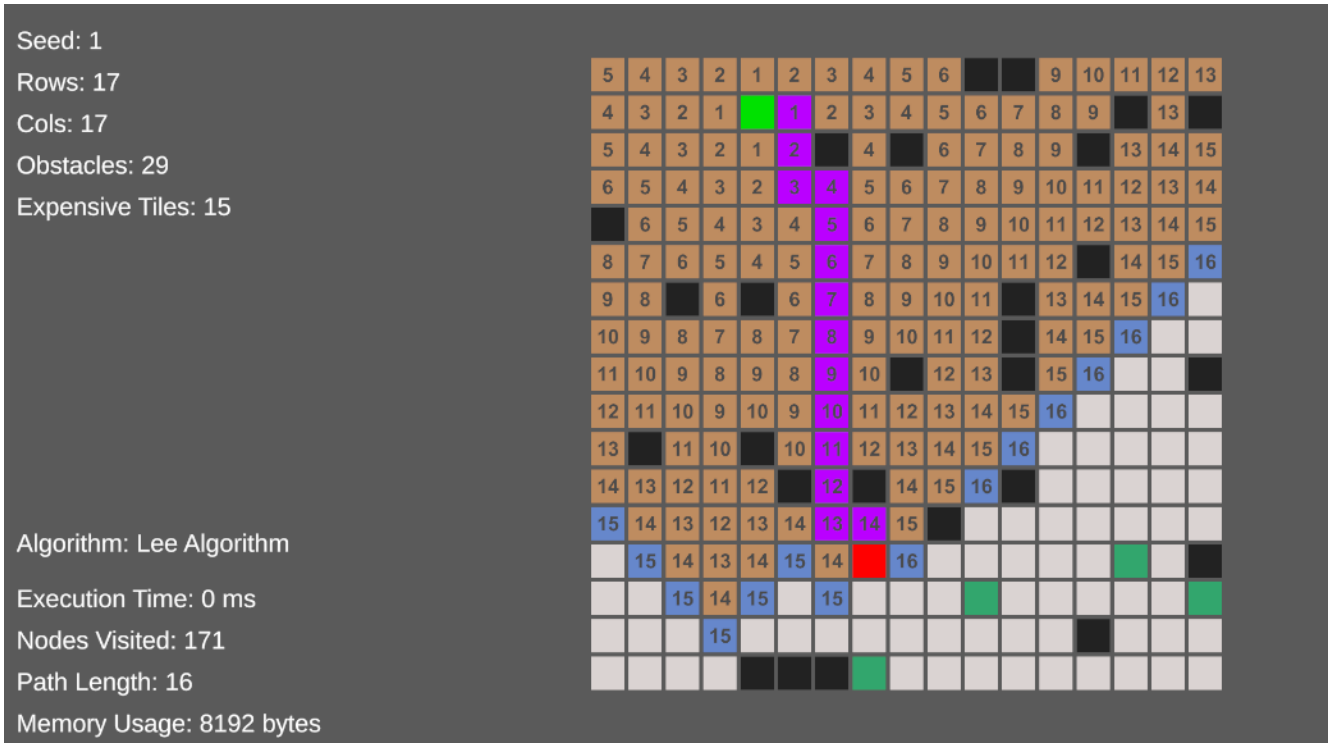


Рисунок 4.17 – Приклад роботи алгоритму Lee на сітці 17x17 тайлів (створено самостійно)

Алгоритм Lee є надійним та ефективним для знаходження найкоротших шляхів у двовимірних сітках, особливо у випадках, коли важливо гарантувати знаходження оптимального шляху.

Часова складність (O(V + E)). Просторова складність (O(V)), оскільки всі використані структури даних потребують пам'яті, пропорційної кількості вершин у графі.

4.1.9 Dynamic Programming Maze

Алгоритм Dynamic Programming Maze, як і багато інших методів динамічного програмування, є результатом розвитку теорії динамічного програмування, яка була значною мірою розроблена американським математиком Річардом Беллманом у 1950-х роках [15]. Річард Беллман ввів поняття динамічного програмування і зробив значний внесок у розвиток цієї теорії.

Сам алгоритм, відомий як Dynamic Programming Maze, не має конкретного автора, оскільки це застосування загальних принципів динамічного

програмування до конкретної задачі пошуку шляху в лабіринтах або сітках. Алгоритми на основі динамічного програмування використовують загальні принципи оптимізації, які були розроблені багатьма дослідниками в галузі інформатики та математики.

Алгоритм Dynamic Programming Maze використовує метод динамічного програмування для знаходження найкоротшого шляху у двовимірних сітках. Цей алгоритм працює шляхом поступового оновлення вартості шляхів від стартової точки до кожної вершини, забезпечуючи найкоротший шлях до цільової вершини. Він особливо ефективний у випадках, коли потрібно знайти шлях через лабіринти або сітки з різними вагами вершин.

Алгоритм Dynamic Programming Maze працює подібно до алгоритму Дейкстри, але з деякими відмінностями в підході до оновлення вартостей та обробки вершин.

Кроки алгоритму:

- ініціалізувати матрицю відстаней, встановивши всі значення на нескінченність, окрім стартової точки, для якої відстань дорівнює нулю;
- додати стартову точку до черги;
- повторювати наступні кроки, поки черга не порожня;
- вийняти вершину з черги;
- перевірити всі суміжні вершини;
- якщо нова відстань менша за поточну відстань до вершини, оновити її відстань і додати до черги;
- записати попередника для кожної вершини, щоб можна було відновити шлях;
- якщо цільова вершина знайдена, побудувати шлях, використовуючи записані попередники;
- якщо черга порожня і цільова вершина не знайдена, повернути повідомлення, що шлях не існує.

Наприклад, обхід графу, зображеного на рисунку 4.18, виглядає так: шлях від a до l: a -> h -> e -> f -> g -> l. Перед визначенням шляху, алгоритм прохоить та перевіряє усі можливі вершини.

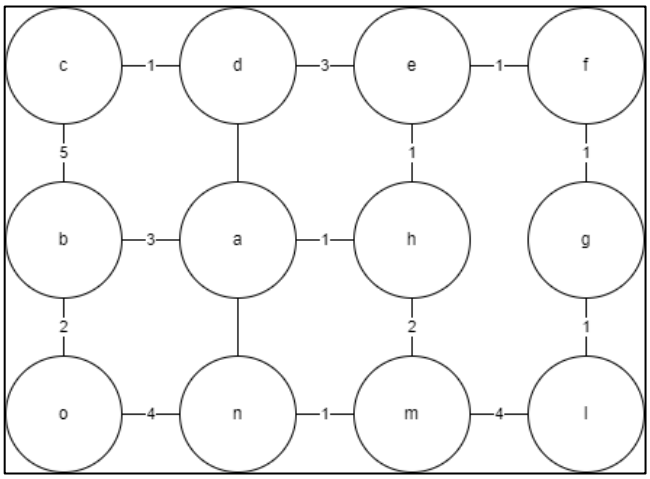


Рисунок 4.18 – Граф для обходу за допомогою алгоритму Dynamic Programming Maze (створено самостійно)

На рисунку 4.19 наведено приклад роботи алгоритму Dynamic Programming Maze у ігровому рушії на сітці розміром 17 на 17 тайлів. На тайлах написано вартість шляху від початкової точки до тайлу. З рисунку видно, що алгоритм рівномірно розширює фронт пошуку до цільової точки, оновлюючи вартості шляхів. Проходить всі тайли, а потім визначає оптимальний шлях.

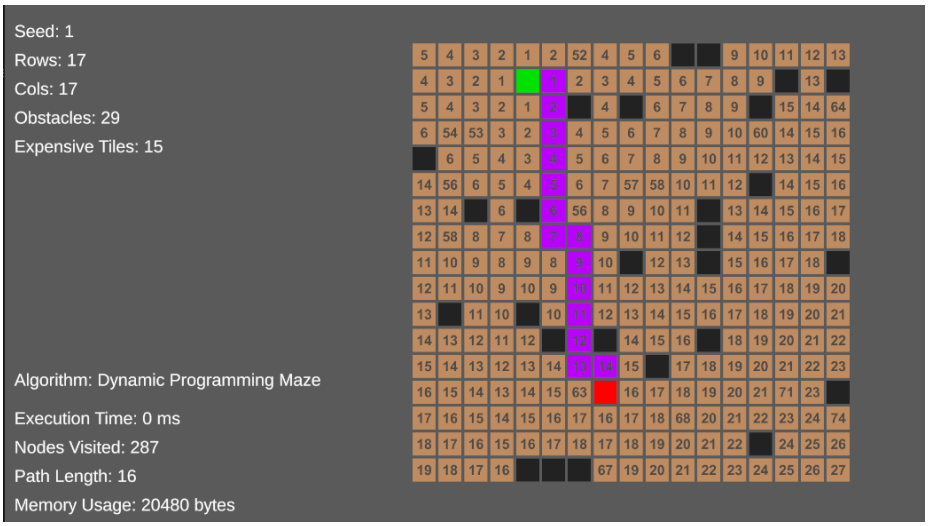


Рисунок 4.19 – Приклад роботи алгоритму Dynamic Programming Maze на сітці 17x17 тайлів (створено самостійно)

Алгоритм Dynamic Programming Maze є надійним та ефективним для знаходження найкоротших шляхів у двовимірних сітках, особливо у випадках, коли важливо гарантувати знаходження оптимального шляху.

Часова складність O(2^(2V)). Просторова складність O(V), оскільки всі використані структури даних потребують пам'яті, пропорційної кількості вершин у графі.

4.1.10 Збереження та відображення даних роботи алгоритмів

Для кожного алгоритму збираються параметри, такі як час виконання, кількість відвіданих вузлів, довжина шляху та використання пам'яті. Ці параметри відображаються на екрані та зберігаються у файл. Результати роботи алгоритмів пошуку шляху виводяться на екран через текстові поля, які показують назву алгоритму, час виконання, кількість відвіданих вузлів, довжину шляху та використання пам'яті. Це проілюстровано на рисунку 4.20.

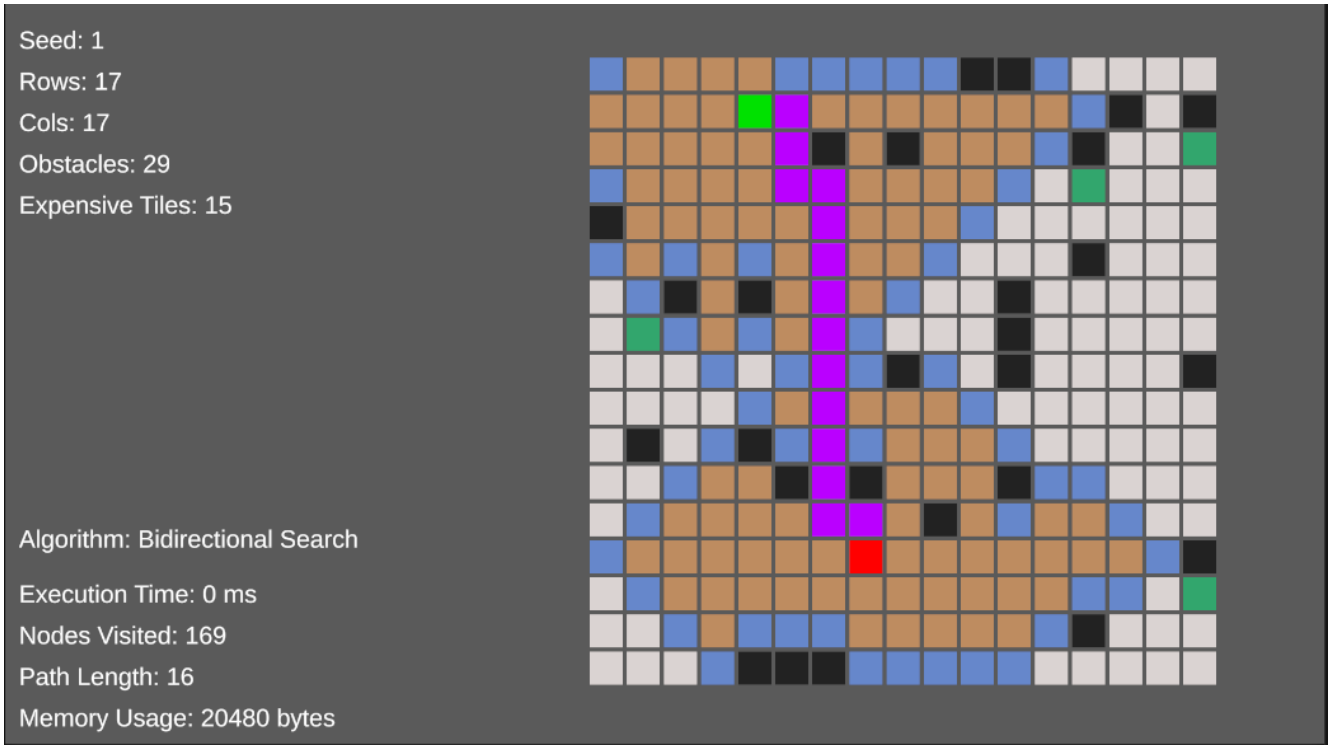


Рисунок 4.20 – Візуалізація результатів роботи алгоритмів (створено самостійно)

На рисунку показано загальну інформацію про сітку, на якій працює алгоритм – у верхній частині, інформацію про роботу алгоритму – у нижній частині та сам результат пошуку шляху – справа. Результати роботи алгоритмів пошуку шляху зберігаються у файл CSV для подальшого аналізу. У файл записуються такі параметри, як назва алгоритму, час виконання, кількість відвіданих вузлів, довжина шляху, використання пам'яті, розміри сітки та значення seed.

Клас TileGrid містить текстові поля, які використовуються для відображення результатів роботи алгоритмів на екрані. Коли алгоритм завершує роботу, метод ShowInfo (див рис 4.21) оновлює ці текстові поля, щоб відобразити результати :

```
Ссылок: 2
private void ShowInfo(string algorithmName, long executionTime,
    int nodesVisited, int pathLength, long memoryUsage)
{
    algorithmNameText.text = $"Algorithm: {algorithmName}";
    executionTimeText.text = $"Execution Time: {executionTime} ms";
    nodesVisitedText.text = $"Nodes Visited: {nodesVisited}";
    pathLengthText.text = $"Path Length: {pathLength}";
    memoryUsageText.text = $"Memory Usage: {memoryUsage} bytes";
    seedText.text = $"Seed: {seed}";
    obstaclesText.text = $"Obstacles: {numberOfObstacles}";
    expensiveText.text = $"Expensive Tiles: {numberOfExpensiveTiles}";
    rowsText.text = $"Rows: {Rows}";
    colsText.text = $"Cols: {Cols}";

    Debug.Log($"Algorithm: {algorithmName}");
    Debug.Log($"Execution Time: {executionTime} ms");
    Debug.Log($"Nodes Visited: {nodesVisited}");
    Debug.Log($"Path Length: {pathLength}");
    Debug.Log($"Memory Usage: {memoryUsage} bytes");
}
```

Рисунок 4.21 – Метод ShowInfo (створено самостійно)

Метод WriteResultsToExcel (див рис 4.22) відповідає за збереження результатів виконання алгоритмів у файл CSV. Він перевіряє наявність файлу та,

якщо файл не існує, створює його та додає заголовки. Потім записує результати виконання алгоритму, такі як назва алгоритму, час виконання, кількість відвіданих вузлів, довжину шляху, використання пам'яті, розміри сітки, значення seed, початкову та кінцеву позиції:

```
Ссылок: 2
private void WriteResultsToExcel(string algorithmName, long executionTime,
    int nodesVisited, int pathLength, long memoryUsage, int rows, int cols, int seed,
    Vector2 startPosition, Vector2 endPosition)
{
    string filePath = "PathfindingResults.csv";

    bool fileExists = File.Exists(filePath);

    using(StreamWriter writer = new StreamWriter(filePath, true))
    {
        if(!fileExists)
        {
            writer.WriteLine("Algorithm,Execution Time (ms),Nodes Visited,Path Length,Memory Usage (bytes)," +
                "Grid Size,Seed,Start Position,End Position");
        }

        writer.WriteLine($"{algorithmName},{executionTime},{nodesVisited},{pathLength},{memoryUsage}," +
            $"{rows}x{cols},{seed},{startPosition.x},{startPosition.y},{endPosition.x},{endPosition.y}");
    }
}
```

Рисунок 4.22 – Метод WriteResultsToExcel (створено самостійно)

Ці фрагменти коду забезпечують виведення інформації про виконання алгоритмів на екран та збереження результатів у файл CSV для подальшого аналізу.

4.2 Алгоритми прийняття рішень

Алгоритми прийняття рішень – це методи та підходи, які використовуються для визначення дій, які повинні виконати агенти у відповідь на певні ситуації. Вони є важливими компонентами штучного інтелекту у відеоіграх, оскільки забезпечують реалістичну та адаптивну поведінку персонажів, роблячи ігровий процес цікавішим та динамічнішим. За допомогою цих алгоритмів, ігрові персонажі можуть реагувати на дії гравців, змінювати свою стратегію залежно від змін у середовищі, а також демонструвати складну поведінку, яка додає глибини та різноманітності ігровому процесу.

Алгоритми прийняття рішень використовуються для реалізації різних аспектів ігрової поведінки, включаючи бойові стратегії, соціальну взаємодію, вирішення головоломок і багато іншого. Вони дозволяють AI персонажам аналізувати навколишнє середовище, оцінювати можливі варіанти дій і вибирати найбільш відповідну відповідь на поточну ситуацію. Це робить ігровий світ більш реалістичним і захоплюючим, адже гравці можуть взаємодіяти з персонажами, які виглядають і поводяться як справжні істоти.

4.2.1 Сцена

Для оцінки та аналізу алгоритмів прийняття рішень було вирішено помістити агентів, що можуть виконувати однакові дії, у певну середу, де вони будуть змагатися між собою (див. рис 4.23).

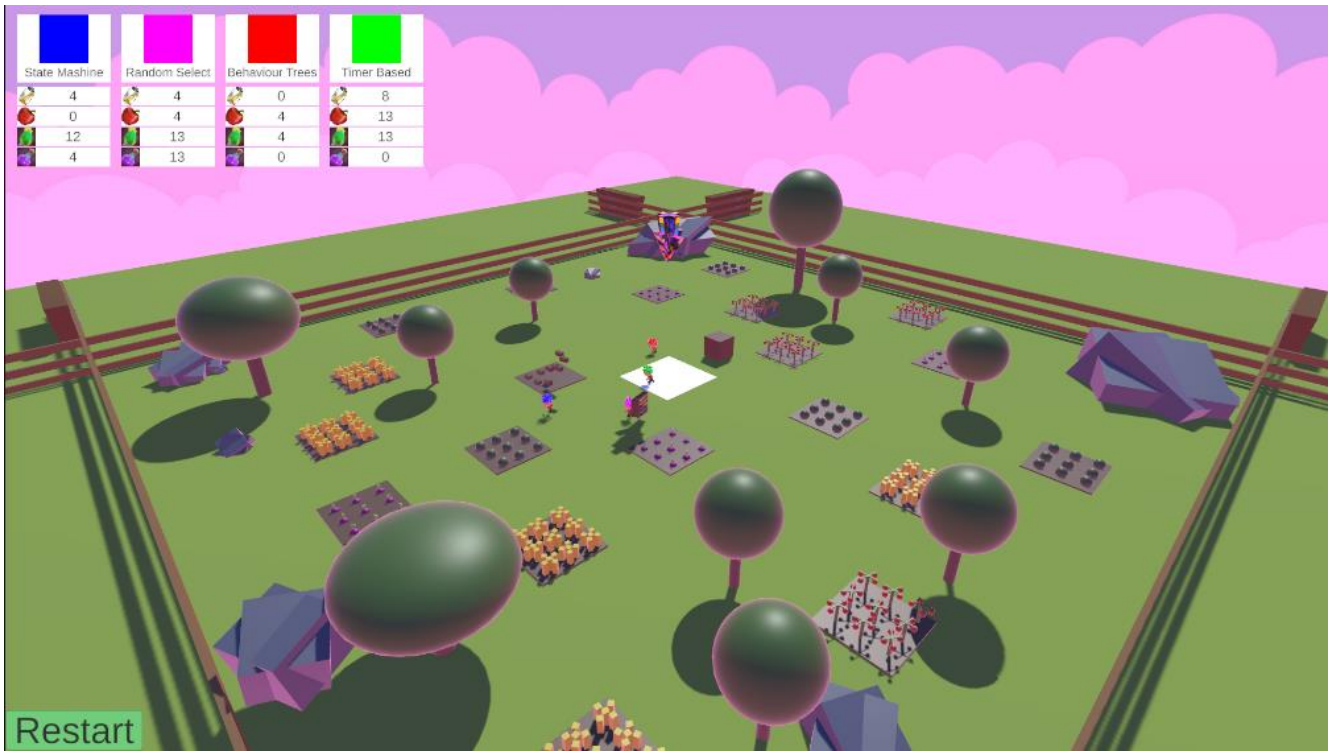


Рисунок 4.23 – Ігрова сцена (створено самостійно)

Змагання полягає у тому, що б зібрати необхідні ресурси швидше, ніж супротивники. Тому хто зібрав усі необхідні ресурси першим, надається перше місце, друге місце надається тому, кому залишилося зібрати найменше по

кількості і так далі. На четвертому місці той, хто зібрав найменше необхідних ресурсів.

Сцена являє собою закриту територію, де можуть переміщуватися агенти.

На запуску програми, клас RivalsGameManager ініціалізує NPC та склад, відповідає за забезпечення взаємодії між різними компонентами гри, скільки та яких ресурсів необхідно зібрати агентам.

В центрі стоїть точка – склад, куди всі агенти приносять зібрані ресурси. Клас RivalsWarehouse є центральним елементом механіки гри, який відповідає за управління ресурсами. Він забезпечує зберігання та додавання ресурсів, перевірку умов перемоги, а також обробку подій, пов'язаних з ресурсами та перемогою. Цей клас інтегрує різні компоненти гри, такі як NPC, склади та таймери, для створення інтерактивного і динамічного ігрового процесу. Інформація про зібрані ресурси зберігається в словнику characterResources. Цей словник містить записи для кожного персонажа ICharacter, де кожен запис містить інший словник, який зберігає кількість зібраних ресурсів ResourceData для цього персонажа.

Реалізація ресурсів у грі включає кілька класів і структур, які забезпечують зберігання даних про ресурси, їх ініціалізацію, збирання та переміщення. ResourceData (див. рис 4.24) зберігає інформацію про тип ресурсу, його іконку, префаб та час відродження.

```
namespace DecisionMaking {
    [CreateAssetMenu(fileName = "ResourceData", menuName = "SO/ResourceData")]
    Скрипт Unity | Ссылок: 35
    public class ResourceData : EntityData {
        [SerializeField] ResourceName resourceName;
        [SerializeField] Sprite icon;
        [SerializeField] GameObject collectedPrefab;
        [SerializeField] float respawnTime;

        Ссылок: 0
        public ResourceName ResourceName => resourceName;
        Ссылок: 2
        public Sprite Icon => icon;
        Ссылок: 1
        public GameObject CollectedPrefab => collectedPrefab;
        Ссылок: 2
        public float RespawnTime => respawnTime;
    }
}
```

Рисунок 4.24 – Код реалізації «ResourceData» (створено самостійно)

Resource (див. рис 4.25) представляє ресурс у грі, , що виростає на грядці, і містить методи для його збирання.

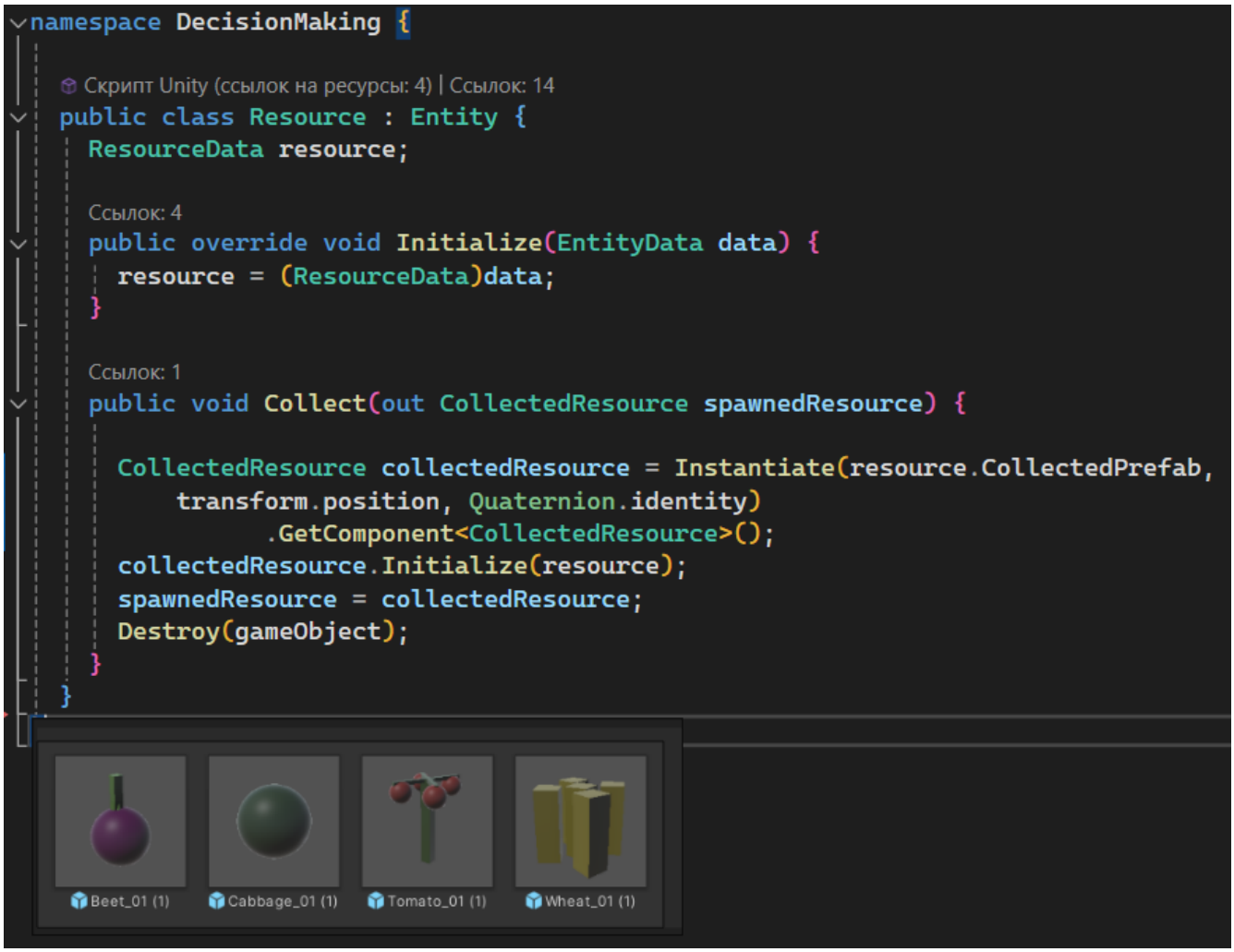


Рисунок 4.25 – Код та приклад «Resource» (створено самостійно)

CollectedResource обробляє анімацію та переміщення зібраного ресурсу до інвентаря або складу. Візуально – це ящик, з кольором зібраного ресурсу. На рисунку 4.26 показано зібраний ресурс – пшеницю. З правої сторони показані скрипти, що відповідають за анімацію польоту до інвентаря NPC. Знизу – 4 інші префааби, кожен відповідає своєму типу ресурсу.

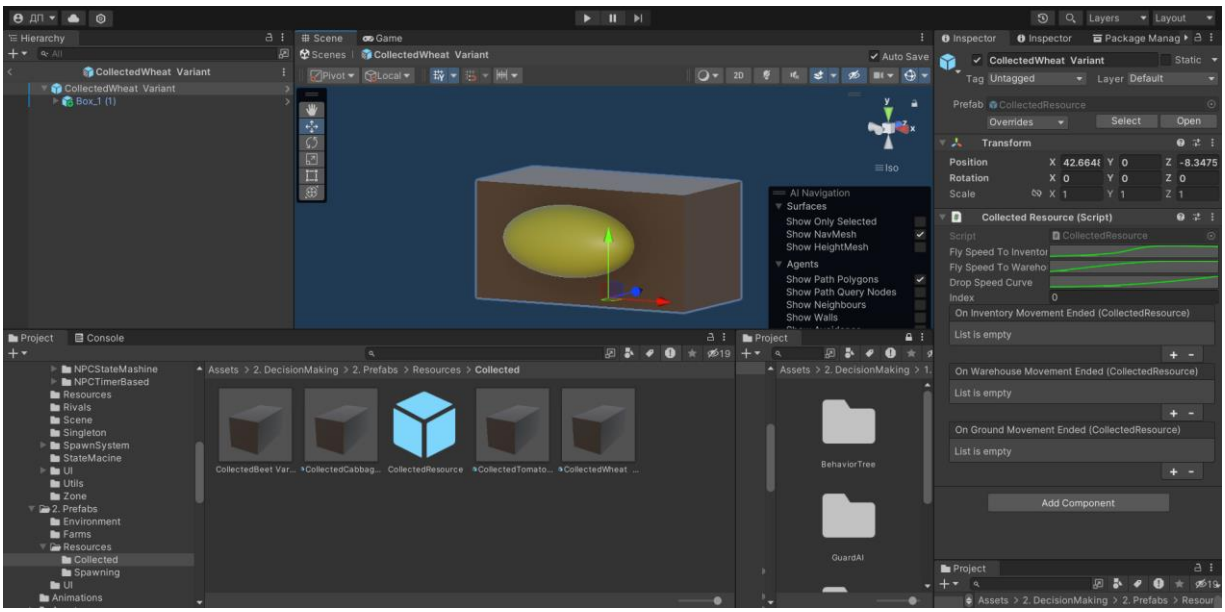


Рисунок 4.26 – Приклад «CollectedResource» (створено самостійно)

CostInResources та CostInResourcesData використовуються для визначення вартості в ресурсах для досягнення певних цілей у грі – перемоги (див. рис 4.27).

```
[CreateAssetMenu(fileName = "CostData", menuName = "SO/CostData")]
Скрипт Unity | Ссылок: 0
public class CostInResourcesData : ScriptableObject
{
    [SerializeField] List<CostInResources> costInResources;

    Ссылок: 0
    public List<CostInResources> CostInResources => costInResources;
}

[Serializable]
Ссылок: 2
public struct CostInResources
{
    public ResourceData Resource;
    public int Amount;
}
```

Рисунок 4.27 – Приклад «CostInResourcesData» (створено самостійно)

По суті, клас CostInResourcesData являє собою аналог серіалізуємого словника, де ResourcesData є ключем, а int – значенням.

Не менш важливим є об’єкт ферми. Клас Farm представляє ферму в грі, де вирощуються ресурси. Цей клас відповідає за управління вирощуванням, збором і відродженням ресурсів. Основні функції включають ініціалізацію ферми, перевірку готовності до збору, збір ресурсів та їх відродження після збору.

На старті ініціалізується менеджер спавну ресурсів

Метод TryCollectResources перевіряє, чи готові ресурси до збору і чи може інвентар гравця прийняти ресурси. Якщо умови виконані, викликає метод CollectResources. Він збирає ресурси з ферми. Спочатку впорядковує зібрані ресурси за відстанню до інвентаря гравця, потім збирає їх у список і додає до інвентаря гравця. Після цього очищає список зібраних ресурсів у менеджері спавну і викликає метод для відродження ресурсів.

Клас SpecifiedAmountOfResourcesSpawnManager управляє спавном фіксованої кількості ресурсів на певних місцях у грі. Цей клас відповідає за ініціалізацію спавну, управління циклом спавну, відродження ресурсів після збору та перевірку готовності ресурсів до збору. Він використовує фабрику для створення ресурсів.

На рисунку 4.28 показано приклад ферми, де зелений бокс це трігер зона, а жовті точки – це точки спавну ресурсів.

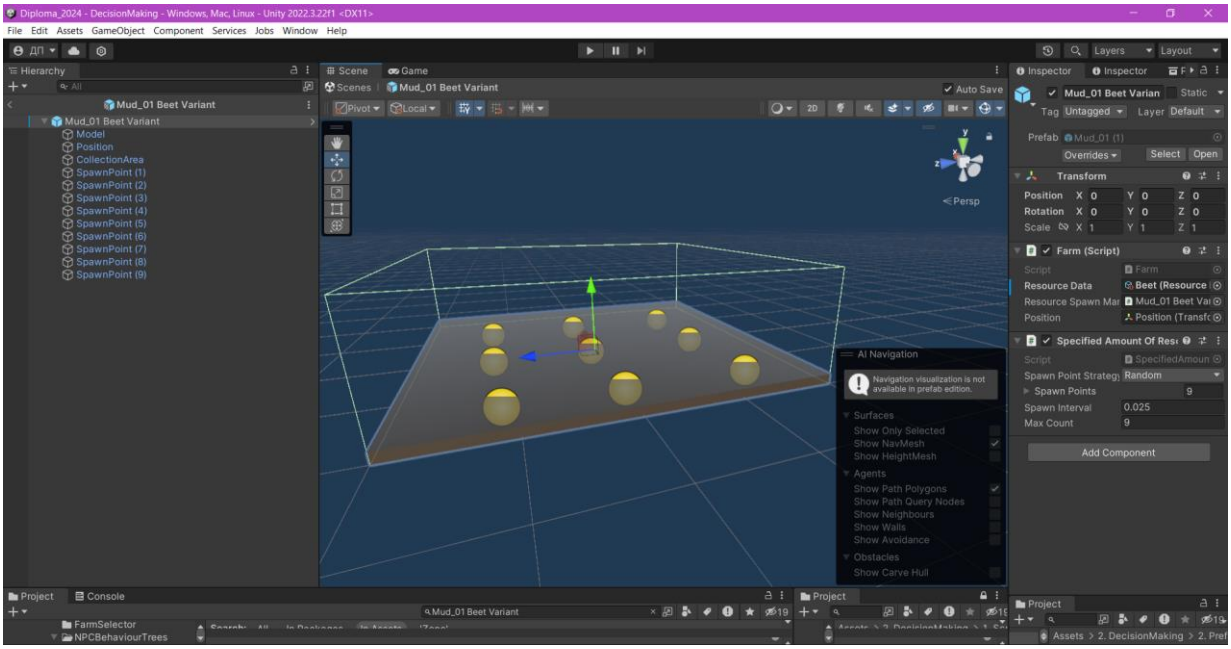


Рисунок 4.28 – Приклад ферми (створено самостійно)

Також є класи що відповідають за збір ресурсів з вказаної ферми вказаним NPC, чи навпаки, переміщення ресурсів на склад. Приклад зони збору можна побачити на рисунку вище у вигляді зеленого боксу.

Zone є базовим класом для всіх зон у грі. Він містить метод OnTriggerEnter, який викликається при вході NPC в зону.

RivalsGiveAwayZone є спадкоємцем класу Zone і представляє зону, де NPC можуть віддати зібрані ресурси на склад. При вході об'єкта в зону, він перевіряє, чи об'єкт є персонажем ICharacter, і якщо так, забирає всі ресурси з інвентаря персонажа та додає їх на склад.

Інший клас CollectionArea є спадкоємцем класу Zone і представляє зону, де NPC можуть збирати ресурси з ферми.

При вході об'єкта в зону викликається метод, який перевіряє, чи об'єкт має інвентар, і викликає метод для спроби збору ресурсів з ферми до інвентаря.

4.2.2 StateMashine

State Machine є одним із найпоширеніших та ефективних алгоритмів прийняття рішень, що використовуються для створення штучного інтелекту у відеоіграх [16]. Вона забезпечує управління поведінкою ігрових персонажів шляхом переходу між різними станами залежно від умов і подій у грі.

Основні поняття: стан (State), перехід (Transition), подія (Event).

Стан визначає поточну поведінку або дію, яку виконує персонаж у грі. Кожен стан має свої унікальні характеристики та функціональність. При вході в новий стан виконується певний набір дій, які належать до цього стану. Це може включати завантаження анімацій, налаштування параметрів або зміна характеристик персонажа та інше. У кожному кадрі гри виконується оновлення поточного стану. Це включає перевірку умов для переходу до іншого стану та виконання дій, які належать до поточного стану. При виході з поточного стану виконуються дії, які потрібні для належного завершення цього стану, наприклад, зупинка анімацій, скидання параметрів або підготовка до наступного стану.

Перехід визначає умови, за яких персонаж змінює свій поточний стан на інший. Це можуть бути як внутрішні умови, так і зовнішні події.

Умова переходу – це умова, яка повинна бути виконана для переходу з одного стану в інший. Наприклад, якщо здоров'я персонажа зменшилося до певного рівня, він може перейти з "атаки" в "відступлення".

Дії при переході – це дії, які виконуються під час переходу між станами. Це може включати зміну анімацій, оновлення параметрів або виклик певних функцій.

Подія є тригером, що викликає перехід між станами. Події можуть бути як зовнішніми так і внутрішніми.

Зовнішні події – це події, які викликані діями гравця або інших NPC. Наприклад, якщо гравець наблизився до NPC на певну відстань, це може викликати перехід NPC з "очікування" в "атака".

Внутрішні події – це події, які викликані внутрішніми умовами або таймерами. Наприклад, якщо таймер закінчився, це може викликати перехід NPC з "відпочинку" в "патрулювання".

State Machine є універсальним інструментом, який можна використовувати в багатьох сферах розробки ігор, окрім управління агентом.

State Machine може використовуватися для управління логікою гри. Наприклад, змінювати стан гри між різними рівнями або етапами. Або ж управляти об'єктами, які змінюють свій стан при взаємодії з гравцем наприклад, двері, що відкриваються при натисканні кнопки.

Машина станів широко використовується для управління анімаціями персонажів, забезпечуючи плавний перехід між різними анімаційними станами. Наприклад, алгоритм займається перемиканням між анімаціями ходьби, бігу, стрибку та стояння.

Для нашого дослідження було розроблено NPC, поведінка якого основана на роботі State Machine

Для початку був розроблений код для самої StateMachine. Цей клас є головним компонентом для реалізації станів і переходів між ними. Він

використовується для керування поведінкою об'єктів, в залежності від різних умов і станів.

Основні елементи класу:

Поточний стан – зберігає поточний стан, в якому знаходиться об'єкт.

Словник станів – зберігає всі можливі стани об'єкта у вигляді словника, де ключем є тип стану, а значенням - StateNode, що містить стан і його переходи.

Загальні переходи – зберігає переходи, які можуть відбутися з будь-якого стану незалежно від поточного.

Метод Update, що викликається кожен кадр і оновлює поточний стан об'єкта. Перевіряє, чи повинна відбутися зміна стану, і виконує відповідні дії.

Метод FixedUpdate, що викликається з фіксованою частотою, зазвичай для оновлень фізики об'єкта.

Метод SetState встановлює новий поточний стан. Викликає методи OnExit для попереднього стану і OnEnter для нового стану.

Метод ChangeState Виконує зміну стану, якщо новий стан відрізняється від поточного. Викликає методи OnExit для поточного стану і OnEnter для нового стану.

Метод GetTransition перевіряє, чи є умови для переходу в інший стан. Спочатку перевіряє загальні переходи, а потім переходи, специфічні для поточного стану.

Методи AddTransition та AddAnyTransition задають логіку переходів між станами.

Клас StateNode – внутрішній клас, який представляє вузол стану. Зберігає стан та його переходи .

Методи OnEnter, OnExit, OnUpdate та OnFixedUpdate викликаються відповідно до змін станів, забезпечуючи необхідну поведінку об'єкта в кожному стані.

Існує інтерфейс IPredicate, який визначає метод Evaluate. Цей метод повертає булеве значення, яке вказує, чи виконана умова для переходу між станами. Інші класи, такі як FuncPredicate, реалізують цей інтерфейс і дозволяють

використовувати делегати як умови для переходів між станами. Це зручно, оскільки можна легко створювати та передавати умови у вигляді функцій.

IState є інтерфейсом, який визначає основні методи, які мають реалізовувати всі стани:

- OnEnter() – викликається при вході в стан;
- OnExit() – викликається при виході зі стану;
- OnUpdate() – викликається кожен кадр для оновлення стану;
- OnFixedUpdate() – викликається з фіксованою частотою для фізичних оновлень.

ITransition є інтерфейсом, який визначає властивості та методи для переходів між станами:

Condition – умова, яка визначає, коли має відбутися перехід.

To – стан, до якого здійснюється перехід.

Було розроблено абстрактний клас NPC.cs, що реалізує інтерфейси , IStunned, ICharacter і є базовим класом для всіх NPC. На початку клас ініціалізує всі необхідні параметри, ініціалізує таймери, налаштовує NavMeshAgent, і реалізує методи, що дозволяють перевести персонажа у стан приголомшення, зупинити чи відновити можливість персонажа рухатися.

Було реалізовано агента, на основі машини станів. NPCStateMashine успадковується від NPC. Також, в якості контролера логіки персонажа додається описана раніше StateMachine. До методу ініціалізації тепер додається логіка налаштування StateMachine.

З рисунку 2.29 можна побачити створення кожного стану, в якому може перебувати персонаж та налаштування переходів між станами за допомогою stateMachine.AddTransition() або stateMachine.AddAnyTransition(). Логіка переходу описана у FuncPredicate().

```
Ссылок: 1
protected virtual void SetupStateMashine()
{
    stateMachine = new StateMachine();

    runToFarmState = new NPCRunToFarmState(this, animator, agent, farms, rivalsWarehouse);
    collectionState = new NPCCollectionState(this, animator);

    runToWarehouseState = new NPCRunToWarehouseState(this, animator, agent, rivalsWarehouse);
    giveAwayResourcesState = new NPCGiveAwayResourcesState(this, animator);

    kamikazeState = new NPCKamikazeState(this, animator, otherCharacters, agent, transform, kamikazeRadius);

    wanderState = new NPCWanderState(this, animator, agent, viewingRadius);
    stunnedState = new NPCStunnedState(this, animator);


    stateMachine.AddTransition(kamikazeState, runToFarmState, new FuncPredicate(() => kamikazeState.TargetNotFound));
    stateMachine.AddTransition(giveAwayResourcesState, kamikazeState, new FuncPredicate(() => runToWarehouseState.IsComplete && CharactersInRadius()));
    stateMachine.AddTransition(giveAwayResourcesState, runToFarmState, new FuncPredicate(() => runToWarehouseState.IsComplete && !CharactersInRadius()));
    stateMachine.AddTransition(runToWarehouseState, giveAwayResourcesState, new FuncPredicate(() => runToWarehouseState.IsComplete));
    stateMachine.AddTransition(collectionState, runToWarehouseState, new FuncPredicate(() => collectionState.IsComplete && inventory.FillPercentage >= 0.7));
    stateMachine.AddTransition(collectionState, runToFarmState, new FuncPredicate(() => collectionState.IsComplete && inventory.FillPercentage < 0.7));
    stateMachine.AddTransition(runToFarmState, collectionState, new FuncPredicate(() => runToFarmState.IsComplete));
    stateMachine.AddTransition(wanderState, runToFarmState, new FuncPredicate(() => wanderState.IsComplete && inventory.FillPercentage < 50));
    stateMachine.AddTransition(stunnedState, wanderState, new FuncPredicate(() => stunTimer.IsFinished));

    stateMachine.AddTransition(kamikazeState, stunnedState,
        new FuncPredicate(() => collisionDetector.IsDetected && lastCollisionStopwatchTimer.GetTime() > delayBeforeNewCollision));
    stateMachine.AddAnyTransition(stunnedState,
       new FuncPredicate(() => collisionDetector.IsDetected && lastCollisionStopwatchTimer.GetTime() > delayBeforeNewCollision));


    stateMachine.SetState(wanderState);
}
```

Рисунок 4.29 – налаштування StateMachine (створено самостійно)

Даний NPC має такий перелік станів:

– runToFarmState – стан, у якому NPC біжить до ферми;

– collectionState – стан збору ресурсів;

– runToWarehouseState – стан, у якому NPC біжить до складу;

– giveAwayResourcesState – стан віддачі ресурсів на склад;

– kamikazeState –стан, у якому NPC шукає ціль для зіткнення;

– wanderState – стан блукання, коли NPC досліджує околиці;

– stunnedState – стан приголомшення NPC.

Далі детальніше розглянемо кожен стан.

а) NPCBaseState є базовим класом для всіх станів NPC. Він містить основні методи та властивості, які використовуються в інших станах:

1) OnEnter() метод, який викликається під час входу в стан;
2) OnExit() метод, який викликається під час виходу зі стану;
3) OnUpdate() метод, який викликається кожного кадру, коли NPC перебуває в цьому стані;
4) OnFixedUpdate() метод, який викликається на фіксованих часових інтервалах, зазвичай використовується для фізики.

б) NPCStunnedState відповідає за стан приголомшення NPC:
   1) OnEnter() встановлює анімацію приголомшення, зупиняє рух NPC та приголомшує NPC;
   2) OnExit() відновлює рух NPC та зупиняє всі фізичні сили.

в) NPCCollectionState відповідає за стан збору ресурсів NPC:
   1) OnEnter() встановлює анімацію для стану збору і відзначає завдання як виконане;
   2) OnExit() скидає статус виконання завдання.

г) NPCGiveAwayResourcesState відповідає за стан передачі ресурсів іншим об'єктам:
   1) OnEnter() встановлює анімацію для передачі ресурсів і відзначає завдання як виконане;
   2) OnExit() скидає статус виконання завдання.

д) NPCWanderState відповідає за стан блуждання NPC:
   1) OnEnter() встановлює анімацію для блуждання та вибирає нове місце призначення;
   2) OnUpdate() перевіряє, чи досягнуто місця призначення.

е) NPCKamikazeState відповідає за стан атаки NPC у стилі "камікадзе":
   1) OnEnter() встановлює анімацію для атаки і вибирає ціль;
   2) OnUpdate() встановлює місце призначення для атаки на ціль;
   3) OnExit() зупиняє всі сили та відновлює стару дистанцію зупинки.

ж) NPCRunToFarmState відповідає за стан переміщення NPC до ферми:
   1) OnEnter() встановлює анімацію для переміщення і вибирає ферму;
   2) OnUpdate() перевіряє, чи досягнуто ферми, та чи можна збирати ресурси.

з) Клас NPCRunToWarehouseState відповідає за стан, в якому NPC переміщується до складу:
   1) OnEnter(): Викликається при вході в стан. Ініціалізує стан, встановлює анімацію та вибирає нову ціль для переміщення;

2) OnUpdate(): Викликається кожного кадру під час перебування в стані. Виконує перевірку, чи досягнута ціль.

Далі, на рисунку 4.30 описана логіка переходів з одного стану в інший.

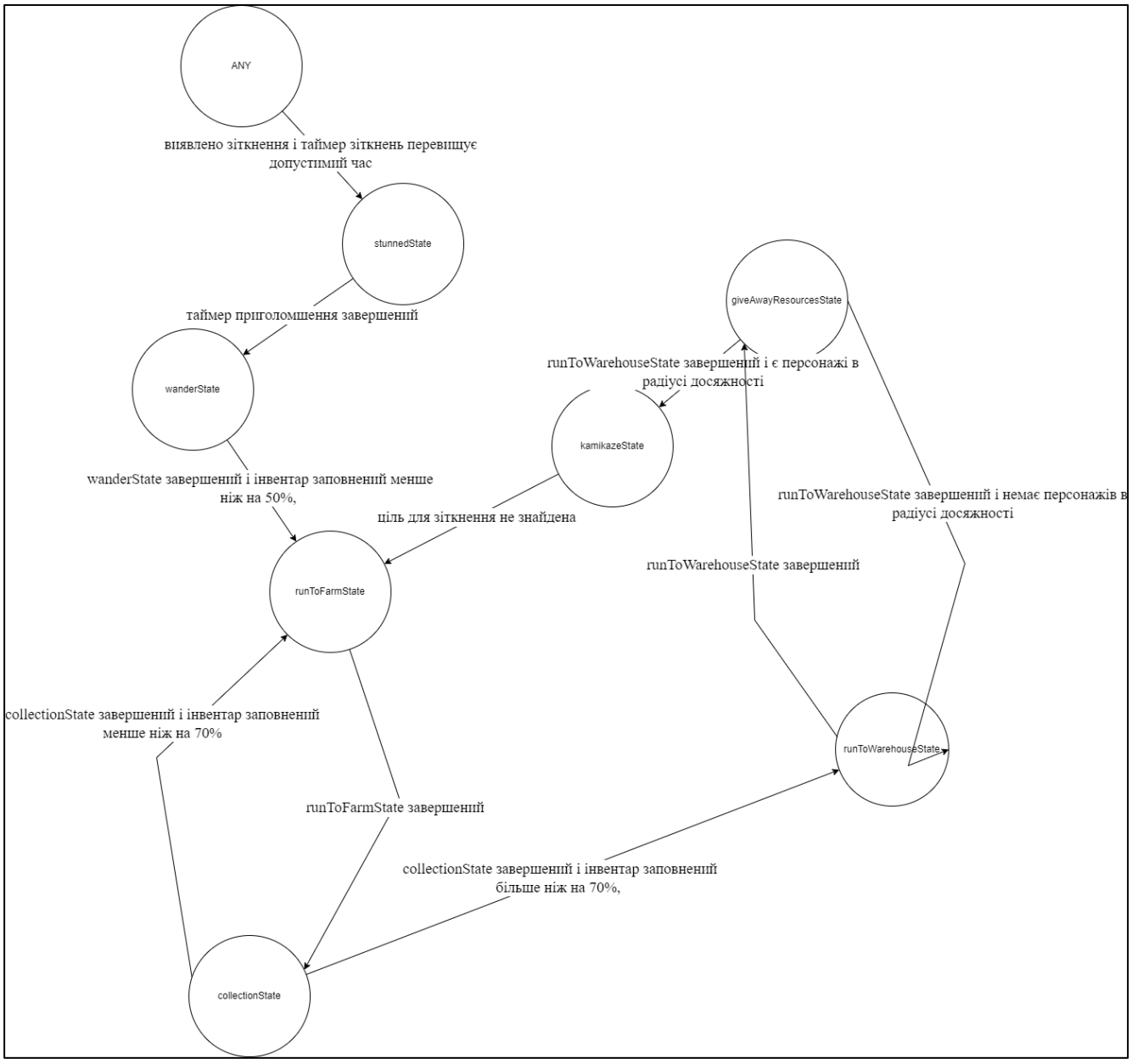


Рисунок 4.30 – схема переходів станів у StateMachine (створено самостійно)

Опис:

– якщо ціль для зіткнення не знайдена, переходить з kamikazeState до runToFarmState;

- якщо runToWarehouseState завершений і є персонажі в радіусі досяжності, переходить з giveAwayResourcesState до kamikazeState;
- якщо runToWarehouseState завершений і немає персонажів в радіусі досяжності, переходить з giveAwayResourcesState до runToFarmState;
- якщо runToWarehouseState завершений, переходить з runToWarehouseState до giveAwayResourcesState;
- якщо collectionState завершений і інвентар заповнений більше ніж на 70%, переходить з collectionState до runToWarehouseState;
- якщо collectionState завершений і інвентар заповнений менше ніж на 70%, переходить з collectionState до runToFarmState;
- якщо runToFarmState завершений, переходить з runToFarmState до collectionState;
- якщо wanderState завершений і інвентар заповнений менше ніж на 50%, переходить з wanderState до runToFarmState;
- якщо таймер приголомшення завершений, переходить з stunnedState до wanderState;
- якщо виявлено зіткнення і таймер зіткнень перевищує допустимий час, переходить до stunnedState.

4.2.3 BehaviourTrees

Дерево поведінки – це математична модель виконання плану, яка використовується в інформатиці, робототехніці, системах керування та відеоіграх [17]. Вона описує перемикання між кінцевим набором завдань модульним способом. Їх сила полягає в їхній здатності створювати дуже складні завдання, складені з простих завдань, не турбуючись про те, як прості завдання реалізуються. Дерева поведінки мають певну схожість з ієрархічними кінцевими автоматами з тією ключовою різницею, що основним будівельним блоком поведінки є завдання, а не стан. Його легкість для розуміння людиною робить дерева поведінки менш схильними до помилок і дуже популярними в спільноті розробників ігор.

Вузли можуть мати властивості, які визначають їхню поведінку

У кожного вузла є 3 стани:

– Success– вузол виконав своє завдання успішно;
– Failure– вузол не зміг виконати своє завдання;
– Running – вузол ще виконує своє завдання і потребує більше часу.

Основні поняття Behaviour Trees:

Root Node – початковий вузол, з якого починається виконання дерева.

Composite Nodes – вузли, які можуть мати одного або більше дочірніх вузлів і визначають логіку їх виконання.

Sequence – це різновид Composite Nodes.

– виконує дочірні вузли послідовно;
– якщо дочірній вузол повертає Failure, секвенсор зупиняє виконання і повертає Failure;
– якщо всі дочірні вузли повертають Success, секвенсор повертає Success;
– якщо дочірній вузол повертає Running, секвенсор зупиняє виконання і повертає Running.

Selector – це різновид Composite Nodes.

– виконує дочірні вузли послідовно;
– якщо дочірній вузол повертає Success, селектор зупиняє виконання і повертає Success;
– якщо всі дочірні вузли повертають Failure, селектор повертає Failure;
– якщо дочірній вузол повертає Running, селектор зупиняє виконання і повертає Running.

Decorator Nodes – вузли, які перевіряють умови перед виконанням дочірнього вузла. Вони можуть змінювати результати виконання дочірнього вузла або повторно його виконувати.

Leaf Nodes – вузли, які не мають дочірніх вузлів і виконують конкретні дії або перевіряють умови.

Actions – це різновид Leaf Nodes.

– виконує конкретні дії, такі як рух до цілі, атака або взаємодія з об'єктом;

– повертає Success, Failure або Running залежно від результату дії.

Conditions – це різновид Leaf Nodes.

– перевіряє певні умови, такі як рівень здоров'я, наявність ворогів або відстань до цілі;

– повертає Success, якщо умова виконується, і Failure, якщо не виконується.

Behaviour Trees працюють за принципом ієрархічної організації та послідовного виконання вузлів. Процес виконання проходить наступні етапи:

Tick – кожен кадр або з певною частотою, Behaviour Tree викликає метод Tick, який ініціює процес виконання дерева. Виконання починається з кореневого вузла і рухається вниз по дереву, перевіряючи умови та виконуючи дії.

Виконання вузлів – кожен вузол повертає один з трьох можливих станів: Success, Failure або Running. Кореневий вузол викликає свої дочірні вузли, які в свою чергу викликають свої дочірні вузли і так далі.

Для нашого дослідження було розроблено NPC, поведінка якого основана на роботі Behaviour Tree.

Для початку був розроблений код для самого дерева поведінки. Він включає в себе такі кліси: NodeState, Node, Sequence, та Selector.

NodeState являє собою перелік станів у яких може перебувати нода (див. рис 4.31).

```
namespace DecisionMaking.BehaviorTree
{
    Ссылок: 41
    public enum NodeState
    {
        RUNNING,
        SUCCESS,
        FAILURE
    }
}
```

Рисунок 4.31 – код NodeState (створено самостійно)

Node.cs являє собою абстрактний клас, що представляє вузол дерева поведінки (див. рис. 4.32).

Містить базову логіку для зберігання дочірніх нод, зберігання контексту даних та методів для їх маніпуляції.

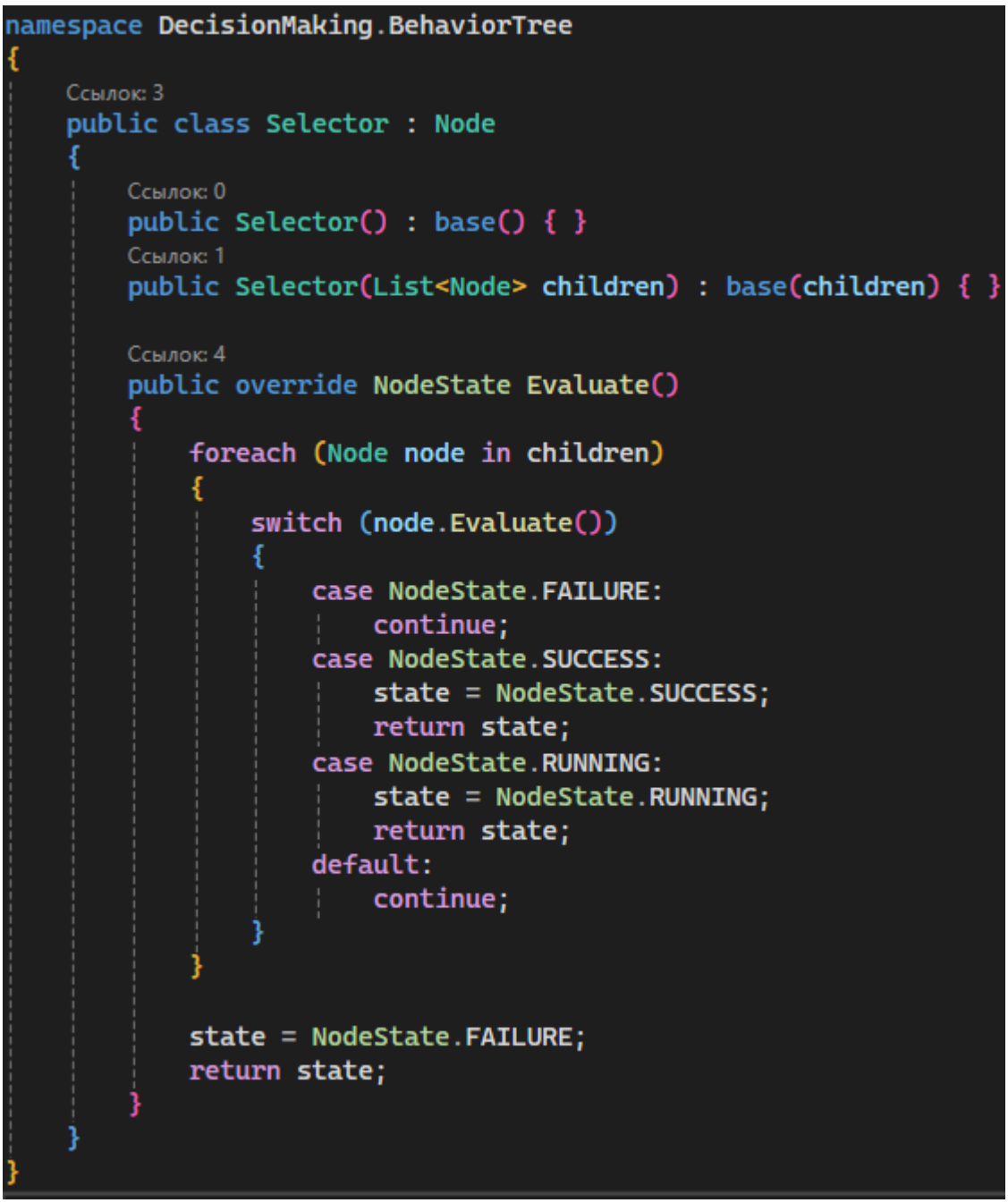
Основні методи: конструктори Attach() та Evaluate().

```
namespace DecisionMaking.BehaviorTree
{
    Ссылок: 32
    public class Node
    {
        protected static readonly int idleHash = Animator.StringToHash("Idle");
        protected static readonly int locomotionHash = Animator.StringToHash("Locomotion");
        protected static readonly int stunnedHash = Animator.StringToHash("Stunned");
        protected static readonly int KamikazeHash = Animator.StringToHash("Kamikaze");

        protected NodeState state;

        public Node parent;
        protected List<Node> children = new List<Node>();

        Ссылок: 2
        public Node()
        {
            parent = null;
        }

        Ссылок: 2
        public Node(List<Node> children)
        {
            foreach(Node child in children)
                Attach(child);
        }

        Ссылок: 1
        private void Attach(Node node)
        {
            node.parent = this;
            children.Add(node);
        }

        Ссылок: 13
        public virtual NodeState Evaluate() => NodeState.FAILURE;
    }
}
```

Рисунок 4.32 – код Node (створено самостійно)

Selector.cs реалізує композиційний вузол типу Selector (див. рис 4.33) .

Вузол проходить свої дочірні вузли і повертає SUCCESS, якщо хоча б один із них повертає SUCCESS. Якщо всі повертають FAILURE, вузол повертає FAILURE.



```
namespace DecisionMaking.BehaviorTree
{
    Ссылок: 3
    public class Selector : Node
    {
        Ссылок: 0
        public Selector() : base() { }
        Ссылок: 1
        public Selector(List<Node> children) : base(children) { }

        Ссылок: 4
        public override NodeState Evaluate()
        {
            foreach (Node node in children)
            {
                switch (node.Evaluate())
                {
                    case NodeState.FAILURE:
                        continue;
                    case NodeState.SUCCESS:
                        state = NodeState.SUCCESS;
                        return state;
                    case NodeState.RUNNING:
                        state = NodeState.RUNNING;
                        return state;
                    default:
                        continue;
                }
            }

            state = NodeState.FAILURE;
            return state;
        }
    }
}
```

Рисунок 4.33 – код Selector (створено самостійно)

Sequence.cs реалізує композиційний вузол типу " Sequence " (див. рис. 4.34).

Вузол проходить через свої дочірні вузли і повертає FAILURE, якщо хоча б один повертає FAILURE. Якщо всі повертають SUCCESS, вузол повертає SUCCESS.

```
namespace DecisionMaking.BehaviorTree
{
    Ссылок: 6
    public class Sequence : Node
    {
        Ссылок: 0
        public Sequence() : base() { }
        Ссылок: 4
        public Sequence(List<Node> children) : base(children) { }

        Ссылок: 4
        public override NodeState Evaluate()
        {
            bool anyChildIsRunning = false;

            foreach (Node node in children)
            {
                switch (node.Evaluate())
                {
                    case NodeState.FAILURE:
                        state = NodeState.FAILURE;
                        return state;
                    case NodeState.SUCCESS:
                        continue;
                    case NodeState.RUNNING:
                        anyChildIsRunning = true;
                        continue;
                    default:
                        state = NodeState.SUCCESS;
                        return state;
                }
            }

            state = anyChildIsRunning ? NodeState.RUNNING : NodeState.SUCCESS;
            return state;
        }
    }
}
```

Рисунок 4.34 – код Sequence (створено самостійно)

Було реалізовано агента, на онові дерева поведінки. NPCBehaviourTree успадковується від NPC. Також, в якості контролера логіки персонажа додається описане раніше BehaviourTree. До методу ініціалізації додається логіка налаштування StateMachine (див. рис 2.32).

На рисунку 2.35 можна побачити створення дерева із різних вузлів. Метод SetupTree() викликається при базовій ініціалізації NPC та створює деревоподібну структуру.

```
void SetupTree()
{
    tree = new Selector(new List<Node>
    {
        new Sequence(new List<Node>
        {
            new CheckStun(collisionDetector, stunTimer),
            new Stun(this, animator,  stunTimer)
        }),
        new Sequence(new List<Node>
        {
            new CheckNeedRunToFarm(this, 0.7f,farms),
            new GoToFarm(this, animator, agent, farms)
        }),
        new Sequence(new List<Node>
        {
            new CheckNeedRunToWarehouse(this, 0.7f),
            new GoToTarget(this, rivalsWarehouse.Position, animator, agent)
        }),
        new Sequence(new List<Node>
        {
            new CantDoAnything(farms),
            new GoToTarget(this, rivalsWarehouse.Position, animator, agent)
        }),
    });
}
```

Рисунок 4.35 – код створення дерева поведінки (створено самостійно)

Створюється нове дерево з кореневим вузлом типу Selector, яке містить кілька послідовностей. Кожна послідовність включає умови та дії, які визначають поведінку NPC залежно від поточної ситуації.

Далі, на рисунку 4.36 наведено схему дерева рішень яке управляє поведінкою NPC.

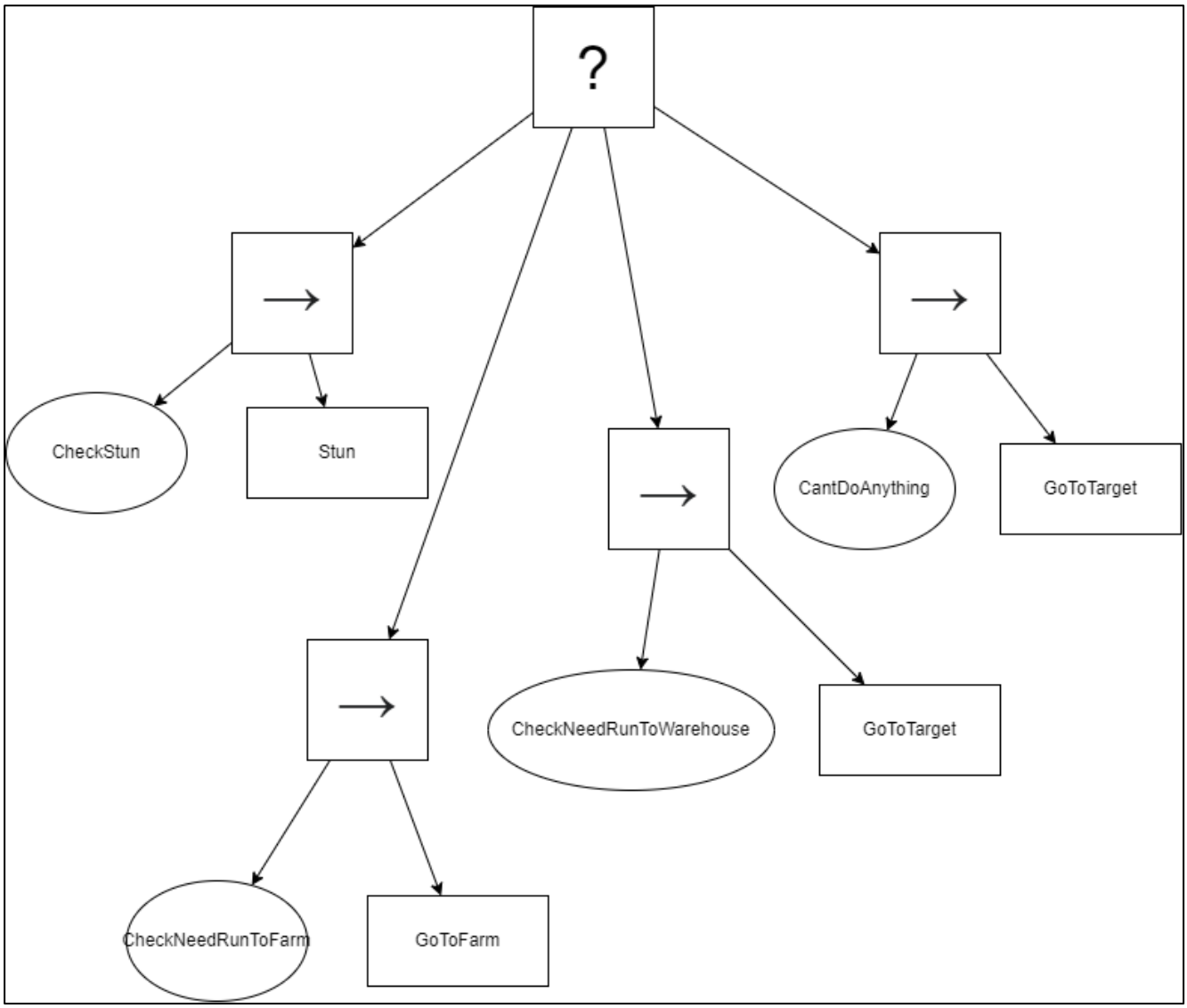


Рисунок 4.36 – Діалгарма робити створеного дерева поведінки (створено самостійно)

Перша послідовність:

– перевірка приголомшення (CheckStun), якщо таймер приголомшення активний, то стан встановлюється як успішний;
– дія приголомшення (Stun), якщо NPC не приголомшений, він переводиться в стан приголомшення та зупиняється.

Друга послідовність:

– перевірка потреби бігти на ферму (CheckNeedRunToFarm), якщо заповнення інвентаря менше ніж 70% і є ферми, де можна збирати ресурси, то стан встановлюється як успішний;

– дія бігу на ферму (GoToFarm) – NPC встановлює нову ціль для переміщення на ферму, де можна збирати ресурси.

Третя послідовність:

– перевірка потреби бігти до складу (CheckNeedRunToWarehouse), якщо заповнення інвентаря більше або дорівнює 70%, то стан встановлюється як успішний;
– дія бігу до цілі (GoToTarget) – NPC біжить до складу для передачі зібраних ресурсів.

Четверта послідовність:

– перевірка неможливості виконання завдань (CantDoAnything), якщо немає ферм, де можна збирати ресурси, то стан встановлюється як успішний;
– дія бігу до цілі (GoToTarget) – NPC переміщується по карті, намагаючись перейти в інший вузол.

Метод Update() оновлює стан таймерів та анімації кожного кадру гри, а також викликає метод Evaluate() для дерева поведінки, що дозволяє NPC обробляти поточні умови та виконувати відповідні дії.

4.2.4 Timer-Based

Алгоритми прийняття рішень на основі таймерів , або Timer-Based Decision Making є простими та ефективними методами для управління поведінкою агентів у відеоіграх. Цей підхід базується на використанні таймерів для визначення моменту виконання певних дій або переходів між станами.

Принцип роботи алгоритмів прийняття рішень на основі таймерів полягає у встановленні таймерів для різних дій або станів агента. Коли таймер закінчується, агент виконує відповідну дію або змінює свій стан. Це дозволяє створювати поведінку, яка залежить від часу і дозволяє реалізувати складні алгоритми без необхідності складних умов і логіки.

Основні елементи:

Кожен стан або дія агента має свій власний таймер. Таймери можуть бути встановлені на різні інтервали часу в залежності від потреб гри.

Стан визначає поточний стан агента. При зміні стану може змінюватися і відповідний таймер.

Основні поняття:

Встановлення таймера: Для кожного стану агента встановлюється таймер, який визначає, як довго агент перебуватиме у цьому стані.

Перевірка таймера: В кожному кадрі гри перевіряється, чи закінчився таймер для поточного стану. Якщо таймер закінчився, виконується відповідна дія або здійснюється перехід до нового стану.

Оновлення таймера: Таймери оновлюються в кожному кадрі, що дозволяє забезпечити плавну і синхронізовану поведінку агентів.

Алгоритми на основі таймерів легко реалізувати і зрозуміти. Можна легко змінювати поведінку агента, змінюючи значення таймерів. Використання таймерів дозволяє уникнути складних умов і перевірок, що робить алгоритми ефективними з точки зору обчислювальних ресурсів.

Алгоритми на основі таймерів не підходять для реалізації складної поведінки агентів. Всі дії залежать від часу, що може бути не завжди прийнятним для всіх типів ігор.

Для початку був розроблений код для самого алгоритму прийняття рішень. Для перемикання станів використовувалася вже раніше описана машина станів. Для опису станів використовувалися такі класи та інтерфейси: IState, NPCBaseState, runToFarmState, runToWarehouseState, kamikazeState, wanderState, stunnedState.

Логіка наведених вище станів описана у попередньому пункті 4.3 State Mashine.

В даному випадку, State Mashine виконує роль контейнера, що викликає у вказаного стана методи OnEnter(), OnExit(), OnUpdate(), OnFixedUpdate() в яких виконується логіка.

Було розроблено клас NPCTimerBased, що містить в собі таймери, стани та стейт машину, які управляють поведінкою NPC (див. рис. 4.37)

```
protected NPCRunToFarmState runToFarmState;
protected NPCRunToWarehouseState runToWarehouseState;
protected NPCKamikazeState kamikazeState;
protected NPCWanderState wanderState;
protected NPCStunnedState stunnedState;

List<NPCBaseState> npcBaseStates = new List<NPCBaseState>();
```

Рисунок 4.37 – Доступні стани (створено самостійно)

Потім, встановлюється початковий стан – блукати і вмикається таймер.

У методі Update() відбувається оновлення таймерів, і коли час спливає, то обирається новий стан зі списку доступних і встановлюється у стейт машину(див рис 4.38).

```
void Update()
{
    stateMachine.Update();
    stunTimer.Tick(Time.deltaTime);
    changeStateTimer.Tick(Time.deltaTime);
    lastCollisionStopwatchTimer.Tick(Time.deltaTime);
    UpdadeteAnimator();

    if(stateMachine.CurrentState == stunnedState)
    {
        if(stunnedState.IsComplete)
        {
            stateMachine.SetState(SelectNewState());
        }
    }
    else
    {
        if(changeStateTimer.IsFinished)
        {
            stateMachine.SetState(SelectNewState());
        }
    }
}
```

Рисунок 4.38 – Метод Update() (створено самостійно)

Метод SelectNewState() бере наступний стан зі списку і збільшує індекс на одиницю.

4.2.5 RandomSelect-Based

Алгоритм прийняття рішень Random Select Based базується на випадковому виборі стану з переліку можливих станів. Це підхід, який дозволяє створити різноманітну і непередбачувану поведінку агентів у відеоіграх.

Принцип роботи:

- стан визначає поточну поведінку NPC. Кожен стан має свою логіку роботи;
- коли NPC завершив поточний стан, випадковим чином вибирається новий стан зі списку доступних станів;
- після вибору нового стану, NPC переходить до нього і починає виконувати відповідні дії.

Основні поняття:

Стан визначає поточну поведінку NPC. Кожен стан має свої методи OnEnter(), OnUpdate(), OnExit() та OnFixedUpdate().

Випадковий вибір – вибір нового стану здійснюється випадковим чином зі списку можливих станів.

Було розроблено клас NPCRandomSelectBased, що успадковується від базового класу NPC і додає логіку для управління станами на основі випадкового вибору. Основні методи включають:

Initialize – ініціалізує змінні, таймери та стан.

Update – оновлює поточний стан, перевіряє таймери і здійснює переходи між станами.

SelectNewState – вибирає новий стан випадковим чином зі списку можливих станів.

Стани:

NPCStunnedState: Стан приголомшення NPC.

- OnEnter(): Ініціалізує стан приголомшення, зупиняє рух;
- OnExit(): Відновлює рух NPC.

NPCRunToWarehouseState: Стан, у якому NPC переміщується до складу.

– OnEnter(): Встановлює нову ціль і анімацію переміщення;

– OnUpdate(): Перевіряє, чи досягнута ціль.

NPCWanderState: Стан блукання NPC.

– OnEnter(): Встановлює нову ціль для блукання;

– OnUpdate(): Перевіряє, чи досягнута ціль.

NPCRunToFarmState: Стан, у якому NPC переміщується до ферми.

– OnEnter(): Вибирає ферму і встановлює ціль переміщення;

– OnUpdate(): Перевіряє, чи досягнута ферма.

NPCKamikazeState: Стан атаки у стилі "камікадзе".

– OnEnter(): Вибирає ціль і починає атаку;

– OnUpdate(): Перевіряє, чи досягнута ціль.

У методі Update() відбувається оновлення станів, і коли стан сигналізує про те, що він завершив свою роботу, обирається новий стан зі списку доступних і встановлюється у стейт машину (див рис 4.39).

```
void Update() {
    stateMachine.Update();
    stunTimer.Tick(Time.deltaTime);
    lastCollisionStopwatchTimer.Tick(Time.deltaTime);
    UpdadeteAnimator();

    if (stateMachine.CurrentState == stunnedState) {
        if (stunnedState.IsComplete) {
            stateMachine.SetState(SelectNewState());
        }
    } else {
        if (current.IsComplete) {
            stateMachine.SetState(SelectNewState());
        }
    }
}
```

Рисунок 4.39 – Метод Update() (створено самостійно)

Метод SelectNewState() бере наступний стан зі списку і збільшує індекс на одиницю.

4.2.6 Збереження та відображення результатів змагання алгоритмів

Всі дії відбуваються на заздалегідь підготованій сцені, де розміщені ферми, склад та агенти. Обраними параметрами для порівняння результатів роботи алгоритмів стали те, яке місце посів агент, та скільки залишилося зібрати ресурсів.

У верхній частині екрану показано інформацію про агентів, їз назва, колір, та кількість ресурсів які потрібно принести.

Колір агента в UI відповідає кольору волосся агента на сцені, це дозволяє стежити за агентом та його поведінкою (див. рис 4.40).

Рисунок 4.40 – UI (створено самостійно)

Після кожного раунду, ці показники відображаються на екрані та зберігаються у файл.

Результати змагання алгоритмів пошуку шляху виводяться на екран через текстові поля, які показують номер раунду, назву алгоритму, місце, скільки ресурсів залишилося зібрати та час раунду. Це проілюстровано на рисунку 4.41.

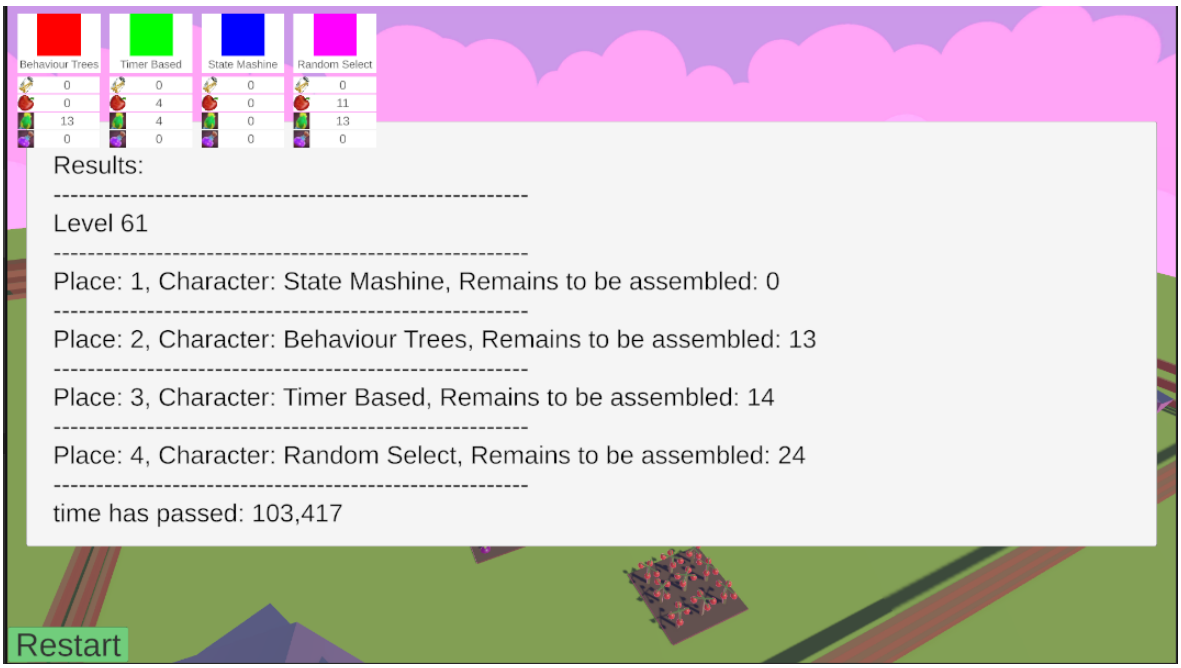


Рисунок 4.41 – Візуалізація результатів роботи алгоритмів (створено самостійно)

Результати раунду зберігаються у файл CSV для подальшого аналізу. У файл записуються такі параметри, як номер раунду, назву алгоритму, місце, скільки ресурсів залишилося зібрати та час раунду.

Наступний фрагмент коду з рисунку 4.42 відповідає за генерацію повідомлення що буде виведено на екран після завершення раунду.

```
for(int i = 0; i < sortedKeyValuePairs.Count; i++)
{
    var kvp = sortedKeyValuePairs[i];
    string str = $"Place: {i + 1}, Character: {kvp.Key.CharacterInfo.Name}, Remains to be assembled: {kvp.Value} \n";
    message += str;
    message += strLine;
    Debug.Log(str);
}
message += "time has passed: " + competitionDuration.ToString();

AlertUI.Instance.ShowAlert(message, 10f);
```

Рисунок 4.42 – Метод виводу результатів на екран (створено самостійно)

Метод WriteResultsToExcel (див рис 4.43) відповідає за збереження результатів виконання алгоритмів у файл CSV. Він перевіряє наявність файлу та, якщо файл не існує, створює його та додає заголовки. Потім записує результати.

```
ссылок: 1
private void WriteResultsToExcel(int levelNumber,
    int BTPlace, int BTRemainsToBeAssembled,
    int SMPlace, int SMRemainsToBeAssembled,
    int TBPlace, int TBRemainsToBeAssembled,
    int RSPlace, int RSRemainsToBeAssembled, int competitionDuration)
{
    string filePath = "DecisionMaking.csv";

    bool fileExists = File.Exists(filePath);

    using(StreamWriter writer = new StreamWriter(filePath, true))
    {
        if(!fileExists)
        {
            writer.WriteLine("Level," +
                "Behaviour Trees Place,Behaviour Trees Remains to be assembled," +
                "State Mashine Place,State Mashine Remains to be assembled," +
                "Timer Based Place,Timer Based Remains to be assembled," +
                "Random Select Place,Random Select Remains to be assembled," +
                "Competition duration");
        }

        writer.WriteLine($"{levelNumber}," +
            $"{BTPlace},{BTRemainsToBeAssembled}," +
            $"{SMPlace},{SMRemainsToBeAssembled}," +
            $"{TBPlace},{TBRemainsToBeAssembled}," +
            $"{RSPlace},{RSRemainsToBeAssembled}," +
            $"{competitionDuration}");
    }
}
```

Рисунок 4.43 – Метод WriteResultsToExcel (створено самостійно)

Ці фрагменти коду забезпечують виведення інформації про виконання алгоритмів на екран та збереження результатів у файл CSV для подальшого аналізу.

5 АНАЛІЗ ОТРИМАНИХ ДАНИХ

5.1 Алгоритми пошуку шляху

Було проведено низку експериментів, в яких різні алгоритми, вказані в пункті 4.1, проходили згенеровані мапи різних розмірів.

Експерименти проводилися для 40 різних сидів при розмірах сітки 10x10, 100x100, 300x300, 700x700, 1000x1000, 1500x1500, 2000x2000, 2500x2500 для кожного алгоритму.

Загалом було отримано таблицю з 2624 строками, частину можна побачити у вигляді таблиці 5.1

Таблиця 5.1 – Частина отриманої таблиці в ході експериментів (таблиця виконана самостійно)

Algorithm	Execution Time (ms)	Nodes Visited	Path Length	Memory Usage (bytes)	Grid Size	Seed
Breadth First Search	1	12	4	0	10x10	0
Dijkstra	26	10	4	0	10x10	0
AStar	0	4	4	0	10x10	0
Greedy Best First Search	0	4	4	12288	10x10	0
Depth First Search	0	60	60	16384	10x10	0
Bidirectional Search	0	20	4	12288	10x10	0
Lee Algorithm	0	12	4	12288	10x10	0
Dynamic Programming Maze	0	95	4	4096	10x10	0

5.1.1 Оцінка часу виконання алгоритму

Для нашої таблиці 5.1 знайдемо середній час роботи кожного алгоритму на кожному розмірі сітки та створимо графік для отриманих результатів (див. рис. 5.1).

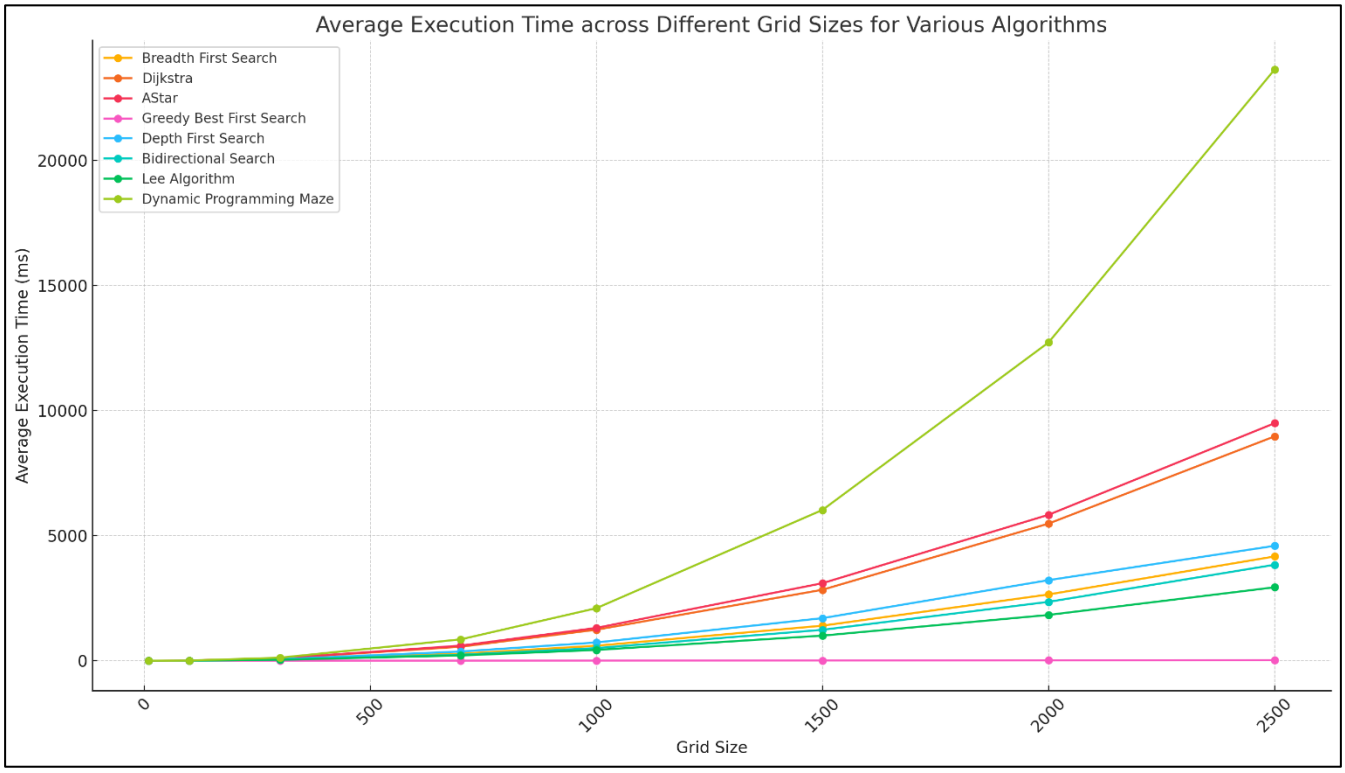


Рисунок 5.1 – Середній час виконання алгоритмів пошуку шляху на різних розмірах сітки (рисунок створено самостійно)

Цей графік дозволяє наочно порівняти продуктивність кожного алгоритму між собою.

Breadth First Search показує лінійне збільшення часу виконання зі збільшенням розміру сітки. Найкраще підходить для невеликих сіток, де можна швидко знайти шлях.

Dijkstra показує трохи кращі результати, ніж Breadth First Search на великих сітках, завдяки більш ефективному обробленню відстаней.

A* один з найефективніших алгоритмів, що використовує евристики для зменшення часу виконання. Показує стабільні результати на різних розмірах сіток, демонструючи високу продуктивність.

Greedy Best First Search використовує жадібний підхід, що іноді призводить до швидшого знаходження шляху, але не завжди оптимального. Показує хороші результати на малих, середніх та великих сітках. Графік алгоритму майже лінійний (див. рис 5.2).

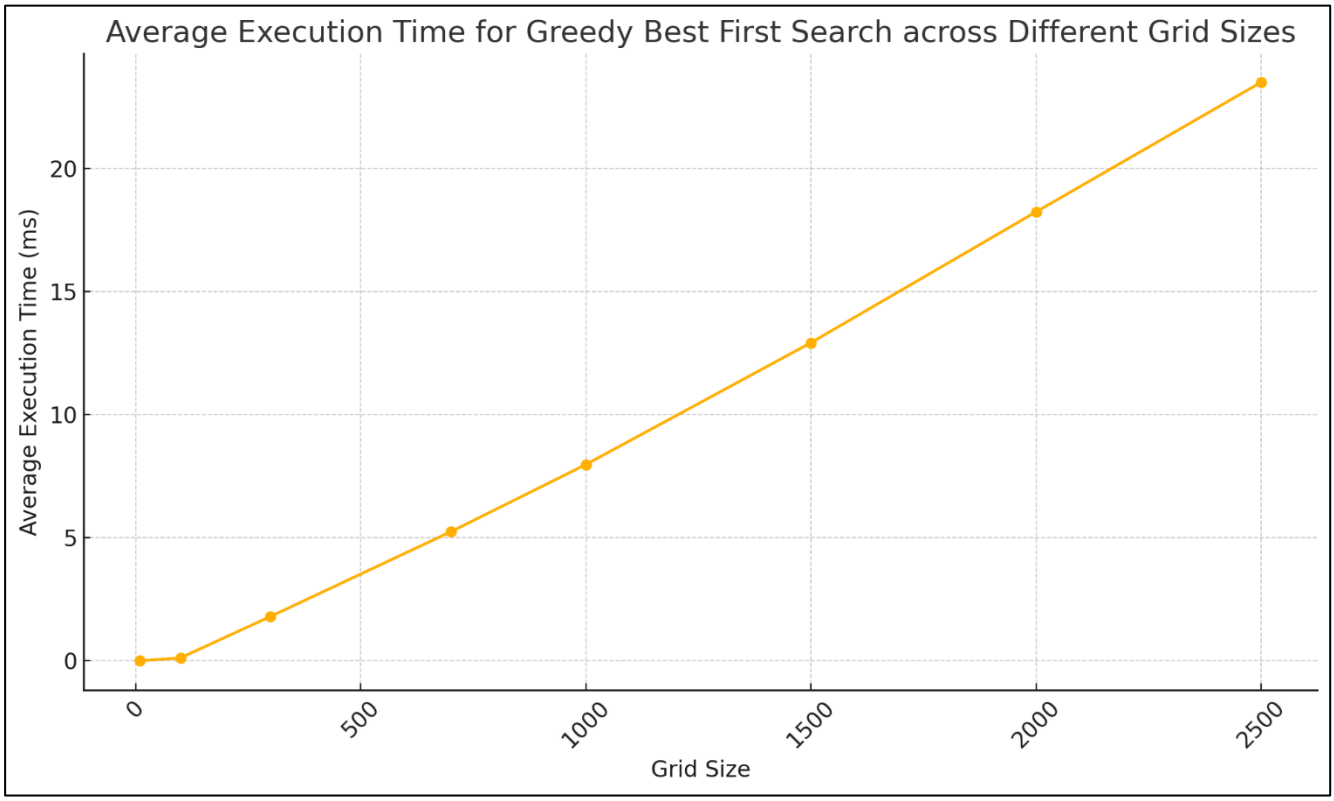


Рисунок 5.2 – Середній час виконання Greedy Best First Search (рисунок створено самостійно)

Depth First Search показує значне збільшення часу виконання на великих сітках через глибокий пошук у всіх напрямках. Найгірше підходить для великих сіток, де потрібні оптимальні та швидкі результати.

Bidirectional Search поєднує два одночасні пошуки з початку та кінця, що скорочує час виконання. Показує високу ефективність на великих сітках, хоча трохи поступається A*.

Lee Algorithm використовується переважно в сценаріях, де важлива оптимальність, але час виконання може бути довшим через ретельний пошук. Показує середні результати, краще підходить для середніх розмірів сіток.

Dynamic Programming Maze використовує динамічне програмування, що забезпечує високу точність, але може бути повільним на великих сітках. Показує найкращі результати на малих та середніх сітках (див рис 5.3).

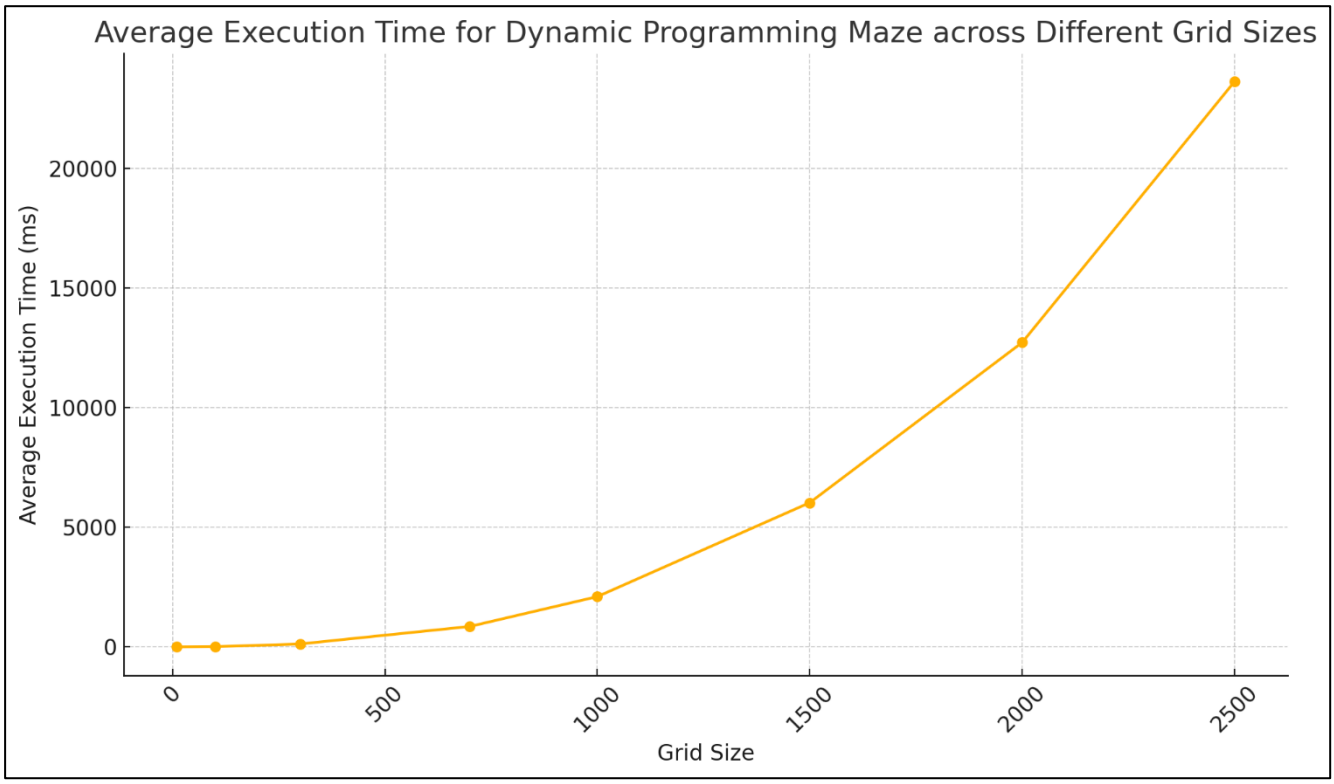


Рисунок 5.3 – Середній час виконання Dynamic Programming Maze (рисунок створено самостійно)

Найгірший графік демонструє саме цей алгоритм. Він демонструє найгірший ріст показників затраченого часу від розмірів мапи.

5.1.2 Оцінка часу виконання алгоритму

Тепер розглянемо алгоритми на кількість затраченої пам’яті. Зробимо аналогічним методом, знайдемо середні значення для кожного алгоритму на всіх сідах на кожний розмір мапи та побудуємо спільний графік (див. рис 5.4).

Цей аналіз допоможе вибрати відповідний алгоритм для їхніх потреб залежно від розміру сітки та вимог до продуктивності.

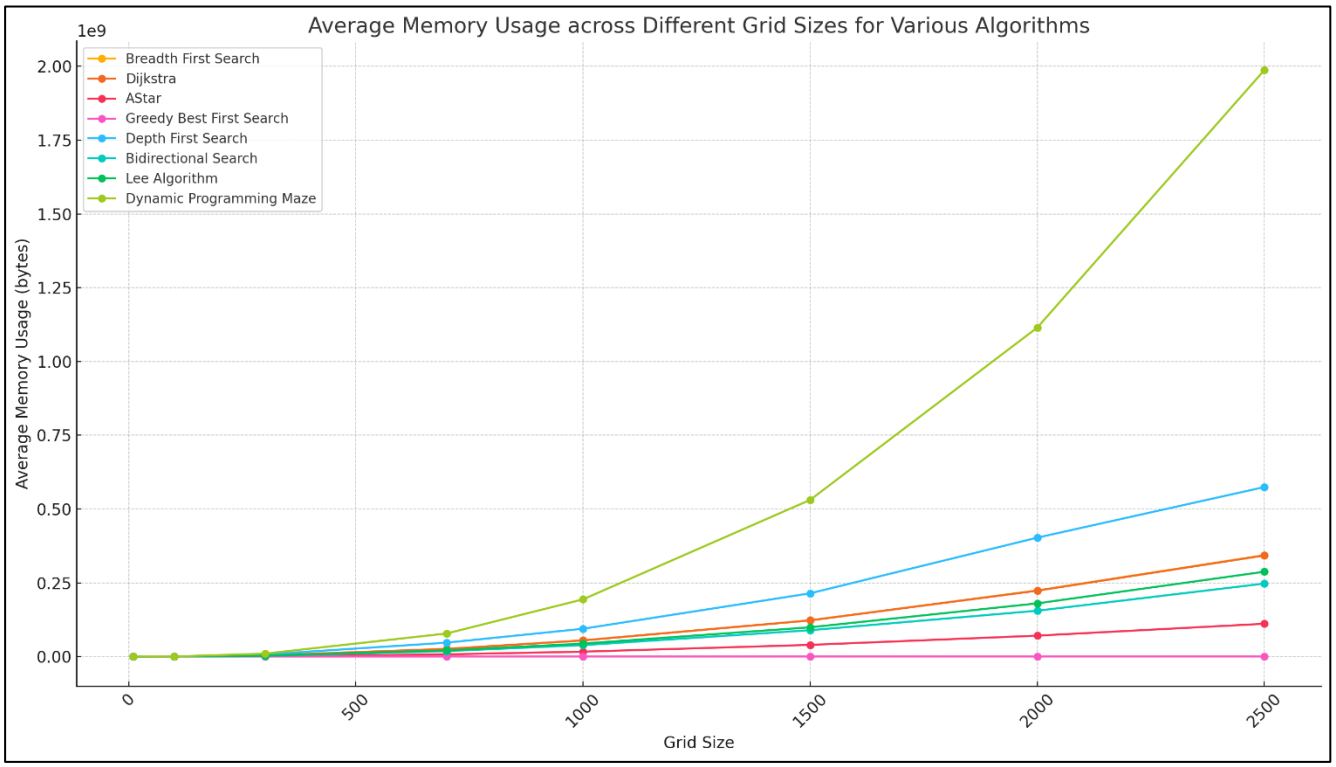


Рисунок 5.4 – Середній час виконання Dynamic Programming Maze (рисунок створено самостійно)

Можна оцінити результати наступним чином:

Breadth First Search використання пам'яті збільшується зі збільшенням розміру сітки. Найкраще підходить для невеликих сіток, оскільки споживає менше пам'яті порівняно з іншими алгоритмами на великих сітках.

Dijkstra вВикористовує більше пам'яті ніж Breadth First Search, особливо на великих сітках, через зберігання додаткової інформації про відстані. Підходить для сценаріїв, де важлива оптимальність шляху.

A* дуже ефективний у використанні пам'яті завдяки евристикам, що обмежують кількість вузлів, які потрібно зберігати. Показує стабільні результати на різних розмірах сіток.

Greedy Best First Search використовує менше пам'яті порівняно з іншими алгоритмами, оскільки зберігає лише вузли, що здаються найближчими до цілі.

Показує хороші результати на малих, середніх та великих сітках. І знову залежність використаної пам’яті вод розміру ситки прямує до лінійної (див. рис. 5.5).

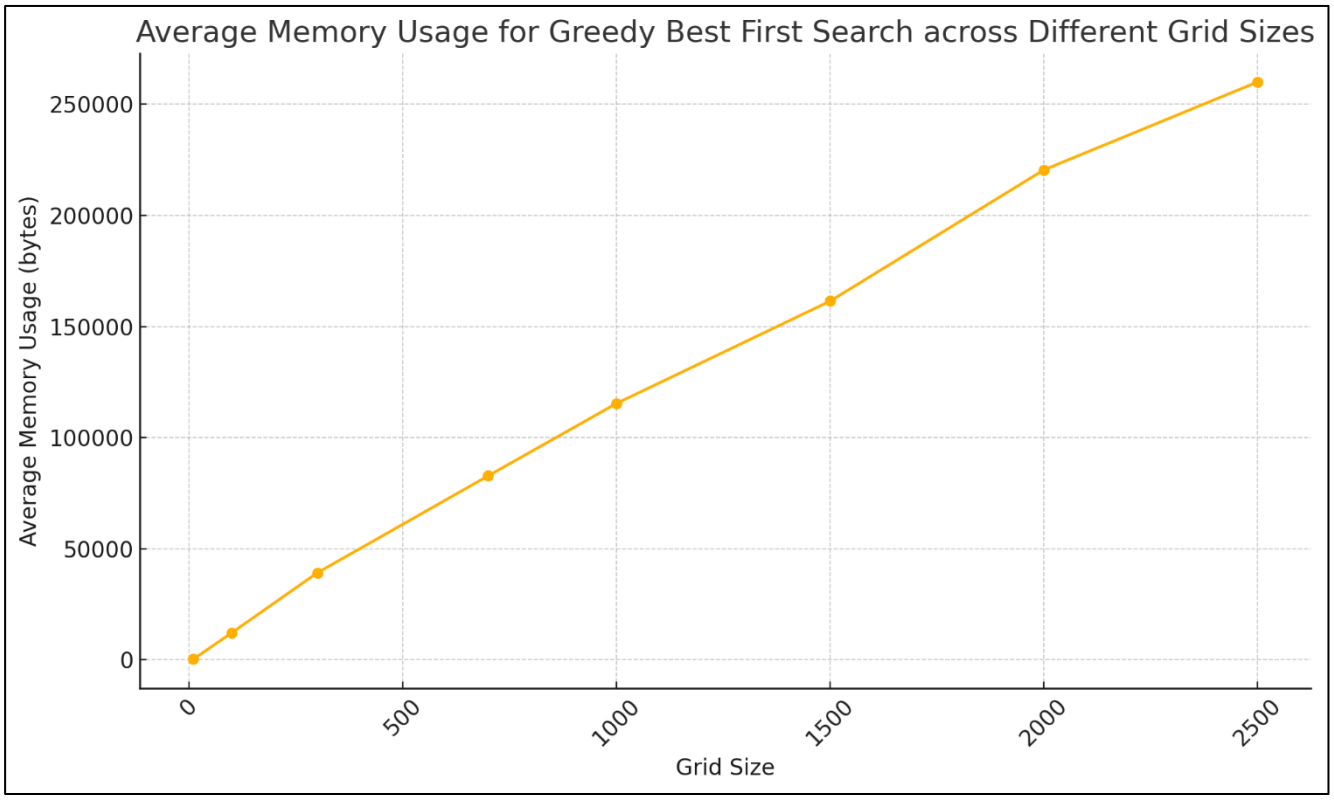


Рисунок 5.5 – Середнє значення використання пам'яті Greedy Best First Search (рисунок створено самостійно)

Depth First Search використовує менше пам'яті на невеликих сітках, але споживає більше пам'яті на великих сітках через необхідність зберігання всіх відвіданих вузлів. Не є оптимальним для великих сіток.

Bidirectional Search ефективний у використанні пам'яті, оскільки об'єднує два одночасні пошуки з початку та кінця. Показує високу ефективність на великих сітках.

Lee Algorithm використовує багато пам'яті на великих сітках через необхідність зберігання інформації про кожний вузол. Найкраще підходить для малих та середніх розмірів сіток.

Dynamic Programming Maze споживає значну кількість пам'яті через використання динамічного програмування. Показує кращі результати на малих та середніх сітках, ніж на великих сітках (див. рис. 5.6).

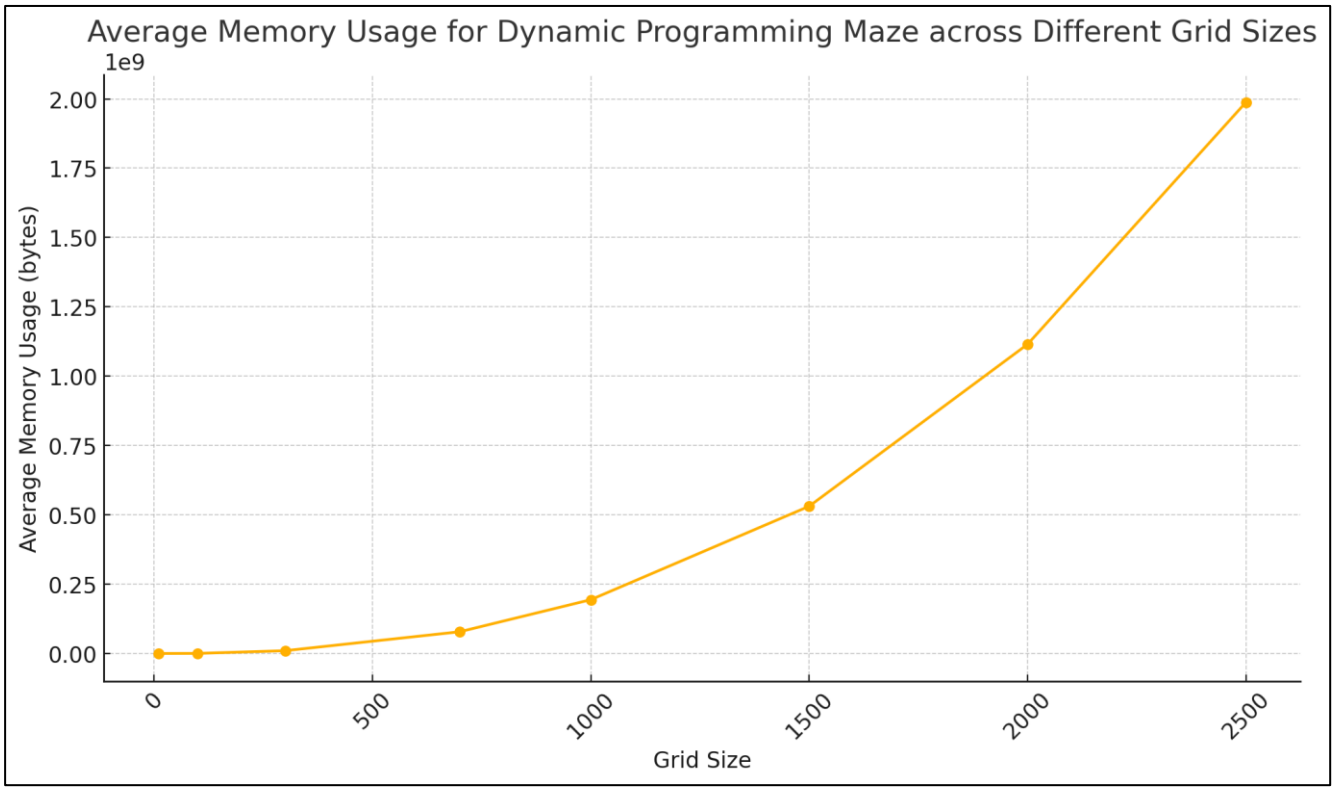


Рисунок 5.6 – Середнє значення використання пам'яті Dynamic Programming Maze (рисунок створено самостійно)

І знову бачимо, що алгоритм Dynamic Programming Maze показує найгірші експоненціальний ріст використаної пом’яті в залежності від розміру сітки.

5.1.3 Оцінка пошуку оптимального шляху

Тепер потрібно визначити які з алгоритмів шукають найкоротший шлях, які не завжди знаходять найкоротший.

Візьмемо конкретні seed, наприклад 10 - 13, для нього візьмемо сітку розміру 2500x2500 та визначимо довжину шляху для кожного алгоритму. Результати можна побачити в таблиці 5.2.

Таблиця 5.2 – Довжина шляху, який знайшов алгоритм на мапі розміру 2500x2500 (таблиця виконана самостійно)

Algorithm	Seed 10	Seed 11	Seed 12	Seed 13
Breadth First Search	630	1789	2647	1591
Dijkstra	630	1789	2647	1591
AStar	630	1789	2647	1591
Greedy Best First Search	634	1807	2669	1591
Depth First Search	3346214	3160993	1160147	4254261
Bidirectional Search	630	1789	2647	1591
Lee Algorithm	630	1789	2647	1591
Dynamic Programming Maze	630	1789	2647	1591

З таблиці видно, що алгоритми Breadth First Search, Dijkstra, AStar, Bidirectional Search, Lee Algorithm, Dynamic Programming Maze знаходят один і той самий шлях, незалежно від сіду, а це свідчить про те, що вони знаходять найкоротший шлях.

Потім іде Greedy Best First Search, за рахунок жадібного підходу пошук призводить до трохи довших шляхів у порівнянні з іншими алгоритмами, але дивлячись на попередні показники часу виконання та обсягів необхідної пам’яті це не суттєве відхилення.

Найгірший результат показує Depth First Search. Це помітно особливо на великих сітках. Такі результати отримуються через те, що алгоритм робить глибокий пошук у всіх напрямках, а так як мапа – це сітка, де кожна вершина пов’язана з сусідніми чотирма, то і отримуємо такий результат

Знову візьмемо конкретні seed, 10 - 13,але вже на сітці розміру 100x100 та визначимо довжину шляху для кожного алгоритму. Результати можна побачити в таблиці 5.3.

Таблиця 5.3 – Довжина шляху, який знайшов алгоритм на мапі розміру 100x100 (таблиця виконана самостійно)

Algorithm	Seed 10	Seed 11	Seed 12	Seed 13
Breadth First Search	27	95	107	65
Dijkstra	27	97	107	65
AStar	27	97	107	65
Greedy Best First Search	27	97	107	65
Depth First Search	5685	3379	2477	6803
Bidirectional Search	27	95	107	65
Lee Algorithm	27	95	107	65
Dynamic Programming Maze	27	97	107	65

Проаналізувавши дані з таблиці можна сказати наступне:

Breadth First Search довжина шляху стабільна та коротка, що свідчить про ефективність алгоритму для пошуку найкоротшого шляху.

Dijkstra аналогічний результату Breadth First Search, що демонструє здатність алгоритму знаходити оптимальний шлях у графі з рівними вагами.

A* показує такі ж результати, що і Dijkstra, оскільки цей алгоритм використовує евристику для оптимізації пошуку шляху.

Greedy Best First Search довжина шляху схожа на результати Breadth First Search, але цей алгоритм не завжди гарантує найкоротший шлях через жадібний підхід.

Depth First Search значно довші шляхи, що свідчить про неефективність алгоритму для знаходження оптимальних шляхів, особливо на великих сітках.

Bidirectional Search показує такі ж результати, як і Breadth First Search, завдяки ефективності одночасного пошуку з початкової та кінцевої точок.

Lee Algorithm аналогічний результату Breadth First Search, підтверджуючи його здатність знаходити оптимальний шлях.

Dynamic Programming Maze показує такі ж результати, як і Dijkstra та A*, забезпечуючи точний і оптимальний шлях.

5.1.4 Підсумок

Середній час виконання алгоритмів було проаналізовано та представлено на графіку (рис. 5.1).

Алгоритм Breadth First Search показує лінійне збільшення часу виконання зі збільшенням розміру сітки. Dijkstra трохи ефективніший на великих сітках. A* використовує евристики для оптимізації часу виконання. Greedy Best First Search має майже лінійний графік часу виконання (рис. 5.2). Depth First Search показує значне збільшення часу виконання на великих сітках. Bidirectional Search поєднує два одночасні пошуки, що скорочує час виконання. Lee Algorithm показує середні результати на великих сітках. Dynamic Programming Maze найкраще підходить для малих та середніх сіток (рис. 5.4).

Аналіз використання пам'яті показує, що Breadth First Search найкраще підходить для невеликих сіток. Dijkstra використовує більше пам'яті на великих сітках. A* ефективний у використанні пам'яті завдяки евристикам. Greedy Best First Search показує хороші результати на різних розмірах сіток (рис. 5.5). Depth First Search використовує більше пам'яті на великих сітках. Bidirectional Search ефективний у використанні пам'яті на великих сітках. Lee Algorithm найкраще

підходить для малих та середніх розмірів сіток. Dynamic Programming Maze споживає багато пам'яті на великих сітках (рис. 5.6).

Алгоритми Breadth First Search, Dijkstra, AStar, Bidirectional Search, Lee Algorithm та Dynamic Programming Maze знаходять один і той самий шлях, незалежно від сіду чи розміру карти, що свідчить про їх здатність знаходити найкоротший шлях. Greedy Best First Search знаходить трохи довші шляхи через жадібний підхід. Depth First Search показує значно довші шляхи, особливо на великих сітках, через глибокий пошук у всіх напрямках.

5.2 Алгоритми прийняття рішень

Було проведено низку змагань між NPC, в яких різні алгоритми, вказані в пункті 4.2, суперничали за ресурси. Експерименти проводилися 54 рази при двох різних необхідних до збору кількостях ресурсів, це по 13 кожного виду і по 20 кожного виду. Загалом було отримано таблицю з 54 строками, частину можна побачити у вигляді таблиці 5.4

Таблиця 5.4 – Частина отриманої таблиці в ході експериментів (таблиця виконана самостійно)

Level	Behaviour Trees Place	Behaviour Trees Remains to be assembled	State Mashine Place	State Mashine Remains to be assembled	Timer Based Place	Timer Based Remains to be assembled	Random Select Place	Random Select Remains to be assembled	Competition duration
1	2	13	1	0	4	30	3	15	41
2	2	13	1	0	4	36	3	15	50
3	1	0	2	8	4	20	3	15	77
4	1	0	2	2	4	16	3	8	52
5	2	2	1	0	4	19	3	11	46

Для початку створимо діаграму, що покаже, скільки разів який алгоритм зайняв певне місце в змаганні. Це дасть змогу визначити який алгоритм прийняття рішень, для конкретної задачі, а тобто збір ресурсів на швидкість, підходить більше.

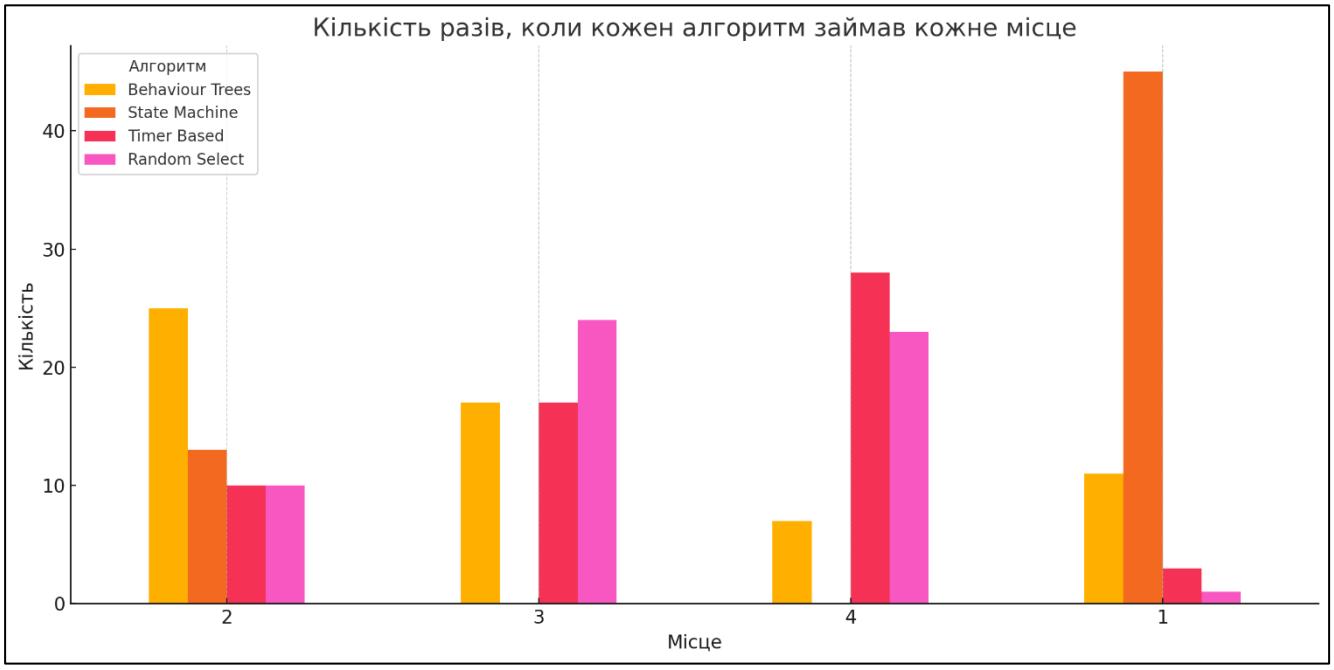


Рисунок 5.7 – Кількість разів, коли кожен алгоритм займав кожне місце (рисунок створено самостійно)

З діаграми бачимо, що Behaviour Tree найчастіше займає друге, трохи рідше перше місце, помірну кількість третіх місць і дуже рідко – четверте.

State Machine найчастіше займає перше місце, значну кількість других місць, майже ніколи не займає третє та четверте місця.

Timer Based переважно займає четверте місце, помірну кількість третіх місць, дуже рідко займає друге місце, ніколи не займає перше місце.

Random Select найчастіше займає друге та третє місця, має помірну кількість четвертих місць, рідко займає перше місце.

Це дозволяє зробити висновок, що State Machine та Behaviour Trees є більш ефективними для даного завдання порівняно з іншими алгоритмами, тоді як Timer Based алгоритм виявився найменш ефективним.

Далі розглянемо у відсотках кількість ресурсів, що залишилося зібрати кожному NPC. Зробимо таблицю 5.5, де покажемо середній відсоток ресурсів які залишилося зібрати.

Таблиця 5.5 – Середній відсоток ресурсів які залишилося зібрати (таблиця виконана самостійно)

Algorithm	Mean Remains to be Assembled (52), %	Mean Remains to be Assembled (80), %
Behaviour Trees	20.192	22.884
State Machine	3.159	0.625
Timer Based	42.032	30.432
Random Select	35.508	31.394

З таблиці видно, що:

Для Behaviour Trees на рівнях з потребою зібрати 52, чи 80 ресурсів середній залишок складає приблизно 20%. Це свідчить про те, що алгоритм демонструє стабільну ефективність незалежно від кількості необхідних для збору ресурсів

Для State Machine на рівнях з потребою зібрати 52, чи 80 ресурсів середній залишок складає 0.6-3%. Це показує, що алгоритм State Machine є найефективнішим серед усіх розглянутих, справляючись як з меншою, так і з більшою кількістю ресурсів.

Для Timer Based на рівнях з потребою зібрати 52 ресурси середній залишок складає приблизно 42%. На рівнях з потребою зібрати 80 ресурсів середній

залишок складає приблизно 30%. Це вказує на те, що алгоритм Timer Based покращує ефективність на рівнях з більшою кількістю ресурсів.

Для Random Select на рівнях з потребою зібрати 52 ресурси середній залишок складає приблизно 36%. На рівнях з потребою зібрати 80 ресурсів середній залишок складає приблизно 31%. Алгоритм Random Select також не є дуже ефективним, але його результати є трохи кращими на рівнях з більшою кількістю ресурсів.

Для більшої зручності побудуємо графік (див рис 5.8)

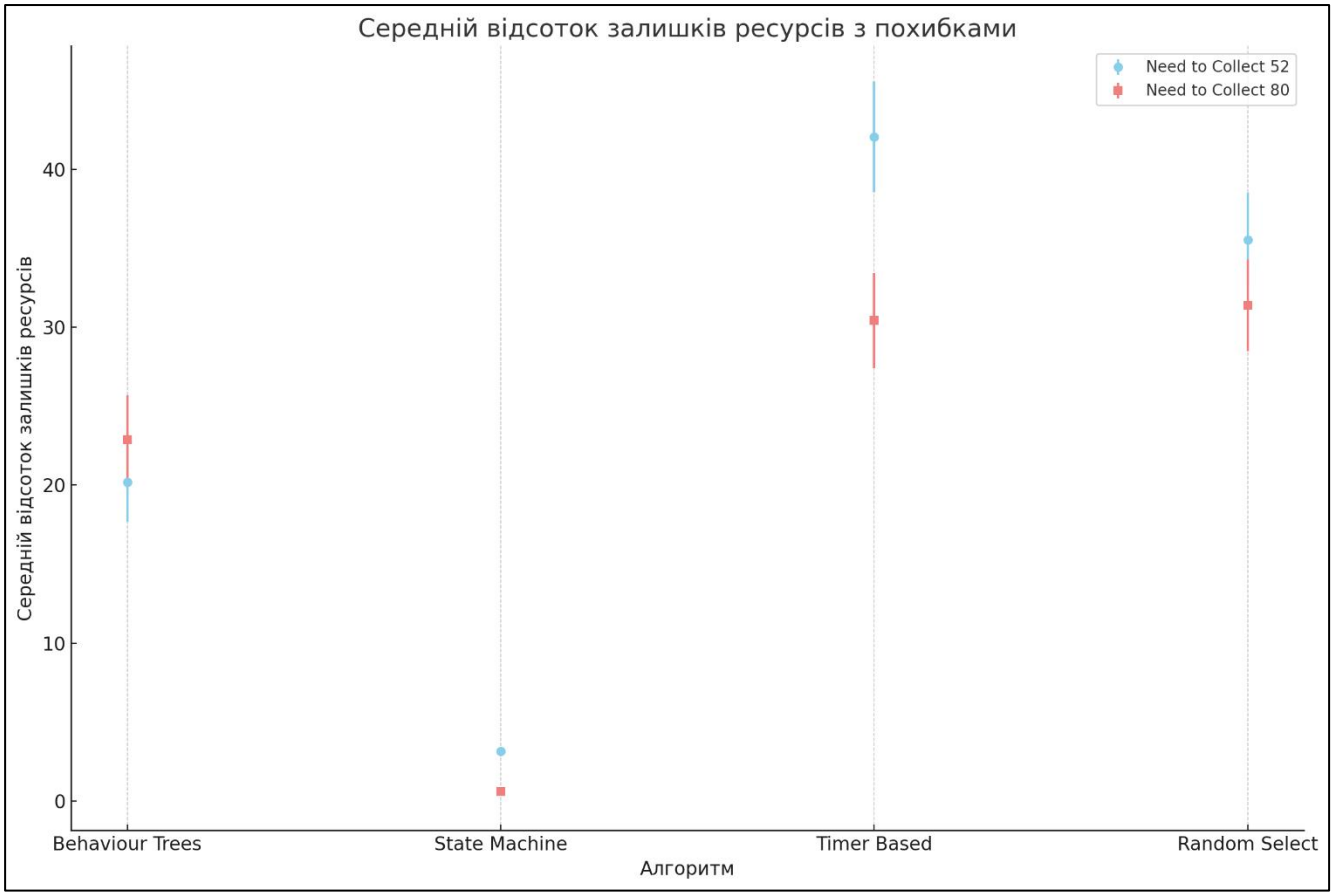


Рисунок 5.8 – Середній відсоток ресурсів які залишилося зібрати (рисунок створено самостійно)

Графік показує середній відсоток залишків ресурсів для кожного алгоритму на рівнях з потребою зібрати 52 та 80 ресурсів.

Отже, State Machine є найефективнішим алгоритмом для збору ресурсів незалежно від кількості ресурсів, які потрібно зібрати.

Behaviour Trees також є досить ефективним, але показує трохи гірші результати при більшій кількості ресурсів.

Timer Based і Random Select є менш ефективними алгоритмами, але вони демонструють деяке покращення на рівнях з більшою кількістю ресурсів.

ВИСНОВКИ

Проведено порівняння різних алгоритмів пошуку шляху, включаючи A*, Дейкстра, BFS, та інші. Результати експериментів показали, що алгоритм A* демонструє найкращу продуктивність в умовах великих та складних середовищ. Алгоритм Дейкстра не сильно програє забезпечує найкоротший шлях, але менш ефективний у великих середовищах. Greedy Best First Search показав хороші результати швидкості та по затратам пам’яті на всіх розмірах сіток, проте не завжди знаходить оптимальний шлях через жадібний підхід

Алгоритм A* виявився найбільш ефективним для складних середовищ з великою кількістю перешкод. Він поєднує швидкість та точність, що робить його придатним для більшості ігрових сценаріїв. Алгоритм BFS забезпечує гарантоване знаходження найкоротшого шляху, але менш ефективний через високі вимоги до пам'яті. Алгоритм Greedy Best First Search обирає вершини на основі евристичної функції, що оцінює вартість до цільової вершини. Це робить алгоритм швидким, але він не гарантує знаходження найкоротшого шляху.

Для покращення продуктивності алгоритмів використано різні підходи до оптимізації, включаючи зменшення кількості обчислень та використання евристичних функцій. Це дозволяє зменшити час виконання та використання пам'яті.

Також у роботі було розглянуто різні підходи до створення агентів на основі алгоритмів прийняття рішень, таких як Behaviour Trees, State Machines, Timer Based та Random Select.

Поведінкові дерева показали високу гнучкість та ефективність у створенні складних моделей поведінки агентів. State Machine виявилася простою у реалізації, але з менш гнучкою структурою.

Behaviour Trees продемонстрували високу адаптивність до змін у середовищі та можливість створення складних поведінкових моделей. State Machines показали себе як прості та надійні інструменти для реалізації базових поведінкових патернів.

Для динамічних середовищ з частими змінами станів найкраще підходять Behaviour Trees. Для статичних середовищ із прогнозованими подіями оптимальними є State Machines та Timer Based підходи. Random Select може бути використаний для створення непередбачуваних поведінкових патернів.

Наукова новизна дослідження полягає у комплексному аналізі та порівнянні різних підходів до створення ігрового штучного інтелекту. Результати дослідження можуть бути основою для подальших наукових робіт у цій галузі.

Практичне значення роботи полягає у створенні практичних рекомендацій для розробників, що дозволить оптимізувати існуючі ігрові проекти та розробляти нові з високим рівнем інтерактивності та реалізму.