Полтавський національний технічний університет імені Юрія Кондратюка

Навчально-науковий інститут інформаційних технологій і робототехніки

Кафедра комп’ютерної інженерії

**Пояснювальна записка**

до дипломної роботи

бакалавра

(ступінь вищої освіти)

на тему:

**«**Розробка апаратно-програмних засобів навігаційної системи по університету з підтримкою АР(AR) технологій на базі Unity3D, C#**»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Виконав: студент | | 4 | курсу, групи | 402-ТК |
| напряму підготовки (спеціальності) | | | | |
| 123 Комп’ютерна інженерія | | | | |
| (шифр і назва напряму підготовки, спеціальності) | | | | |
| Мовчан В.С. | | | | |
| (прізвище та ініціали) | | | | |
| Керівник | Черницька І.О. | | | |
|  | (прізвище та ініціали) | | | |
| Рецензент | Гененко Т.А. | | | |
|  | (прізвище та ініціали) | | | |

м. Полтава – 2021

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**«ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»**

**НАВЧАЛЬНО НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ**

**КАФЕДРА КОМП’ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ**

**КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА**

**спеціальність 123 «Комп’ютерна інженерія»**

**на тему**

**«Розробка апаратно-програмних засобів навігаційної системи по університету з підтримкою АР(AR) технологій на базі Unity3D, C#»**

**Студента групи 402-ТК Мовчана Владислава Сергійовича**

Керівник роботи

старший викладач   
Черницька І.О.

Завідувач кафедри

кандидат технічних наук,

доцент Головко Г.В.

Полтава – 2021

РЕФЕРАТ

Склад дипломної роботи:

* 62 сторінок;
* 28 рисунків;
* список використаної літератури з 25 позицій;
* 1 додаток.

В роботі розібрано, та описано основні принципи розробки мобільних додатків/ігор, вимоги до структуризації проєкту. Детально розібрано всі частовикористовувані алгоритми пошуку дляху, можливості їх пришвидчення, та покращення показників, на основі попередньо описаних данних буо створено мобільний доодаток “NUPP Navigator”, який, використовуючи алгоритми, автоматично знаходить шлях від входу, до потрібної аудиторії.

Об’єктом дослідження є процес розродки мобільного Android додатку на базі Unity3D, з використання мови програмування С#.

Предметом дослідження є методи автоматичноо пошуту шляху у 2D просторі, за допомоги алгоритму А\*

Метою дослідження є оптимізація втраценого часу стулентом, при пошуку необхідної аудиторії, шляхом використання розробленого додатку для

Ключові слова: Unity3D, C#, Pathfinding, Navigator, Android, A\* algorithm, Dijkstra’s algorithm.

ABSTRACT

Thesis composition:

• 62 pages;

• 28 drawings;

• list of used literature from 25 positions;

• 1 application.

The paper develops and describes the basic principles of mobile application / game development, requirements for project structuring. All frequently used search algorithms for search, possibilities of their acceleration, improvement of indicators are analyzed in detail, on the basis of previously described data the mobile application "NUPP Navigator" is created, which, using algorithms, automatically finds a way from an entrance to the necessary audience.

The object of research is the process of developing an Android mobile application based on Unity3D, using the C # programming language.

The subject of research is the methods of automatic path search in 2D space, to help the algorithm A \*

The purpose of the study is to optimize the lost time of the student, while finding the required audience, using the developed additional for

Keywords: Unity3D, C #, Pathfinding, Navigator, Android, A \* algorithm, Dijkstra algorithm.

ЗМІСТ

РЕФЕРАТ…………………………………………………………………………….3

ЗМІСТ………………………………………………………………………………...5

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕННЬ, СКОРОЧЕННЬ І ТЕРМІНІВ…………...8

ВСТУП………………………………………………………………………………..9

РОЗДІЛ 1 ОСНОВНІ ПОНЯТТЯ РОЗРОБКИ ДОДАТКІВ ДЛЯ ANDROID.

1.1 Вибір основи для проєкту…………………….………………………….10

1.2 Налаштування проєкту…………………………………………………..12

1.3 Тестування………………………………………………………………..13

1.4 Організація ієрархії………………………………………………………14

1.5 Висновок за розділом…………………………………………………….14

РОЗДІЛ 2 МЕТОДИ ТА ЗАСОБИ ПОШУКУ ШЛЯХУ У 2D ПРОСТОРІ.

2.1 Вступ до А\*……………………………...…………………………………15

2.2 Алгоритми……………………………………………..…………………..16

2.3 Алгоритм Дейкстри ………………………………………………...……..17

2.4 Алгоритм А\*………………………………………………………..……...21

2.5 Використання евристики А\*………………………………………..…….23

2.5.1 Швидкість чи точність………………………………………….24

2.5.2 Шкала А\*………………………………………………………...25

2.5.3 Точна евристика А\*……………………………………………..26

2.5.4 Попередньо обчислена точна евристика ………………………26

2.5.5 Лінійна точна евристика…………………………………….….26

2.5.6 Евристика для сітчастих карт…………………………………..27

2.5.7 Манхеттенська метрика………………………………………...28

2.5.8 Діагональна відстань……………………………………………29

2.5.9 Евклідова відстань……………………………………………....30

2.5.10 Евклідова відстань у квадраті…………………………………31

2.5.11 Шлях до кількох цілей…………………………………………32

2.5.12 Розрив зв’язків…………………………………………………32

2.5.13 Приблизна евристика………………………………………….38

2.5.14 Ескіз…………………………………………………………….38

2.5.15 Підключення………………………………………………...…39

2.5.16 Продуктивність………………………………………………...39

2.6 Двійкові купи…………………………………...………………………….40

2.7 Відсортовані списки пропуску……………………………………...…….42

2.8 Проіндексовані масиви……………………………………………...…….43

2.9 Хеш-таблиці………………………………………………………..……...43

2.10 Splay-дерево………………………………………………………..…….44

2.11 Bucket queues………………………………………………………..……45

2.12 Горячі черги………………………………………………………..…….46

2.13 Порівняння структури данних……………………………………..……47

2.14 Гібридні уявлення…………………………………………………..……48

2.15 Взаємодія з ігровим циклом………………………………………..……48

2.16 Завчасний вихід………………………………………………..…………49

2.17 Груповий рух……………………………………………………..………50

2.18 Доопрацювання …………………………………………………..……...51

2.19 Висновок за розділом……………………………………………..……..51

РОЗДІЛ 3 ОПИС ТА РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ “NUPP NAVIGATOR”.

3.1Розробка дизайну…………………………………………………..……...53

3.2 Розробка алгоритму………………………………………………..……...58

ВИСНОВКИ………………………………………………………………………...60

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…………………………………………..61

ДОДАТОК А. ЛІСТИНГ КОДУ ДОДАТКУ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

**UI** – (User Interface) інтерфейс додатку, з котрим працює користувач.

**UE** – (Unreal Engine) ігровий двигун для розробки.

**AAA** – (triple-A) неформальний термін, що обозначує клас високобюджетних проектів.

**IOS** – мобільна [операційна система](https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) від [Apple](https://uk.wikipedia.org/wiki/Apple).

**Ctrl + S** – комбінація клавіш для збереження файлу.

**ПК** – персональний комп’ютер.

**Юніт** – ігрова одиниця (персонаж).

**AI** – (artificial intelligence) здатність комп’ютерного коду обробляти окремі об’єкти, використовуюч здобуті ранише знання та вміння.

**VS** – Visual Studio – програма для написання коду, і компіляції його.

**PNG** – розширення файлів типу Зображення з альфаканалом.

ВСТУП

Кожного року до університету вступають тисячі абітурієнтів, і що разу вони зустрічаються з такими труднощами, як орієнтація у приміщеннях нового закладу навчання, Студенти першого курсу витрачають значну частину часу, блюждаючи поверхами університету, у пошуках потрібної аудиторії. Більш досвідчені студенти також іноді стикаються зі складнощаи пошуку аудиторій, вперше відвідавши новий корпус.

Шляхом полегшення даної проблеми є створення електроннго додатку для телефонів, котрі є у кожного студена, для знаходження шляху, Це значно полегшить, та пришвидчить адаптацію першокурсників у нових стінах університету, студенти будуть мати більше часу на підготовку до занятть, не витрачаючи його на пошуки.

Додаток буде побудовано на основі Unity3D, та мові програмування C#, дизайн буде розроблятися з нуля у таких програмах як Adobe Photoshop 2020,

та Adobe Illustrator, автоматични пошук шляху буде основано на досить розповсюдженному для ігровоъ сфери рішенні, такому як алгорим пошуку шляху А\*.

Пошук шляху до аудиторії оснований вдосконаленому алгоритмі А\*, що дає змогу додатку працювати з високою продуктивнісю, та малими енерго-затратами. Вибір данного алгоритму пошуку є оптимальним, ажде він включає в себе всі найкращі риси інших відомих алгоритмів, таких як аглоритм ДЕйкстри, та Best-First-Search.

РОЗДІЛ 1.

ОСНОВНІ ПОНЯТТЯ РОЗРОБКИ ДОДАТКІВ ДЛЯ ANDROID

* 1. **Вибір основи для проєкту**

У багатьох людей може бути помилкова думка, що розробка надзвичайно складна і вимагає великої кількості попередніх знань. Однак це далеко не правда. Використання ігрового двигуну різко зменшить час навчання та дозволить більше зосередитись на тому, щоб зробити додаток більш веселим, та інтерактивним. Існує безліч різних ігрових двигунів, найпопулярніші варіанти:

* Unity;
* Unreal Engine;
* CryEngine.

Unity та Unreal Engine популярніші за CryEngine. Це тому, що вони спрямовані на розробників Indie разом із розробниками AAA, на відміну від CryEngine, який орієнтований лише на розробників AAA[1].

Різниця між інді та AAA: Індії складаються з невеликих команд, як правило, від 1 до 20 осіб, і вони не мають значних коштів тощо.

Деякі приклади незалежних розробників:

* Coffee Stain Studios;
* Halfbrick Studios;
* Mojang;
* Ndmeic Creations.

AAA складаються з набагато більших команд і мають великий досвід роботи в галузі, а також багато коштів. Кілька прикладів:

* EA;
* Ubisoft;
* Gameloft;
* Treyarch;
* Activision.

Ігровий движок – це додаток, який покликаний значно спростити дизайн / розробку додатку. Замість того, щоб кодувати кожен окремий аспект гри самостійно, ви можете просто написати сценарій поведінки, використовуючи ігровий движок. Це стає можливим завдяки тому, що ігрові двигуни виконують досить велику роботу за вас, такі речі, як фізика, освітлення, візуалізація тощо робляться за вас. Все, що вам потрібно зробити, це створити ігрові об’єкти (це просто контейнери з прикріпленими різними елементами), налаштувати порядок і макет, призначити відповідні властивості об’єктам гри та сценарій поведінки[2].

Деякі ігрові движки насправді дозволяють створювати цілу гру без будь-якого кодування, взагалі, найпопулярніший – Unreal Engine. Він має систему Blueprints, яка використовує структуру, що базується на вузлі (що, по суті, є просто системою, яка використовує поля та стрілки. Ці поля та стрілки пов'язані, якщо вони є частиною тієї самої функції). Скажімо, ви хотіли перехопити введення з клавіатури (це просто означає сказати програмі стежити за натисканням клавіші на клавіатурі), а потім змусити програвач рухатися до тих пір, поки утримується ця конкретна клавіша. Ви створили б базовий вузол, який містив би необхідні підфункції. У цьому базовому вузлі ви оголосите ключ, який хочете використовувати, а потім створите інший вузол, який повідомляє грі, що робити. Базовим вузлом буде вікно, що містить функцію натискання клавіші.

У цьому випадку другий вузол призведе до того, що гравець рухатиметься у напрямку (наприклад, вперед), доки утримується клавіша “W”. Другий вузол – це ще одне поле, пов’язане з першим вікном. Це надзвичайно корисно не лише для дизайнерів, а й для програмістів, оскільки дозволяє швидко створювати прототипи. Unity також має цю функцію, яка відома як візуальні сценарії. Однак він приймає форму активу, який ви повинні придбати[3].

Unity насправді краще для початківців, які хочуть створювати ігри, оскільки вони вчать вас писати сценарії. Unreal Engine набагато вдосконаленіший, хоча він має візуальні сценарії. А це означає, що існує більша крива навчання.

* 1. **Налаштування проекту та створення збірних елементів для частинок**

Після успішного створення проекту переконаймося, що у нас є наступні папки у папці проєкту:

* Fonts;
* materials;
* menu;
* prefabs;
* scenes;
* scripts;
* textures.

Це єдині папки, які нам знадобляться для цього проекту. Однак, якщо ми вирішимо розширити його, тоді ми можемо додавати будь-які інші папки (звуки, анімація тощо).

Після створення вищевказаних папок перейдемо у меню Файл> Параметри побудови. У цьому вікні натиснемо опцію Android під заголовком "Платформи", а потім натиснемо "Переключити платформу" в нижньому лівому куті. Це покаже Unity, що ми будуємо для Android, і відповідно налаштує проект. Щоб це працювало, нам потрібно буде встановити Android Studio. Потім збережемо сцену, натиснувши Ctrl + S або вибравши "Зберегти сцену" в меню "Файл"[4].

* 1. **Тестування**

Тестування має вирішальне значення при розробці будь-якого програмного забезпечення. Без тестування ви б не знали, чи все функціонує як слід. В ідеалі, потрібно тестувати, як тільки впроваджуємо нову функцію.

Існує кілька методів тестування. Найефективніший – це створювати та запускати програму на телефоні Android, коли ходите. Unity насправді дозволяє легко керувати цим, за умови встановлення правильних драйверів.

Інший спосіб тестування – це редактор Unity. Натиснувши кнопку “Відтворити” у верхній частині екрана посередині, Unity створить і запустить вашу програму / гру. Це позбавляє нас від необхідності використовувати телефон для незначного тестування. Однак, якщо ми хочемо використовувати цей метод для тестування своєї гри на Android, нам потрібно буде завантажити Unity Remote 4 з Google Play Store або iOS App Store. Unity Remote дозволяє протестувати свою гру в Unity, не створюючи при цьому повну збірку кожного разу. Це економить багато часу. Зазначимо лише те, що це знижує якість графіки вашого телефону. Але це необхідно. Це все ще робить хорошу роботу, показуючи вам, як ваш додаток / гра буде виглядати на вашому телефоні. Якщо ви будуєте для ПК (Windows), ви можете протестувати гру в редакторі без будь-яких додаткових інструментів[5].

Ще одним простим методом тестування є натискання на “Build” замість “Build and Run”, а потім вибір місця для збереження.apk (розширення файлу Android для своїх програм). Після того, як цей файл буде збережено та побудовано, вам потрібно буде перенести його в пам’ять телефону та встановити вручну на телефоні. Цей спосіб - це спосіб надсилання програми іншим людям, якщо ви цього забажаєте.

* 1. **Організація ієрархії**

Головна сцена буде трохи неорганізована без підтримки правильної ієрархії. Повернімося до головної сцени та створимо порожній ігровий об’єкт, позначений як 1st Flooor. Перетягнемо усі об'єкти, що відносяться до першого поерху. Так-же помістумо у окремі пусті об’єкти всі елементи що відносяться до 2-го та 3-го поверхів. Головна камера та система подій можуть залишатися там, де вони є.

Цю систему для організації необхідно використотувати для будь-якого проекту, який ми створюємо. Будь-що того ж типу може переходити під власний Порожній ігровий об’єкт (наприклад, декілька ворогів можуть бути дочерними об’єктами ігрового об’єкта ворогів, кілька камер – ігровий об’єкт камер тощо). Це дозволить зберегти нашу ієрархію охайною і не дозволить заплутатись у структурі проєкту, якщо ми створимо багато об'єктів[6].

* 1. **Висновок за розділом**

Отже, найкращим вибором для розробки невеликого додатку за коротки термін для мобільних пристроїв на базі ОС Android буде використання Unity3D, адже він має досить великі перевги над іншими у розробці мобільних додатків, та дає можливість з навеликим обсягом вміннь створити дійсно хороший проєкт.

РОЗДІЛ 2.

МЕТОДИ ТА ЗАСОБИ ПОШУКУ ШЛЯХУ У 2D ПРОСТОРІ

**2.1 Вступ до A\***

Рух по одному об’єкту здається легким. Пошук шляхів складний. Навіщо турбуватися про пошук шляхів? Розглянемо таку ситуацію (Рис 2.1).

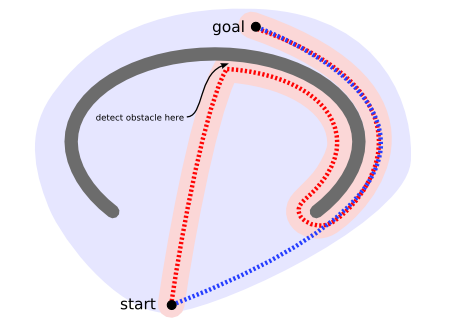


Рисунок 2.1 – Обхід перешкоди при русі по прямій

Юніт спочатку знаходиться внизу карти і хоче дістатися до верхньої частини. У області, яку він сканує (показано рожевим кольором), нічого не вказує на те, що юніт не повинен рухатися вгору, тому він продовжує свій шлях. Біля вершини він виявляє перешкоду і змінює напрямок руху. Потім вона знаходить шлях навколо перешкоди у формі U, рухаючись червоною стежкою. На відміну від них, шукач шляху просканував би більшу ділянку (показаний світло-блакитним), але знайшов би коротший шлях (синій), ніколи не посилаючи юніт на увігнуту перешкоду[7].

Однак можна розширити алгоритм руху, щоб обійти пастки, як показано вище. Необхідно або уникати створення увігнутих перешкод, або позначати їх опуклі корпуси як небезпечні (вводити, лише якщо мета всередині)(Рис 2.2).

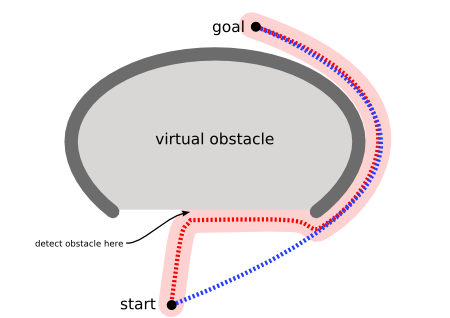


Рисунок 2.2 – обхід перешкоди за допомоги шукачів шляху

Шукачі шляху дозволяють планувати заздалегідь, а не чекати до останнього моменту, щоб виявити проблему. Існує компроміс між плануванням із пошуковими механізмами та реагуванням з алгоритмами руху. Як правило, планування відбувається повільніше, але дає кращі результати. Рух, як правило, швидший, але може застрягти. Якщо ігровий світ часто змінюється, планування на майбутнє є менш цінним[8].

Я рекомендую використовувати обидва: пошук шляхів для великих карт, з повільно змінюваними перешкодами та довгими шляхами, і рух по місцевості, із швидкими змінами та короткими шляхами.

**2.2 Алгоритми**

Алгоритми пошуку шляхів з підручників з інформатики працюють на графіках у математичному сенсі – наборі вершин з ребрами, що їх з'єднують. Тайловою ігровою картою можна вважати графік, причому кожна плитка є вершиною, а ребра, намальовані між плитками прилягають одне до одного (Рис 2.3)[9].

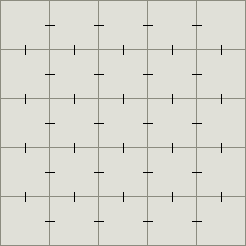


Рисунок. 2.3

Більшість алгоритмів пошуку шляхів дослідження AI або Algorithms розроблені для довільних графіків, а не мобільних додатків на основі сітки. Деякі речі котрі для людини є елементарними речами алгоритми цього не розуміють. Ми знаємо дещо про відстані: загалом, коли дві речі віддаляються, перехід від однієї до іншої займе більше часу. Ми знаємо дещо про напрямки: якщо ваш пункт призначення на схід, найкращий шлях, швидше за все, буде знайдено, пройшовши на схід, ніж пройшовши на захід. На сітках ми знаємо дещо про симетрію: більшу частину часу переміщення на північ, то на схід те саме, що рух на схід, то на північ. Ця додаткова інформація може допомогти нам змусити алгоритми пошуку шляхів працювати швидше.

**2.3** [**Алгоритм Дейкстри та найкращий перший пошук**](http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#dijkstras-algorithm-and-best-first-search)

Алгоритм Дейкстри працює, відвідуючи вершини на графіку, починаючи з початкової точки об’єкта. Потім він неодноразово досліджує найближчу ще не досліджену вершину, додаючи її вершини до безлічі вершин, які слід дослідити. Він розширюється назовні від початкової точки до досягнення цілі[10].

Алгоритм Дейкстри гарантовано знайде найкоротший шлях від початкової точки до мети, якщо жоден з країв не має негативних витрат. (Я пишу "найкоротший шлях", оскільки часто існує кілька еквівалентно коротких шляхів.) На наступній діаграмі рожевий квадрат є початковою точкою, синій квадрат – метою, бірюзові зони показують, які області сканував Алгоритм Дейкстри. Найсвітліші бірюзові ділянки – найдальші від вихідної точки, отже, утворюють "рубіж розвідки"(Рис. 2.4)[11,12].

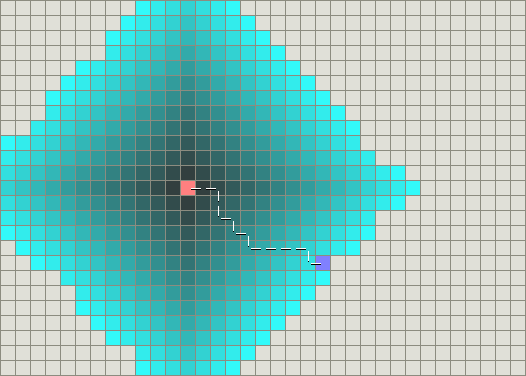


Рисунок 2.4 – Робота алгоритву Дейкстри

Алгоритм “Best-First-Search” працює подібним чином, за винятком того, що він має певну оцінку (яку називають евристичною) того, наскільки далеко від мети знаходиться будь-яка вершина.

Замість вибору вершини, найближчої до початкової точки, вона вибирає вершину, найближчу до мети. Best-First-Search – не так гарантовано знайде найкоротший шлях. Однак він працює набагато швидше, ніж Алгоритм Дейкстри, оскільки використовує евристичну функцію, щоб вести свій шлях до мети дуже швидко. Наприклад, якщо мета – на південь від вихідної позиції, Best-First-Search спочатку сфокусується на стежках, що ведуть на південь.

На наступній діаграмі жовтий колір представляє вузли з високою евристичною цінністю (висока вартість для досягнення мети), а чорні – вузли з низькою евристичною цінністю (низька вартість для досягнення цілі). Це показує, що Best-First-Search може дуже швидко знайти шляхи порівняно з алгоритмом Дейкстри (Рис. 2.5).

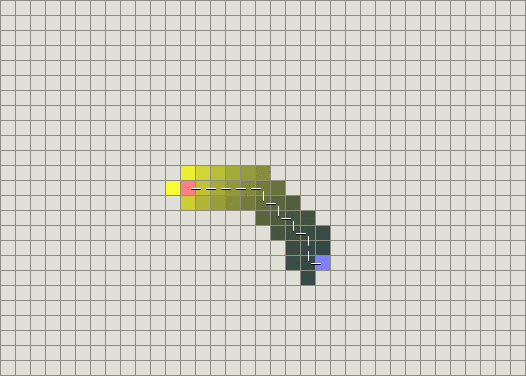


Рисунок. 2.5 – Приклад роботи алгоритму Best-First-Search

Однак обидва ці приклади ілюструють найпростіший випадок, коли на карті немає перешкод, а найкоротший шлях справді є прямою лінією. Давайте розглянемо увігнуту перешкоду, як описано в попередньому розділі. Алгоритм Дейкстри працює інтенсивніше, але гарантовано знайде найкоротший шлях (Рис. 2.6).



Рисунок. 2.6 – Алгоритм Дейкстри з перешкодою

З іншого боку, Best-First-Search робить менше роботи, але його шлях явно не такий хороший(Рис. 2.7).

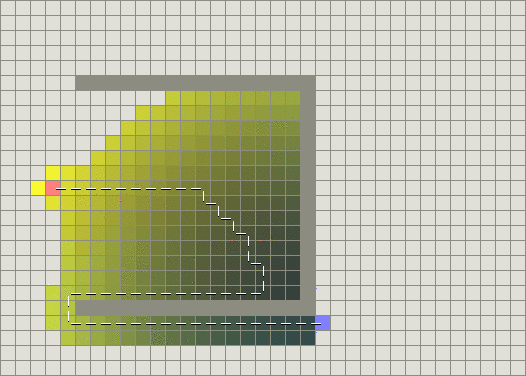


Рисунок. 2.7 – алгоритм Best-First-Search з перешкодою

Проблема в тому, що Best-First-Search – “жадібний” і намагається рухатися до мети, навіть якщо це не правильний шлях. Оскільки він враховує лише вартість досягнення цілі та ігнорує вартість шляху, він продовжує рухатися, навіть якщо шлях, яким він рухається, став справді довгим.

Чи не було б непогано поєднати найкраще з обох? A\* був розроблений в 1968 році для поєднання евристичних підходів, таких як Best-First-Search, та формальних підходів, таких як Алгоритм Дійскри. Це трохи незвично, оскільки евристичні підходи зазвичай дають вам приблизний спосіб вирішення проблем, не гарантуючи, що ви отримаєте найкращу відповідь. Однак A\* будується поверх евристики, і хоча сама евристика не дає вам гарантії, A\* може гарантувати найкоротший шлях.

**2.4** [**Алгоритм A\***](http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm)

Я зупинюсь на [алгоритмі A\*](http://en.wikipedia.org/wiki/A-star_search_algorithm). A\* – найпопулярніший вибір для пошуку шляхів, оскільки він досить гнучкий і може використовуватися в широкому діапазоні контекстів.

A\* схожий на Алгоритм Дейкстри, оскільки його можна використовувати для знаходження найкоротшого шляху. A\* схожий на Best-First-Search, оскільки він може використовувати евристику, щоб керувати собою. У простому випадку це так само швидко, як Best-First-Search(Рис. 2.8)[13].

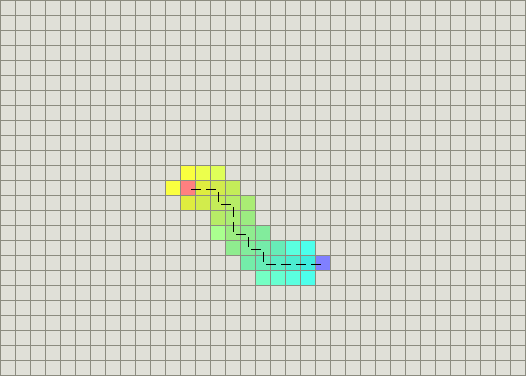


Рисунок. 2.8

У прикладі з увігнутою перешкодою A\* знаходить шлях, такий же добрий, як той, що знайшов Алгоритм Дейкстри(Рис. 2.9) Секрет його успіху полягає в тому, що він поєднує інформацію, яку використовує Алгоритм Дейкстри (надаючи перевагу вершинам, близьким до вихідної точки), та інформацію, яку використовує Best-First-Search (надаючи перевагу вершинам, близьким до мети)(Рис. 2.9).

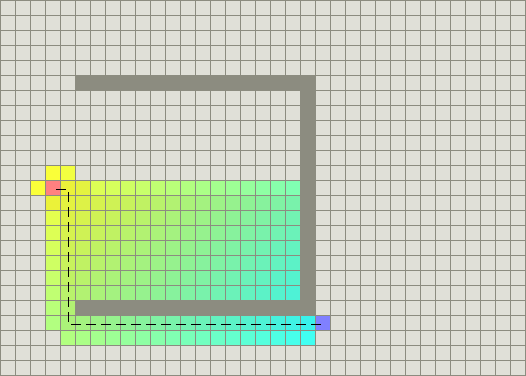


Рисунок. 2.9 – Алгоритм А\* з перешкодою

У стандартній термінології, що використовується, коли говорять про A\*, представляє точну вартість шляху від початкової точки до будь-якої вершини n та представляє евристичну розрахункову вартість від вершини n до мети. На наведених діаграмах жовтий (h) представляє вершини, далекі від мети і чирок (g) представляє вершини, далекі від вихідної точки. A\* врівноважує два, рухаючись від початкової точки до мети. Кожного разу, проходячи головний цикл, він досліджує вершину, n яка має найменшу .

**2.5** [**Використання евристики A\***](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#a-stars-use-of-the-heuristic)

Евристична функція повідомляє A\* оцінку мінімальних витрат від будь-якої вершини n до мети. Важливо вибрати хорошу евристичну функцію.

В одному крайньому випадку, якщо дорівнює 0, тоді лише грає роль, і A\* перетворюється на алгоритм Дейкстри, який гарантовано знайде найкоротший шлях.

Якщо вартість завжди менша (або дорівнює) вартості просування від n цілі, то A\* гарантовано знайде найкоротший шлях. Чим нижче , тим більше вузол A\* розширюється, роблячи його повільнішим.

Якщо точно дорівнює вартості переходу від n цілі, то A\* буде йти лише найкращим шляхом і ніколи не розширюватиме нічого іншого, роблячи це дуже швидко. Незважаючи на те, що не можна зробити це у всіх випадках, можна зробити це точним у деяких особливих випадках. Приємно знати, що з урахуванням досконалої інформації A\* буде поводитися чудово[15].

Якщо іноді перевищує вартість переходу від n цілі, то A\* не гарантовано знайде найкоротший шлях, але може працювати швидше.

В іншому ж крайньому випадку, якщо він дуже високий щодо, тоді лише відіграє певну роль, і A\* перетворюється на Best-First-Search.

Технічно алгоритм **A\*** слід називати просто **A,** якщо евристика є заниженою фактичною вартістю. Однак я продовжуватиму називати це **A\*,** оскільки реалізація однакова, і спільнота програмування не відрізняє **A** від **A\***.

Отже, у нас є цікава ситуація, коли ми можемо вирішити, що ми хочемо отримати від A\*. Маючи 100% точні оцінки, ми отримаємо найкоротші шляхи дуже швидко. Якщо ми занадто низько, то ми продовжуватимемо проходити найкоротші шляхи, але це сповільнюватиметься. Якщо ми занадто високо, то ми відмовляємось від найкоротших шляхів, але A\* працюватиме швидше.

У програмі ця властивість A\* може бути дуже корисною. Наприклад, ми можемо виявити, що в деяких ситуаціях ми віддаємо перевагу «хорошому» шляху, аніж «ідеальному». Щоб зрушити баланс між і , ми можемо змінити будь-який.

**2.5.1 Швидкість чи точність.** Здатність A\* змінювати свою поведінку на основі евристичних та вартісних функцій може бути дуже корисною у програмі. Компроміс між швидкістю та точністю можна використати, щоб зробити програму швидшою. Для більшості програм насправді не потрібен **найкращий** шлях між двома точками. [Потрібно щось близьке](http://realtimecollisiondetection.net/blog/?p=56). Те, що вам потрібно, може залежати від того, що відбувається в програмі на даниц момент, або від того, наскільки швидко працює комп'ютер/телефон. Використання функції, яка гарантує, що вона ніколи не завищує вартість, означає, що вона іноді трохи занижує вартість.

Припустимо, у вашій “грі” є два типи місцевості, рівнинна та гірська, а витрати на пересування становлять 1 для рівнинної землі та 3 для гір, A\* буде намагатися знайти найкородший шлях з уразуванням витрат, і шлях по рівнинній місцевості може бути в 3 рази довшим по відстані, але таким-же по витратам. Це тому, що це можливощо є шлях по рівнинній місцевості, яка обходить гори. Ви можете пришвидшити пошук A\*, використовуючи 1,5 як евристичну відстань між двома просторами карти. Тоді A\* буде порівнювати 3 до 1,5, і це буде виглядати не так погано, як порівняння 3 до 1. Він не такий незадоволений гірською місцевістю, тому не буде витрачати стільки часу на спроби обійти її. Крім того, ви можете пришвидшити пошук A\*, зменшивши кількість, яку він шукає для доріжок навколо гір. Як вздовж гірської місцевості. Будь-який із підходів відмовляється від ідеальних шляхів, щоб отримати щось швидше.

Вибір між швидкістю та точністю не повинен бути статичним. Ви можете динамічно вибирати на основі швидкості процесора, частки часу, що йде на пошук шляху, кількості одиниць на карті, важливості одиниці, розміру групи, рівня складності або будь-якого іншого фактора. Одним із способів зробити динаміку компромісу є побудова евристичної функції, яка передбачає, що мінімальна вартість подорожі одним простором сітки дорівнює 1, а потім побудувати функцію витрат, яка масштабується.

(2.1)

Якщо alpha дорівнює 0, то модифікована функція витрат завжди буде 1. Із за цього налаштування витрати на місцевість повністю ігноруються, і A\* працює на рівні простих прохідних / непрохідних сіткових просторів. Якщо alpha 1, то буде використана функція оригінальної вартості, і ми отримаємо повну вигоду від A\*. Можна встановити alpha де завгодно між ними.

Також слід подумати про перехід від евристичного повернення абсолютних мінімальних витрат до повернення очікуваних мінімальних витрат. Наприклад, якщо більша частина карти – це луки з вартістю пересування 2, але деякі місця на карті є дорогами з вартістю пересування 1, тоді ви можете подумати, що евристика не передбачає доріг та повернення 2 \* distance.

Вибір між швидкістю та точністю не повинен бути загальним. Можна динамічно вибирати деякі речі, виходячи з важливості точності в певному регіоні карти. Наприклад, може бути важливішим вибрати хороший шлях поблизу поточного місця, припускаючи, що ми в кінцевому підсумку можемо перерахувати шлях або змінити напрямок у якийсь момент, то навіщо турбуватися про точність щодо далекої частини шляху? Або, можливо, не так важливо мати найкоротший шлях у безпечній ділянці карти, але коли пробираєшся повз вороже село, безпека та швидкість дуже важливі.

**2.5.2** [**Шкала**](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#scale)**.** A\* обчислює Щоб додати два значення, ці два значення мають бути в одному масштабі. Якщо вимірюється в годинах і вимірюється в метрах, тоді A\* буде враховувати g або h занадто багато, або занадто мало, і ми або не отримамо настільки хороших шляхів, або A\* будете працювати повільніше, ніж могли б.

**2.5.3** [**Точна евристика**](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#exact-heuristics)**.** Якщо наша евристика точно дорівнює відстані по оптимальному шляху, ми побачимо, що A\* розширює дуже мало вузлів, як на схемі, показаній у [наступному розділі](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#manhattan-distance). Що відбувається всередині A\*, так це те, що він обчислює на кожному вузлі. Коли точно збігається , значення шляху не змінюється по шляху. Усі вузли, що не знаходяться на правильному шляху, матимуть вищі значення, f ніж вузли, які знаходяться на правильному шляху. Оскільки A\* не розглядає f вузли з більшою вартістю, поки не розглянув f вузли з меншою вартістю, він ніколи не збивається з найкоротшого шляху.

**2.5.4 Попередньо обчислена точна евристика.** Одним із способів побудови точної евристики є попереднє обчислення довжини найкоротшого шляху між кожною парою точок. Це неможливо для більшості карт. Однак існують способи наблизити цю евристику:

1. Необхідно встановити грубу сітку поверх тонкої сітки. Попередньо обчислити найкоротший шлях між будь-якою парою грубих місць розташування сітки.
2. Попередньо необхідно обчислити найкоротший шлях між будь-якою парою [точок](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html#waypoints). Це узагальнення підходу грубої сітки.
3. Необхідно додати евристику, h' яка оцінює вартість переїзду з будь-якого місця до сусідніх пунктів. (Останні також можна заздалегідь обчислити.) Кінцевою евристикою буде:

(2.2)

або якщо ви хочете кращу, але дорожчу евристику, оцініть вищезазначене з усіма парами w1, w2 які знаходяться близько до вузла та цілі, відповідно.

**2.5.5 Лінійна точна евристика.** За особливих обставин ви можете точно встановити евристику, нічого попередньо не обчислюючи. Якщо у вас є карта без перешкод і повільної місцевості, то найкоротший шлях від початкової точки до цілі повинен бути прямою лінією.

Якщо ви використовуєте просту евристику (таку, яка не знає про перешкоди на карті), вона повинна точно відповідати евристичній. Якщо цього не сталося, то у вас можуть бути проблеми із масштабом або типом евристики, який ви вибрали.

**2.5.6** [**Евристика для сітчастих карт**](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#heuristics-for-grid-maps)**.** У сітці є добре відомі евристичні функції, які слід використовувати.

**Використаймо евристику відстані, яка відповідає дозволеному руху:**

1. На квадратній сітці, яка дозволяє **4 напрямки** руху, використовуємо відстань Манхеттена (L1).
2. На квадратній сітці, яка дозволяє **8 напрямків** руху, використовуємо діагональну відстань (L∞).
3. На квадратній сітці, що дозволяє **будь-який напрямок** руху, ми можемо дозволити, або заборонити використання евклідової відстані (L2). Якщо A\* знаходить шляхи на сітці, але ви дозволяєте рух не по сітці, ви можете розглянути [інші зображення карти](https://www.redblobgames.com/pathfinding/grids/algorithms.html).
4. На шестигранній сітці, що дозволяє здійснювати **6 напрямків** руху, використовуємо відстань Манхеттена, [пристосовану до шестикутних сіток](https://www.redblobgames.com/grids/hexagons/#distances).

Помножимо відстань у кроках на мінімальну вартість кроку. Наприклад, якщо ми вимірюємо в метрах, відстань дорівнює 3 квадратам, а кожен квадрат – 15 метрів, то евристика поверне метрів. Якщо ми вимірюємо в часі, відстань дорівнює 3 квадратам, і кожен квадрат займає не менше 4 хвилин, щоб перетнути, то евристика повернеться хвилин. Одиниці (метри, хвилини тощо), що повертаються евристикою, повинні відповідати одиницям, що використовуються функцією витрат[16].

**2.5.7 Манхеттенська метрика.** Стандартною евристикою для квадратної сітки є [відстань](http://en.wikipedia.org/wiki/Taxicab_geometry) до [Манхеттена](http://en.wikipedia.org/wiki/Taxicab_geometry). Подивимося на свою функцію витрат і знайдемо мінімальну вартість D переїзду з одного простору в сусідній простір. У простому випадку ми можемо встановити D рівне 1. Евристика на квадратній сітці, де ми можемо рухатись у 4 напрямках, повинна бути в D раз більше, ніж Манхеттенська відстань:

function heuristic(node) =

dx = abs(node.x - goal.x)

dy = abs(node.y - goal.y)

return D \* (dx + dy)

Як вибрати D? Використовумємо шкалу, яка відповідає нашій функції витрат. Для найкращих шляхів та «допустимої» евристики встановимо D як найнижчу вартість між сусідніми квадратами. За відсутності перешкод та на місцевості, що має мінімальну вартість руху D, просування на крок ближче до мети повинно збільшуватися g на D і зменшуватися h на D. Коли ви додасте два, f (для якого встановлено значення g + h) залишиться незмінним. це ознака відповідності шкал евристики та функції витрат.

Ми також можемо відмовитись від оптимальних шляхів, щоб A\* працював швидше, збільшуючи D або зменшуючи співвідношення між найнижчими та найвищими граничними витратами(Рис 2.10).

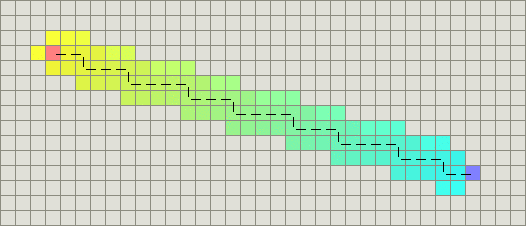


Рисунок. 2.10 **–** A\* з евристичним [вимикач](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties)ем.

**2.5.8 Діагональна відстань.** Якщо карта дозволяє діагональне переміщення, потрібна інша евристика. Відстань до Манхеттена для (4 схід, 4 північ) буде 8. Однак замість цього ми можемо просто рухатися (4 на північний схід), тому евристика повинна бути , де D2 – вартість переміщення по діагоналі(Рис 2.11).

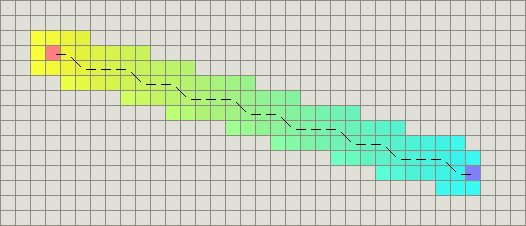


Рисунок. 2.11 – Діагональна відстань

function heuristic(node) =

dx = abs(node.x - goal.x)

dy = abs(node.y - goal.y)

return D \* (dx + dy) + (D2 - 2 \* D) \* min(dx, dy)

Тут ми обчислюємо кількість кроків, які ви робите, якщо ви не можете зробити діагональ, потім віднімаємо кроки, які ви зберігаєте, використовуючи діагональ. Є діагональні сходинки, і кожен з них коштує, D2 але економить вам 2 \*D недіагональні сходинки.

Коли D = 1 і D2 = 1, це називається [відстань Чебишева](http://en.wikipedia.org/wiki/Chebyshev_distance). Коли D = 1 і D2 = sqrt (2), це називається октильною відстанню.

Інший спосіб це написати . [Патрік Лестер пише це ще по-іншому](https://web.archive.org/web/20171019182159/http:/www.policyalmanac.org/games/heuristics.htm), Якщо . Всі вони рівнозначні.

**2.5.9 Евклідова відстань.** Якщо юніти можуть рухатися під будь-яким кутом (замість напрямків сітки), то слід використовувати пряму відстань:

function heuristic(node) =

dx = abs(node.x - goal.x)

dy = abs(node.y - goal.y)

return D \* sqrt(dx \* dx + dy \* dy)

Однак у цьому випадку у вас можуть виникнути проблеми із безпосереднім використанням A\*, оскільки функція витрат g не буде відповідати евристичній функції h.

Оскільки евклідова відстань коротша за Манхеттенську або діагональну відстань, ми все одно отримаємо найкоротші шляхи, але прорахування шляху A\* займе більше часу(Рис. 2.12).

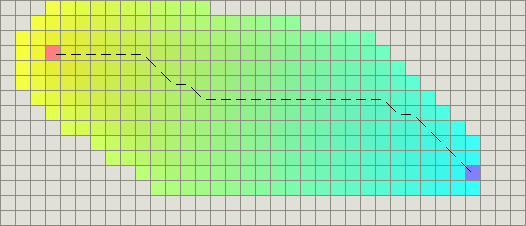


Рисунок. 2.12 – Евклідова відстань

**2.5.10 Евклідова відстань, у квадраті.** Я бачив, як кілька веб-сторінок A\* рекомендують уникати дорогого квадратного кореня на евклідовій відстані, використовуючи відстань у квадраті:

function heuristic(node) =

dx = abs(node.x - goal.x)

dy = abs(node.y - goal.y)

return D \* (dx \* dx + dy \* dy)

Ми, безумовно, стикаємося з проблемою масштабу. Масштаб g і h має збігатись, тому що ми додаємо їх у форму f. Коли A\* обчислює , квадрат відстані буде набагато вищим, ніж вартість, g і ми отримаємо завищену евристику. На великих відстанях це наближається до крайньої міри, якщо не сприятиме, і A\* перетвориться на Best-First-Search(Рис. 2.13).

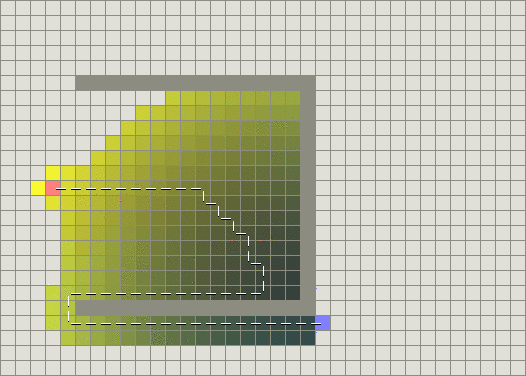


Рисунок. 2.13 – Best-First-Search

Щоб спробувати це виправити, ми можете зменшити евристику. Однак тоді ви стикаєтесь із протилежною проблемою: на менших відстанях евристика буде занадто малою в порівнянні з, g(n) а A\* перетворюэться в алгоритм Дейкстри.

Якщо після профілювання ми виявимо, що вартість квадратного кореня є значною, або скористаймося швидким наближенням квадратного кореня з евклідовою відстанню, або використовуймо діагональну відстань як наближення до евклідової.

**2.5.11 Шлях до кількох цілей.** Якщо ми хочемо знайти будь-яку з кількох цілей, побудуємо евристику, яка є мінімальною, є евристика для кожного з найближчих місць.

Одним із способів реалізувати це є те, що ми можемо додати новий край нульових витрат від кожної з цілей до нового вузла графіка. Шлях до цього нового вузла обов'язково пройде через один із вузлів цілі.

Якщо ми хочемо шукати шляхи до кількох цілей, кращий варіант може бути алгоритм Дейкстри з раннім виходом, коли ми знайдемо всі цілі. Може існувати варіант A\*, який може обчислити ці шляхи.

Якщо ми хочемо знайти одну кінцеву точку, за допомоги A\* знайдемо шлях до центру області. Під час обробки вузлів із набору OPEN відкриємо вихід, коли ми потягнемо вузол, який знаходиться досить близько.

**2.5.12 Розрив зв’язків.** На деяких сіткових картах є багато шляхів однакової довжини. Наприклад, на рівнинних ділянках без зміни рельєфу місцевості використання сітки призведе до багатьох шляхів однакової довжини. A\* може досліджувати всі шляхи з однаковими f значеннями, а не лише одним(Рис. 2.14).

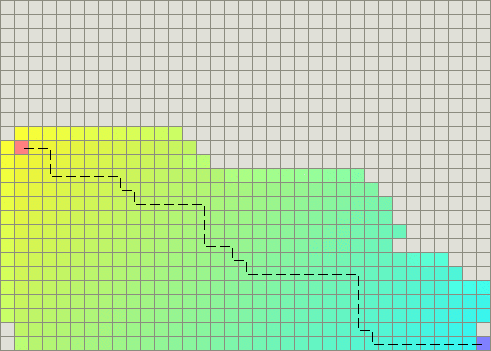


Рисунок. 2.14

Швидкий спосіб вирішити цю проблему – це скоригувати значення g або h. Вимикач повинен бути детермінованим по відношенню до вершини (тобто це не повинно бути випадковим числом), і він повинен змусити f значення відрізнятися. Оскільки A\* сортується за f значенням, якщо вони різні, це означає, що буде досліджено лише одне із “еквівалентних” f значень.

Одним із способів розірвати зв’язки є h злегка змінити масштаб. Якщо ми масштабуємо його вниз, тоді f буде збільшуватися, коли ми рухаємось до мети. На жаль, це означає, що A\* воліє розширювати вершини поблизу вихідної точки замість вершин, близьких до мети. Натомість ми можемо h трохи масштабувати вгору (навіть на 0,1%). A\* воліє розширювати вершини поблизу мети. heuristic \* = (1,0 + p).

Коефіцієнт p слід вибирати таким чином, щоб:

p < (мінімальна вартість кроку) / (очікувана максимальна довжина шляху).

Припускаючи, що ми не очікуємо, що шляхи будуть більше 1000 кроків, ми можемо вибрати p = 1/1000. (Зверніть увагу, що це трохи порушує "допустимість" евристики, але в мобільних програмах/іграх це майже ніколи не має значення.) Результатом цього поштовху є те, що A\* досліджує набагато менше карти, ніж раніше(Рис. 2.15)[17].

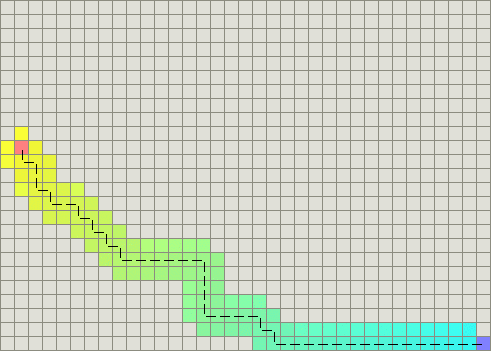


Рисунок. 2.15 – Розрив зв’язків А\*

Звичайно, коли є перешкоди, йому все одно потрібно дослідити, щоб знайти їх, але зауважте, що після подолання перешкоди A\* досліджує дуже мало(Рис. 2.16).

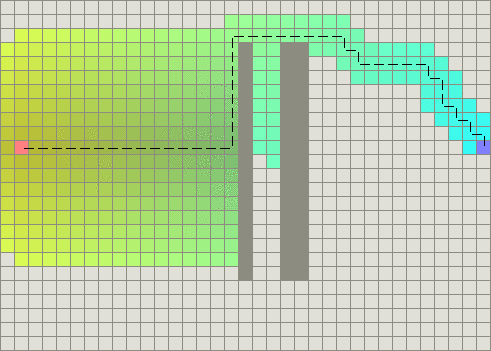


Рисунок. 2.16 Масштабування розривів, додане до евристики, добре працює з перешкодами.

Стівен ван Дейк припускає, що більш простий спосіб зробити це – передати h до функції порівняння. Коли f значення рівні, функція порівняння розірве зв'язок, дивлячись на h.

Інший спосіб розірвати зв'язки – це додати детерміноване випадкове число до евристичних або граничних витрат. (Одним із способів вибору детермінованого випадкового числа є обчислення хешу координат.) Це розриває більше зв’язків, ніж коригування, h як зазначено вище.

Ще один спосіб розірвати зв’язки – віддати перевагу шляхам, які проходять вздовж прямої лінії від початкової точки до мети:

dx1 = current.x - goal.x

dy1 = current.y - goal.y

dx2 = start.x - goal.x

dy2 = start.y - goal.y

cross = abs(dx1\*dy2 - dx2\*dy1)

heuristic += cross\*0.001

Цей код обчислює поперечний добуток вектора між вектором від початку до цілі та поточним вектором від точки до цілі. Коли ці вектори не вибудовуються, перехресний добуток буде більшим. Результат полягає в тому, що цей код надасть деяку незначну перевагу шляху, який лежить вздовж прямолінійного шляху від початку до мети. Коли немає перешкод, A\* не тільки досліджує менше карти, але й шлях виглядає дуже добре(Рис 2.17).

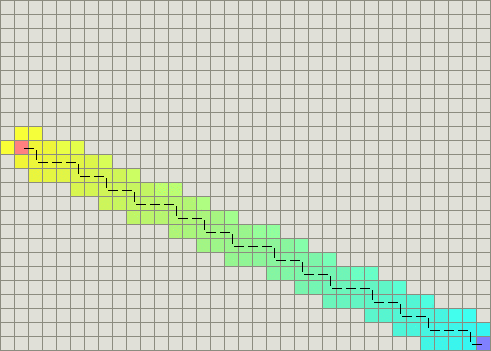


Рисунок. 2.17 Поперечний продукт, що порушує зв'язок, доданий до евристики, створює гарні шляхи.

Однак, оскільки цей вимикач віддає перевагу шляхам по прямій від вихідної точки до мети, при об’їзді перешкод з'являються невеликі помилки (Рис 2.18).

Можемо використати “Clear”, щоб очистити карту, і обрати дві точки на протилежних кутах карти. Коли ми використовуємо метод «Classic A\*», ми можемо побачити ефект зав’язок. Коли ми використовуємо метод “Фаджа”, ми побачимо ефект вищезазначеного перехресного продукту, доданий до евристики[18].

Ще один спосіб розірвати зв’язки – це ретельно побудувати свою чергу пріоритетів A\*, щоб нові вставки з певними f значеннями завжди ранжирувались краще (нижче), ніж старі вставки з тим самим f значенням.

І ще один спосіб розірвати зв’язки на сітках – мінімізувати повороти. Зміна x, y від батьківського до поточного вузла повідомляє нам, у якому напрямку ми рухалися. Для всіх ребер, що враховуються від поточного до сусіднього, якщо зміна x, y відрізняється від тієї, що від батьківського до поточного, тоді додати невеликий штраф до вартості руху(Рис. 2.18).

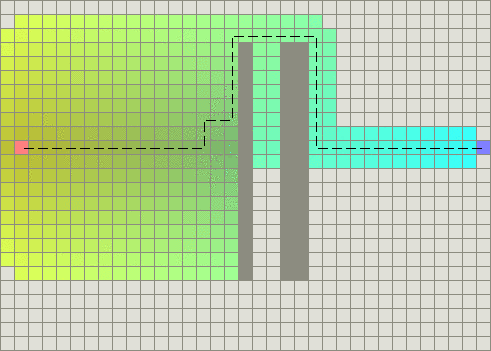


Рисунок. 2.18 Перехресний продукт, що розриває зв'язок, додається до евристики із перешкодами.

У своїх власних проектах я використовую «шаховий» підхід. У першому пошуку за шириною я змінюю порядок вставки так, що шляхи чергуються між горизонтальними та вертикальними кроками. В Алгоритмі Дейкстри та A\* я варіюю вартість руху так, що шляхи чергуються між горизонтальними та вертикальними кроками.

**Вищезазначені модифікації евристики або витрат є "допоміжним засобом", що виправляє основну неефективність.** Зв’язки виникають, коли існує безліч однаково хороших шляхів, що приводить до великої кількості вузлів для дослідження. Розглянемо способи «працювати розумніше, а не важче»:

Альтернативні [подання карт](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html) можуть вирішити проблему, **зменшивши кількість вузлів на графіку**. Згортання кількох вузлів в один або шляхом видалення всіх, крім важливих вузлів. [Прямокутне зменшення симетрії](http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/) – це спосіб зробити це на квадратних сітках. [Ієрархічне визначення](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html#hierarchical) шляху використовує графік високого рівня з кількома вузлами, щоб знайти більшу частину шляху, потім графік низького рівня з більшою кількістю вузлів для уточнення шляху.

Деякі підходи залишають кількість вузлів у спокої, але **зменшують кількість відвідуваних вузлів**. [Пошук точки](http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/) переходу пропускає великі ділянки вузлів, які містять багато зв'язків, він призначений для квадратних сіток. [Посилання для](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html#skip-links) пропуску додають краї ярликів, які пропускають ділянки карти. Алгоритм [AlphA\*](https://web.archive.org/web/20060303021605/http:/home1.stofanet.dk/breese/astaralpha-submitted.pdf.gz) додає певний пошук у глибину до звичної поведінки A\*, що стосується широти, так що він може досліджувати єдиний шлях, замість того, щоб обробляти їх усі одночасно[19].

Натомість [Fringe Search](http://cswww.essex.ac.uk/cig/2005/papers/p1039.pdf) вирішує проблему, **пришвидшуючи обробку вузлів.** Замість того, щоб отримати набір OPEN відсортованим і відвідувати вузли по одному, він обробляє вузли партіями, розширюючи лише ті вузли, які мають низькі значення f. Це пов’язано з підходом [горячих черг](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#hot-queues).

**2.5.13** [**Приблизна евристика**](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#approximate-heuristics)**.** Евристика, що має точну відстань, ідеально підходить для швидкості A\*, але це, як правило, недоцільно. Ми часто можемо попередньо обробити графік для побудови приблизної відстані та використовувати це наближення в евристиці A\*.

[ALT A\*](http://research.microsoft.com/pubs/154937/soda05.pdf) використовує “орієнтири” та нерівність трикутника для попередньої обробки графіка пошуку шляхів, щоб зробити пошук шляхів набагато швидшим. ALT також робить кілька інших речей, але евристичне вдосконалення – це та частина, яка привернула мою увагу. Це напрочуд просто для реалізації, іноді менше 15 рядків коду, і створює вражаючі прискорення.

Назва "орієнтир" трохи вводить в оману. Ці точки потрібно розмістити на зовнішніх краях карти. Деякі автори називають це "диференціальною евристикою"[20,21].

Орієнтирний підхід зберігає багато даних, які можна стиснути. [Стиснута диференціальна евристика](https://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4020/4340) показує результати стиснення даних орієнтиру. Ми можемо зберігати набагато більше орієнтирів в одному просторі, завдяки чому ми отримаємо покращені евристичні значення.

**2.5.14** [**Ескіз**](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#sketch)**.** Алгоритм A\*, позбавлений всього коду, досить простий. Є два набори, OPEN та CLOSED. Набір OPEN містить ті вузли, які є кандидатами на перевірку. Спочатку набір OPEN містить лише один елемент: вихідну позицію. Набір CLOSED містить ті вузли, які вже були розглянуті. Спочатку набір CLOSED порожній. Графічно набір «OPEN» – це «кордон», а «CLOSED» – «інтер’єр» відвідуваних районів. Кожен вузол також зберігає вказівник на батьківський вузол, щоб ми могли визначити, як його було знайдено.

Існує основний цикл, який неодноразово витягує найкращий вузол n в OPEN (вузол з найменшим f значенням) і перевіряє його. Якщо n це мета, то ми закінчили. В іншому випадку вузол n видаляється з OPEN та додається до CLOSED. Потім n′ обстежує своїх сусідів. Сусіда, який перебуває у CLOSED, уже бачили, тому нам не потрібно перевіряти його. Заплановано перегляд сусіда, який перебуває у відкритому стані, тому нам не потрібно його перевіряти зараз. В іншому випадку ми додаємо його до OPEN, з батьківським значенням n. Вартість шляху до n′, g(n′) буде встановлена g(n) + movementcost(n, n′).

**2.5.15 Підключення**. Якщо у вашій програмі/грі є ситуації, коли старт і мета зовсім не пов’язані графіком, A\* триватиме багато часу, оскільки їй доведеться дослідити кожен вузол, підключений з самого початку, перш ніж він зрозуміє, що шляху немає. Обчислити [зв'язкові компоненти](http://en.wikipedia.org/wiki/Connected_component_(graph_theory)) першого і використовувати тільки A\*, якщо початок і мета знаходяться в тому ж регіоні.

**2.5.16** [**Продуктивність**](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#performance)**.** Основний цикл A\* читає з черги пріоритетів, аналізує його та вставляє вузли назад у чергу пріоритетів. Крім того, він відстежує, які вузли були відвідані. Для підвищення продуктивності розглянемо:

Чи можемо ми зменшити розмір графіку? Це зменшить кількість вузлів, які обробляються, як на шляху, так і на тих, які не опиняються на кінцевому шляху.

Чи можемо ми підвищити точність евристики? Це зменшить кількість вузлів, які не знаходяться на кінцевому шляху. Чим ближче евристика до фактичної довжини шляху (а не відстані), тим менше вузлів A\* буде досліджуватися. Розглянемо [ці евристики для сіток](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#exact-heuristics). Розглянемо ALT (A\*, орієнтири, нерівність трикутника) для графіків загалом (включаючи сітки)[22].

Чи можемо ми швидше зробити чергу пріоритетів? Розглянемо [інші структури даних](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#set-representation) для нашої черги пріоритетів. Розглянемо можливість обробки вузлів пакетами, як це робить [пошук по](http://en.wikipedia.org/wiki/Fringe_search) краях. Розглянемо приблизне сортування.

Чи можемо ми зробити евристику швидшою? Евристична функція викликається для кожного відкритого вузла.

Що перше, що ви подумаєте використовуючи для наборів OPEN та CLOSED? Якщо ви схожі на мене, ви, мабуть, думали "масив". Можливо, ви також думали про “зв’язаний список”. Існує багато різних структур даних, якими ми можемо скористатися, але, щоб вибрати одну, слід подивитися, які операції потрібні.

Є три основні операції, які ми виконуємо на наборі OPEN: основний цикл багаторазово знаходить найкращий вузол і видаляє його, сусід, який відвідує, перевірить, чи є вузол у наборі, а сусід, який відвідує, вставить нові вузли. Вставка та видалення найкраого – це операції, типові для [черги пріоритетів](http://en.wikipedia.org/wiki/Priority_queue).

Вибір структури даних залежить не тільки від операцій, але і від кількості запусків кожної операції. Перевірка членства виконується один раз для кожного сусіда для кожного відвіданого вузла. Вставка виконується один раз для кожного вузла, що розглядається. Видалення найкращих запусків один раз для кожного відвіданого вузла. Більшість вузлів, які розглядаються, буде відвідано, ті, що не є окраїною простору пошуку. Оцінюючи вартість операцій з цими структурами даних, нам потрібно враховувати максимальний розмір грані (F).

Крім того, існує четверта операція, яка є відносно рідкісною, але все ще потребує реалізації. Якщо вузол, що перевіряється, уже знаходиться в наборі OPEN (що часто трапляється), і якщо f значення краще, ніж те, що вже є в наборі OPEN (що рідко), тоді значення в наборі OPEN має бути відкоригованим. Операція регулювання передбачає видалення вузла (що є не найкращим f) та повторне його вставлення. Ці два кроки можуть бути оптимізовані для операції підвищення пріоритету, яка переміщує вузол (це також називається ключем зменшення).

Найкращий загальний вибір – [це двійкова купа](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#binary-heaps). Якщо є бібліотека двійкової купи, можна використати її. Якщо ні, можна почати з [відсортованих масивів](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#sorted-arrays) або [невідсортованих масивів](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#unsorted-arrays-or-linked-lists) і перейти до двійкових куп, якщо необхідно збільшити продуктивність. Якщо у наборі OPEN більше 10 000 елементів, можна розглянути більш складні структури, такі як система сегментування.

**2.6 Двійкові купи**

[Двійкові купи](http://en.wikipedia.org/wiki/Heap_(data_structure)) (не плутати з купою пам'яті) являє собою деревоподібну структуру, яка зберігається в масиві. На відміну від більшості дерев, які використовують покажчики для посилання на дочерні об'єкти, двійкова купа використовує індексацію для пошуку дочерних об'єктів.

Операція збільшення пріоритету складна. На жаль, більшість пріоритетних бібліотек черг не включають цю операцію. На щастя, це не є суворо необхідним. Замість збільшення пріоритету можна вставити новий елемент у чергу пріоритетів. Ми потенційно в кінцевому підсумку обробляємо вузол двічі, але це відносно дешево порівняно із впровадженням пріоритету.

У C ++ можна використовувати клас [priority\_queue](http://en.cppreference.com/w/cpp/container/priority_queue), який не має пріоритету збільшення, або [змінній черзі пріоритетів Boost](http://www.boost.org/doc/libs/1_50_0/doc/html/heap/concepts.html#heap.concepts.mutability). У Python можна використати [бібліотеку heapq](https://docs.python.org/3/library/heapq.html).

Ми можемо поєднати хеш-таблицю або індексований масив для членства та черги пріоритетів для управління пріоритетами.

Варіант бінарної купи – це [d-ary](http://en.wikipedia.org/wiki/D-ary_heap) heap, яка має більше 2 дочірніх елементів на вузол. Вставки та збільшення пріоритету стають трохи швидшими, але видалення стають трохи повільнішими. Вони можуть мати кращу продуктивність кешу. [B-heaps теж варто подивитися, якщо ваш кордон великий](https://queue.acm.org/detail.cfm?id=1814327)

Якщо двійкові купи недостатньо добре відпрацьовують, то можна подумати про поєднання купи, куп послідовності або підходу на основі сегмента. Статтю [Пріоритетні черги та Алгоритм Дейкстри](http://www3.cs.stonybrook.edu/~rezaul/papers/TR-07-54.pdf) варто продивится, якщо ми не можемо зменшити графік і будемо потребувати швидшої черги пріоритетів.

**2.7 Відсортовані списки пропуску**

Пошук несортованого пов’язаного списку відбувається повільно. Ми можемо зробити це швидше, якщо замість пов’язаного списку будемо використовувати [пропущений](http://en.wikipedia.org/wiki/Skip_list) список. У списку пропуску членство є швидким, якщо у вас є ключ сортування, як зв’язаний список, якщо ви знаєте, куди вставити. Пошук найкращого вузла відбувається швидко, якщо ключ сортування – f. Операція збільшення пріоритету передбачає пошук вузла, його видалення та повторне вставлення.

Якщо ми використовуємо пропущені списки з місцем на карті як ключем, після того, як ми провели тест на членство, знаходження найкращого вузла та видалення попереднього вузла. Це краще, ніж невідсортовані зв’язані списки, оскільки членство швидше.

Якщо ми використовуємо пропущені списки зі f значенням як ключем, членство – O (F), вставка O (1), пошук найкращого вузла – O (1), а видалення вузла – O (1). Це не краще відсортованих зв’язаних списків.

**2.8 Проіндексовані масиви**

Якщо набір вузлів є кінцевим і має розумні розміри, ми можемо використовувати структуру прямого індексування, де функція індексу i(n) відображає кожен вузол n до індексу в масив. На відміну від невідсортованих та відсортованих масивів, розмір яких відповідає найбільшому розміру OPEN, при індексованому масиві розмір масиву завжди max(i(n)) перевищує всі n. Якщо функція щільна ( тобто індекси не використовуються), тоді буде вказано кількість вузлів у графіку. Завжди, коли карта є сіткою, функцію легко зробити щільною.

**2.9 Хеш-таблиці**

Індексовані масиви займають багато пам'яті для зберігання всіх вузлів, яких немає в наборі OPEN. Альтернативою є використання хеш-таблиці з хеш-функцією, яка відображає кожен вузол nу хеш-код. Зберігайте хеш-таблицю вдвічі більшу за N, щоб імовірність зіткнень була низькою. Припускаючи , що , тест на членство очікується O (1), вставка очікується O (1), а найкраще видалити O (numnodes), оскільки нам доводиться шукати всю структуру. Операція збільшення пріоритету – O (1).

Хеш-таблиці найкраще підходять для встановленого членства, але не для управління пріоритетами. Я використовую [хеш-таблиці для членства та двійкові купи для пріоритетів](https://www.redblobgames.com/pathfinding/a-star/implementation.html). Я об’єднав набори OPEN та CLOSED в один, який я називаю VISITED.

**2.10 Splay-дерево**

Купи – це деревоподібна структура з очікуваними операціями часу . Однак проблема полягає в тому, що для A\* загальна поведінка полягає в тому, що у вас є недорогий вузол, який видаляється (виклиепючи , оскільки значення повинні рухатися вгору від самого низу дерева), за якими слідують не дорогі вузли, котрі додаються (що спричиняє поведінку , оскільки ці значення мають рухатись до самого верху дерева). Очікувана поведінка куп тут є еквівалентно найгіршою поведінкою. Ми можемо зробити краще, якщо знайдемо структуру даних, де очікуваний випадок кращий, навіть якщо гірший випадок не кращий.

Дерева нахилу – це саморегулююча структура дерева. Будь-який доступ до вузла у дереві має тенденцію підносити цей вузол до верху. Результатом є ефект «кешування», коли рідко використовувані вузли йдуть внизу і не сповільнюють роботу. Не має значення, наскільки велике ваше дерево відтворення, оскільки операції є настільки ж повільними, як “розмір кешу”. В A\* недорогі вузли використовуються часто, а дорогі вузли не використовуються тривалий час, тому ці дорогоцінні вузли можуть переміщатися внизу дерева.

З деревами візуалізації, членство, вставка, найкраще видалення та підвищення пріоритету – це все, що очікується , в найгіршому випадку . Однак, як правило, кешування утримує найгірший випадок.

Алгоритм Дейкстри та A\* із заниженою евристикою, однак, мають деякі своєрідні характеристики, які можуть перешкодити деяким деревам бути швидшими. Зокрема, для вузлів n та сусіднього вузла n'. Коли це трапляється, може статися, що всі вставки відбуваються на одній стороні дерева і в підсумку виводять його з рівноваги.

**2.11 Bucket queues**

[Bucke](https://en.wikipedia.org/wiki/Bucket_queue)t queues корисні, коли діапазон витрат руху обмежений малими цілими числами, особливо для алгоритму Дейкстри. Враховуючи обмежений діапазон, часто існують кращі алгоритми. Наприклад, сортування може здійснюватися за довільними значеннями за час , але коли є фіксований діапазон.

Розглянемо алгоритм Дейкстри: межа має пріоритети p шляхом, де k має найбільшу вартість руху. Наприклад, якщо ваші витрати на рух 1, 2 або 3, то все в межах має пріоритет . Використаймо 4 відра, по одному на кожне відро. Не потрібно сортувати в сегменті, оскільки пріоритети однакові. І немає необхідності сортувати відра. Це означає, що вставка – O (1), а видалення найкращого – O (1).

З A\* це дещо складніше, оскільки також потрібно дивитись на вплив евристики на пріоритет. Відрів буде більше, ніж з алгоритмом Дейкстри.

**2.12 Горячі черги**

Горячі черги – це варіант [Bucke](https://en.wikipedia.org/wiki/Bucket_queue)t queues, які приймають більший діапазон значень. Замість того, щоб кожен сегмент мав рівно один пріоритет, кожен сегмент має ряд пріоритетів. Оскільки ми видаляємо лише елементи з верхнього сегмента, потрібно замовити лише верхній сегмент.

Горячі черги змушують верхній сегмент використовувати двійкову купу. Усі інші сегменти є несортованими масивами. Перевірка членства – , оскільки ми не знаємо, в якому сегменті знаходиться вузол. Вставлення та видалення – найкраще у верхньому сегменті – , для K сегментів. Вставлення в інші сегменти має значення O (1), а функція видалення–найкраще ніколи не працює на інших сегментах. Якщо верхній сегмент спорожняється, тоді нам потрібно перетворити наступний сегмент, невідсортований масив, у двійкову купу. Виявляється, цю операцію (“heapify”) можна запустити за час . Операцію з пріоритетом збільшення найкраще розглядати як видалення з подальшим введенням або .

У A\* багато вузлів, які ми вводимо в OPEN, нам насправді ніколи не потрібні. Горячі черги – це великий виграш, оскільки елементи, які не потрібні, вставляються за час O (1). Тільки елементи, які потрібні, стають важкими (що не надто дорого). Єдина операція, яка перевищує , – це видалення вузла з купи, що є лише .

Швидким варіантом гарячої черги є дворівнева черга: помістімо “хороші” вузли в одну структуру даних (купу або масив), а “погані” вузли – в іншу структуру даних (масив або пов'язаний список). Оскільки більшість вузлів, що вводяться в OPEN, є "поганими", вони ніколи не перевіряються, і немає шкоди, якщо їх розмістити у великому масиві.

Теоретично, купи Фібоначчі – хороші черги пріоритетів для A\*. Однак на практиці вони не використовуються. [Парування купи](http://en.wikipedia.org/wiki/Pairing_heap) можна розглядати як спрощену купу Фібоначчі. Кажуть, вони добре працюють на практиці.

Для пошуку шляхів нам часто не потрібен точний найкоротший шлях, і зазвичай ми віддаємо перевагу швидкому обчисленню досить короткого шляху. Таким чином, це може бути один з тих випадків, коли це є корисним на практиці.

"Накопичення послідовностей в даний час може бути найшвидшою доступною структурою даних для великих черг пріоритетів на основі порівняння як в кешованій, так і в зовнішній пам'яті.

**2.13 Порівняння структури даних**

Важливо пам’ятати, що ми шукаємо не просто асимптотичну (“велику О”) поведінку. Ми також хочемо шукати низьку константу. Щоб зрозуміти, чому, розглянемо алгоритм, який має значення O (log F), а інший – O (F), де F – кількість елементів у купі. Можливо, на вашій машині реалізація першого алгоритму займає 10000 \* (F) секунд, тоді як реалізація другого – 2 \* F секунд. Для F = 256 перший займає 80 000 секунд, а другий – 512 секунд. "Швидший" алгоритм займає в цьому випадку більше часу, і він починає бути швидшим, коли F> 200 000.

Ми не можемо просто порівняти два алгоритми. Слід також порівняти реалізації цих алгоритмів. Ми також повинні знати, якого розміру можуть бути наші дані. У наведеному вище прикладі перша реалізація швидша для F> 200 000, але якщо у нашому додатку F залишається менше 30000, то друга реалізація була б кращою.

Жодна з основних структур даних не є цілком задовільною. Несортовані масиви або списки роблять вставки дуже дешевими, а членство та видалення дуже дорогими. Відсортовані масиви або списки роблять членство дещо дешевим, видалення дуже дешевим і введення дуже дорогим. Бінарні купи роблять введення та видалення дещо дешевими, але членство дуже дороге. Горячі черги роблять вставки дешевими, видалення досить дешевими, а тести на членство дещо дорогими. Індексовані масиви роблять членство та вставку дуже дешевими, але видалення неймовірно дорогі, і вони також можуть зайняти багато пам'яті. Таблиці хеш-функцій працюють подібно до індексованих масивів, але в загальному випадку вони можуть зайняти набагато менше пам'яті, а видалення просто дорогі, а не надзвичайно дорогі.

**2.14 Гібридні уявлення**

Щоб отримати найкращу продуктивність, нам потрібна гібридна структура даних. Для мого коду A\* я використав індексований масив для тесту членства O(1) та двійкову купу для вставки O(log F) та O(log F) remove-best. Для збільшення пріоритету я використовував індексований масив для тесту O(1), чи дійсно мені потрібно було змінити пріоритет (шляхом збереження g значення в індексованому масиві), а потім у тих рідкісних випадках, коли мені потрібно було збільшити пріоритет, я використав пріоритет збільшення O(F) у двійковій купі. Ми також можемо використовувати індексований масив для зберігання розташування в купі кожного вузла, це дасть вам для підвищення пріоритету.

**2.15** [**Взаємодія з ігровим циклом**](http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#interaction-with-the-game-loop)

Інтерактивні (особливо в режимі реального часу) ігри вводять вимоги, які впливають на нашу здатність обчислювати найкращий шлях. Можливо, важливіше отримати будь-яку відповідь, ніж отримати найкращу відповідь. І все-таки за інших рівних умов коротший шлях кращий, ніж довший.

Загалом, обчислення частини шляху, близького до вихідної точки, важливіше шляху, близького до мети. Принцип негайного запуску: запустити пристрій якомога швидше, навіть по неоптимальному шляху, а потім [обчислити кращий шлях пізніше](http://theory.stanford.edu/~amitp/GameProgramming/MovingObstacles.html#recalculating-paths). У іграх в режимі реального часу затримка A\* часто важливіша, ніж його пропускна здатність.

Одиниці можуть бути запрограмовані слідувати або своїм інстинктам (простий рух), або їхнім мозку (заздалегідь розрахований шлях). Одиниці будуть слідувати своїм інстинктам, якщо мозок не скаже їм іншого. (Цей підхід застосовується в природі, а також у архітектурі роботів Родні Брука.) Замість того, щоб обчислювати всі шляхи одночасно, обмежте гру лише одним шляхом, кожним, двома або трьома ігровими циклами. Тоді нехай юніти починають ходити відповідно до інстинкту (який може просто рухатися прямолінійно до мети), а повертайтеся пізніше, щоб знайти їм шляхи. Цей підхід дозволяє вирівняти вартість пошуку шляхів, щоб воно не відбулося одразу.

**2.16 Завчасний вихід**

Можна рано вийти з головного циклу A\* і отримати частковий шлях. Зазвичай цикл виходить, коли знаходить вузол цілі. Однак у будь-який момент до цього він може повернути шлях до найкращого на даний момент вузла в OPEN. Цей вузол є нашим найкращим шансом досягти мети, тож це розумне місце, куди треба піти.

Кандидати для дострокового виходу включають вивчення певної кількості вузлів, витрачення певної кількості мілісекунд в алгоритмі A\* або вивчення вузла на деякій відстані від вихідного положення. При використанні зрощування контурів, зрощеному шляху слід надати меншу межу, ніж повний шлях.

**2.17 Груповий рух**

Запити на шлях не надходять рівномірно. Типовою ситуацією в стратегічній грі в режимі реального часу є те, що гравець обирає кілька одиниць і наказує їм рухатися до однієї мети. Це створює велике навантаження на систему пошуку шляхів.

У цій ситуації дуже ймовірно, що шлях, знайдений для одного, буде корисним для інших підрозділів. Одна ідея полягає в тому, щоб знайти шлях P від центру одиниць до центру пунктів призначення. Потім використаймо більшу частину цього шляху для всіх одиниць, але замінемо перші 10 кроків і останні 10 кроків шляхами, знайденими для кожного окремого блоку. Блок i отримає шлях від вихідного місця до , за яким слідує загальний шлях , а потім шлях від до пункту призначення.

Знайдені шляхи для кожного блоку є короткими (в середньому приблизно 10 кроків), і довгий шлях є спільним. Більшість шляхів знайдено лише один раз і поділено між усіма юнітами. Однак користувач може бути не вражений, якщо побачить, що всі блоки рухаються по одному шляху. Щоб покращити зовнішній вигляд системи, змусимо агрегати йти дещо іншими шляхами. Один із способів зробити це – змінити самі шляхи, вибравши сусідні місця.

Інший підхід полягає в тому, щоб поінформувати підрозділи один про одного (можливо, випадковим чином вибравши одиницю «керівника» або вибравши ту, яка найкраще відчуває, що відбувається), і знайти лише шлях для керівника. Потім використовючи [алгоритм збивання,](http://www.red3d.com/cwr/boids/) щоб змусити їх рухатися в групі.

Існують деякі [варіанти A\*,](http://en.wikipedia.org/wiki/Incremental_heuristic_search) які обробляють пункт призначення або оновлені знання про пункт призначення. Деякі з них можуть бути пристосовані для обробки декількох одиниць, що прямують до одного пункту призначення, виконавши A\* в зворотному напрямку (знаходячи шлях від пункту призначення до одиниці).

**2.18 Доопрацювання**

Якщо на карті мало перешкод, але натомість міститься місцевість із різними витратами, то початковий шлях можна обчислити, обробляючи місцевість як дешевшу за звичайну. Наприклад, якщо пасовища коштують 1, пагорби коштують 2, а гори коштують 3, тоді A\* розгляне можливість прогулянки 3 луками, щоб уникнути 1 гори. Натомість обчисліть початковий шлях, обробляючи пасовища як 1, пагорби як 1,1, а гори як 1,2. Тоді A\* витратить менше часу, намагаючись уникнути гір, і швидше знайде шлях. (Це наближує переваги [точної евристики](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#linear-exact-heuristic).) Як тільки шлях відомий, одиниця може почати рух, а цикл може продовжуватися. Коли доступний запасний процесор, обчисліть кращий шлях, використовуючи реальні витрати руху.

**2.19 Висновок за розділом**

Отже, найкращим вибором серед найпоширеніших алгоритмів буде саме А\*, так як він поєднує в собі всі найкращі риси інших алгоритмів, показує найкоротший шлях як Алгоритм Дейкстри, але не витрачає так багато часу на його пошук, так як використовує алгоритм Best-First Search для пришвидшення руху по прямих.

Алгоритм Дейкстри хороший в пошуку найкоротшого шляху, але він витрачає час на дослідження всіх напрямків, навіть безперспективних. Жадібний пошук досліджує перспективні напрямки, але може не знайти найкоротший шлях. Алгоритм A \* використовує і справжню відстань від початку, і оцінену відстань до цілі.

Отже, для нашого варыанту, де багато перешкод у вигляді стін, найкращим варіантом буде А\*

РОЗДІЛ 3.

ОПИС ТА РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ

“NUPP NAVIGATOR”

Як вже було зазначено у 1 розділі, проєкт буде побудовано на базі Unity3D, з використанням мови програмування С#, оскільки це є самим найкращим варіантом для данного типу проєкту на сьогоднішній день.

У другому розділі ми зупинилися на виборі алгоритму для знаходження шляху A\*, оскільки данний вибір поєднує всі найкращі риси інших алгоритмів, максимальна швидкість роботи з мінімальним навантаженням на пристрій, а оптимізація в данному випадку дуже важлива, аже розроблюємо додаток для телефону.

Для початку роботи над проєктом нам знадобиться таке програмне забезпечення, як:

* Unity3D;
* Visual Studio 2019;
* Adobe Photoshop;
* Adobe Illustrator.

Роботу над проєктом можна розбити на 3 основні частини

* **Розробка дизайну** (План Університету, UI/UX дизайн) за допомоги Adobe Photoshop та Adobe Illustrator;
* **розробка алгоритму** пошуку шляху, алгоритм написано на мові програмування C#, оскільки Unity3D підтримує тільки його;
* **розробка макету**, обєднуємо попередні 2 пункти у одному проєкті.

В такому порядку і будемо працювати, почнемо з розробки дизайну.

**3.1 Розробка дизайну**

Основними кольорами для плану університету було обрано нейтральні пастельно сірі кольори, масштаб приміщеннь максимально збережено, наскільки це було можливо, оскільки в майбутньому буде необхідно синхронізувати данний план з сіткою алгоритму А\*, через це є невеликі розбіжності.

Використовувались такі інструменти фотошоу, як:

* Move(Переміщення);
* Brush tool(Кісточка) - Для промальовування карти;
* Marquee Tool (Виділення об’єктів різної форми).

MOVE – пересування об'єктів по області. Окрім того, коли затиснути клавішу CTRL і натиснути по об'єкту, тоді активується шар, обраний об’єкт на якому розташований[23].



Рисунок 3.1.1 – Інструмент Move

Ще одна функція «Переміщення» – вирівнює об'єкти (центри або краї) відносно одне одного, фонового полотна або виділеної частини полотна (області). (Див. Рисунок 3.1.2)[23].



Рисунок 3.1.2

Marquee Tool – В розділ виділення входять «Овальний область», «Прямокутна область», «Область (Горизонтальна)», «Область (Вертикальна)» (Рисунок 3.1.3).

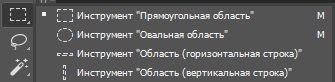


Рисунок 3.1.3 – Інструмент переміщення

Brush tool – Інструмент пензля використовується для малювання штрихів над зображенням та шарами.

Brush tool можна використовувати для таких цілей:

* Легко імітувати досить багато пензлів із реального життя, як пастелі, так і мокрі щітки;
* створювати текстури та візерунки, такі як сніг або пісок, з невеликим вмінням;
* змінювати кольори всіх або частин зображення;
* створювати картинки з нуля, використовуючи графічний планшет.

Звичайно, більше, що можна створити за допомогою пензлика. Насправді, одна з речей, що робить інструмент пензля справді потужним, – це кількість налаштовуваності, яку він забезпечує. Ви можете завантажити додаткові пензлі в Інтернеті або навіть створити власні. Що дозволяє робити необмежену кількість справ, і це за допомогою лише одного інструменту[25].

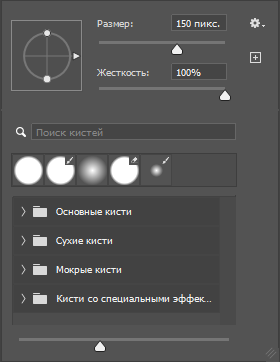


Рисунок 3.1.4 – Інструмент пензля

При розробці даного проєкту ми будемо використовувати восновному тільки ці інструменти фотошопу, адже задача намалювати план не є занадто складною, і не потребує використання більш складніх, та сильних інструментив фотошопу, що мають більший функціонал (Див. Рисунку 3.1.5).

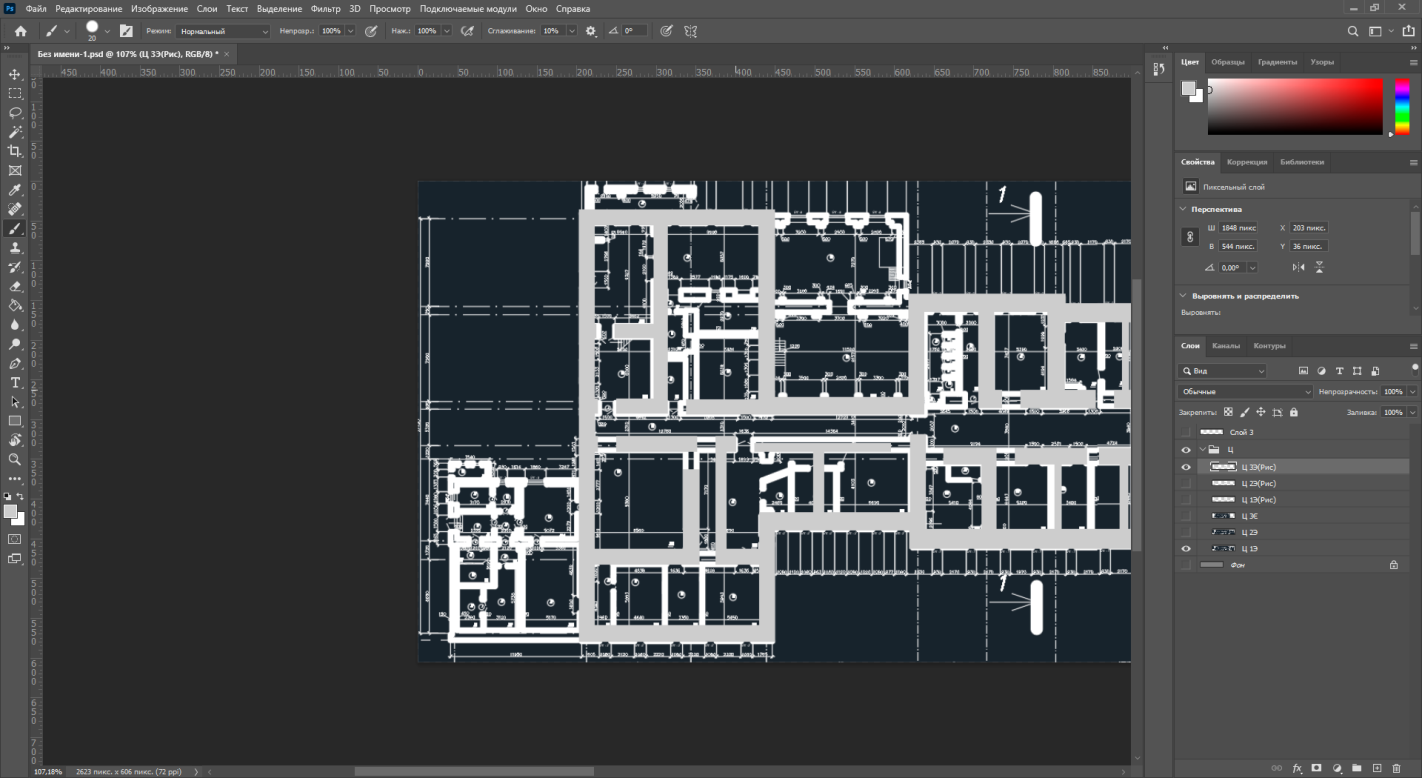
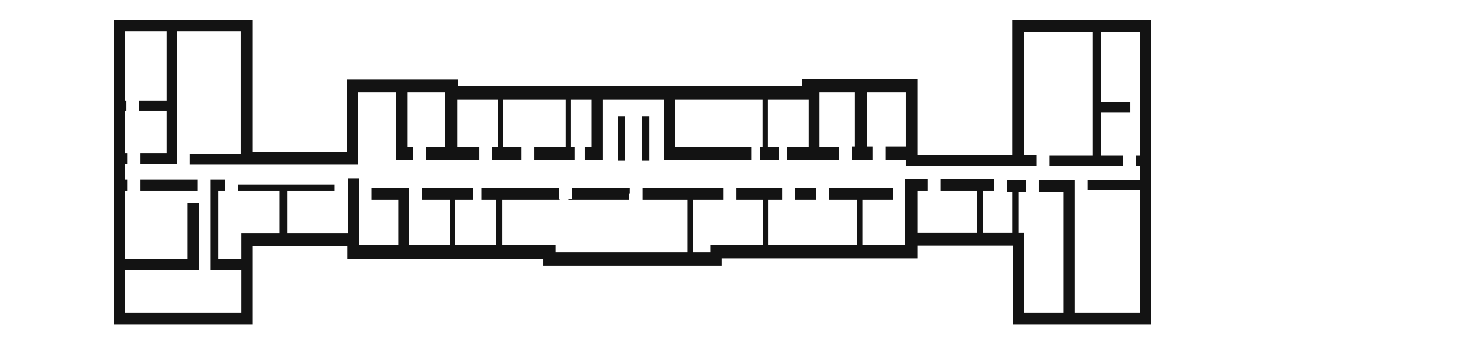
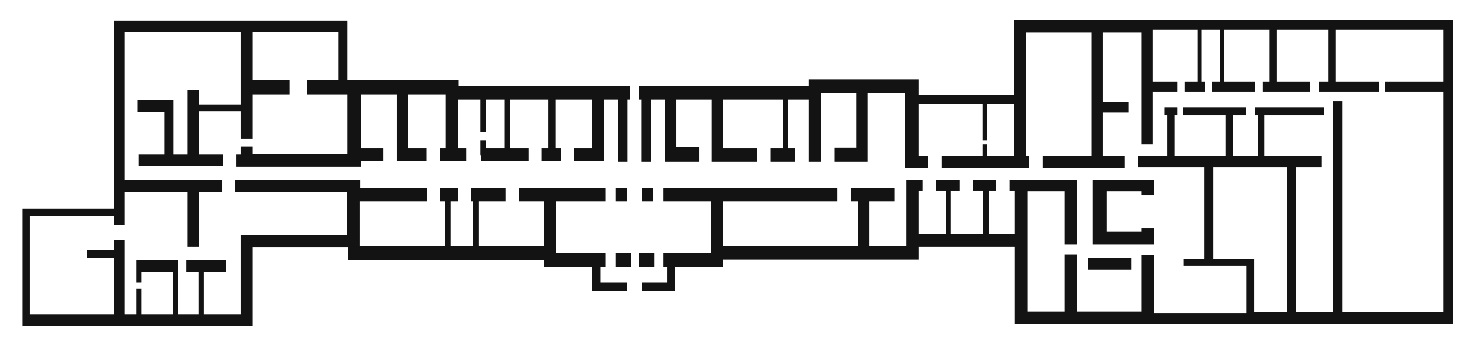


Рисунок 3.1.5

По завершенню промальовки карти ми маємо 3 зображення(Див. Рисунок 3.1.6, Рисунок 3.1.7, Рисунок 3.1.8), обов’язково в PNG форматі, оскульки нам необхідно мати прозори й задній фон, і твльуи данний фортам має підтримку альфаканалу (Колір змінив, щоб будо краще видно на білому фоні)[25].

Рисунок 3.1.6 – Перший поверх

Рисунок 3.1.6 – Другий поверх

 Рисунок 3.1.8 – Третій поверх

Також необхідно промалювати всі елементи UX/UI дизайну, та стрілки, що будуть показувати шлях. Намальовані у Векторі, та експортовані у PNG форматі (див Рисунок 3.1.9).

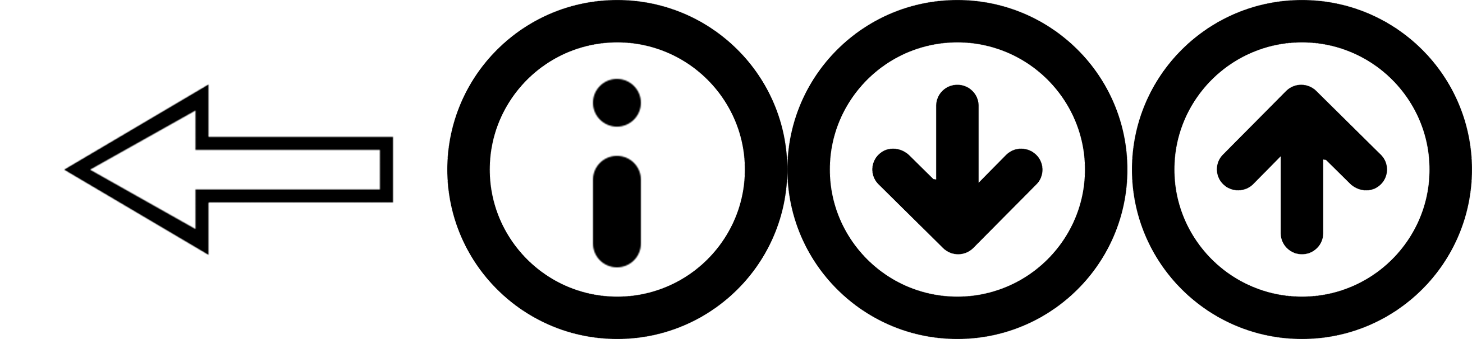


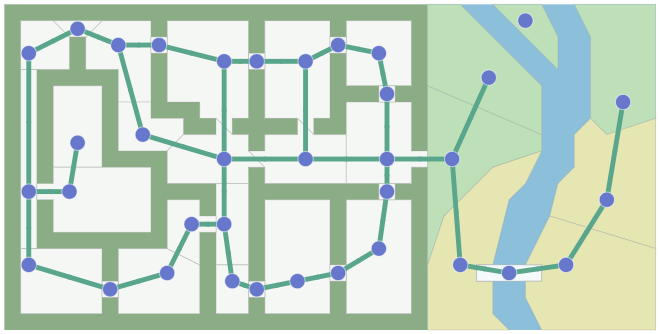
Рисунок 3.1.9 (Елементи UI)

* 1. **Розробка алгоритму**

Проблема, яку ми намагаємось вирішити, полягає в тому, щоб віднайти шлях від початкової точки до мети. Pathfinding вирішує проблему, допомогаючи знайти хороший шлях від початкової точки до мети, уникаючи перешкод, а також для зниження витрат (час, відстань, і т.д.). З одного боку, вишуканий пошук шляху у поєднанні з тривіальним алгоритмом руху знайде шлях, об’єкт почне рухатися, не зважаючи на все інше. З іншого боку, система, що призначена лише для руху, не буде шукати шлях вперед (натомість початковий «шлях» буде прямою лінією), а замість цого буде робити по одному кроку, вираховуючи місцеве середовище в кожній точці. Найкращі результати досягаються використанням як алгоритмів пошуку шляху, так і руху.

Щоб знайти цей шлях, ми можемо використовувати алгоритм пошуку графіків, який працює, коли карта представлена у вигляді графіка. **A\*** – популярний вибір для пошуку по графіку. **Breadth First Search** – найпростіший з алгоритмів пошуку графіків, тож давайте почнемо там, і ми пройдемо шлях до A\*.

**Вхідні дані:** Алгоритми пошуку графіків, включаючи A\*, бере “графік” як вхідні дані. Графік – це набір розташувань (“вузлів”) та з’єднань (“країв”) між ними Рис 3.2.1.

Рисунок 3.2.1 – Візуалізація графіку А\*

A\* не бачить нічого іншого. Він бачить лише графік. Він не знає, чи є щось у приміщенні чи на відкритому повітрі, чи це кімната чи дверний отвір, чи велика площа. Він бачить лише свій графік.

**Результат:** Шлях, знайдений A\* зроблений з графових вузлів та ребер. Ребра – абстрактні математичні поняття. Знак \* скаже вам переїхати з одного місця в інше, але не скаже, як. Необхідно пам’ятати, що воно нічого не знає про кімнати чи двері. Все, що він бачить – це графік. Вам доведеться вирішити, чи означає край графіка, повернутий символом A\*, перехід від плитки до плитки або пряму лінію, чи відкриття дверей.

Ключова ідея всіх цих алгоритмів полягає в тому, що ми відстежуємо кільце, що розширюється, зване кордоном. У сітці цей процес іноді називають «заливкою», але той же метод працює і не для сіток. Переглянемо це біль детально. Плитка нумерується в тому порядку, в якому ми її відвідуємо. Покроково переглянемо процес розширення у презентації.

Результатом виконання дипломної роботи маємо додаток, для ОС Android, написаниц на базі Unity3D, та C#, дизайн розроблено з нуля у таких програмах як Adobe Photoshop 2020, та Adobe Illustrator, автоматични пошук шляху основано на досить розповсюдженному для ігровій сфері рішенні, такому як алгорим пошуку шляху А\*.

Інтерфейс головного меню у мінімалісьичному Flat-дизайні, з мінімальним зайвим контентом. Дизайно розроблено у пастельних сірих тона, котрі не перевантажують зір(Рис. 3.2.2).

Користувач має можливість ввести номер необхідної аудиторіх, та корпус, у котрому ця аудиторія знаходиться. Елементи інтерфейсу самого навігаційного меню включають в себе:

* Кнопки переходу між поверхами;
* кнопку повернення до меню вводу номера аудиторії(Головного меню);
* кнопку відображення інформації про аудиторію.



Рисунок 3.2.2 – Інтерфейс головного меню

Після заповнення текстового поля номером аудиторії, та вибором потрібного корпусу необхідно натиснути кнопку “Знайти”, і перед нами з’явиться карта університету з прокладеним шляхом до аудиторії.

Всі аудиторії підписані, є можливість прочитати більш детальну інформацію про аужиторію, натиснувши кнопку “i”(Рис. 3.2.3).

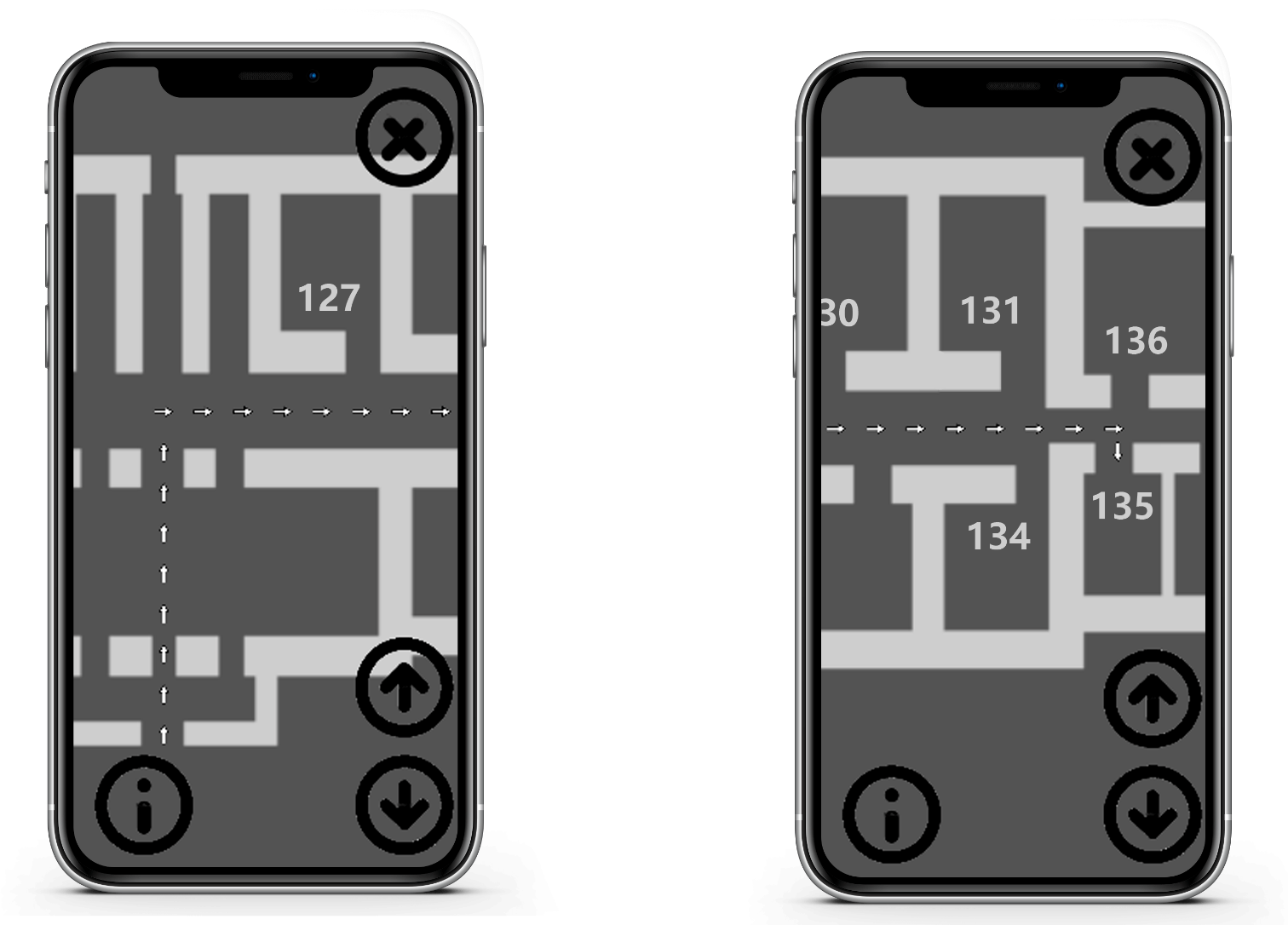


Рисунок 3.2.3 – Приклад роботи навігатору

**ВИСНОВКИ**

В даній дипломній роботі було проаналізовано вихідні документи навчального процесу НУПП стосовно навчальних планів та педагогічного навантаження викладачів, стандарти та вимоги, що пред'являються до їх складання.

За допомогою проведеного аналізу розроблено архітектуру та створено додаток на Android для пошуку шляху у приміщенні Університету на основі Unity3D та C#, Пошук шлху виконуться завдяки вдосконаленому алгоритму А\*.

Створена програма призначена для використання студентами університету всіх кафедр, та спеціальностей. Вона дозволяє автоматично знаходити шлях від входу до потрібної аудиторії, що дозволяє значно зменшити затрачений час, на пошук необхідної, особливо на початку навання.

Пошук шляху побудовано на основі розповсюдженого алгоритму А\*, який дає можливість без зайвих втрат часу знайти найоптимальніший шлях до цілі, обходячи всі перешкоди, великою перевагою в данному випадку є те, що алгоритм може враховувати вартість шляху, що дозволяє мінімізувати нам пересування по сходам, чи наприклад вихід навулицю.

Наступним кроком у розвитку системи навігації має бути розширення карти для всіх корпусів університету, а також створення можливості прокладання шляшу від однієї аудиторії до іншої.

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. How to Make an Android App Using Unity [Електронний ресурс] – Режим доступу до ресурсу: <https://www.instructables.com/How-to-Make-an-Android-App-Using-Unity/>.
2. Unity3D [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/ru/products.
3. Unity (игровой движок) [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Unity\_(%D0%B8%D0%B3%D1%80%D0%BE%D0%B2%D0%BE%D0%B9\_%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA).
4. Программирование игры с Unity: руководство для начинающих [Електронний ресурс] – Режим доступу до ресурсу: http://helpexe.ru/programmirovanie/programmirovanie-igry-s-unity-rukovodstvo-dlja.
5. Создаем 2D-игру на Unity: инструкция для новичка [Електронний ресурс] – Режим доступу до ресурсу: https://proglib.io/p/sozdaem-2d-igru-na-unity-instrukciya-dlya-novichka-2020-09-01.
6. 2D ИГРА НА UNITY. ПОДРОБНОЕ РУКОВОДСТВО. ЧАСТЬ 1 [Електронний ресурс] – Режим доступу до ресурсу: http://websketches.ru/blog/2d-igra-na-unity-podrobnoye-rukovodstvo-p1.
7. Введение в алгоритм A\* [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/331192/.
8. Алгоритм A\* [Електронний ресурс] – Режим доступу до ресурсу: https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC\_A\*.
9. A\* [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/A\*.
10. Алгоритм Дейкстры [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC\_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B.
11. Алгоритм Дейкстры. Поиск оптимальных маршрутов на графе [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/111361/.

12. Граф (математика ) [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84\_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0).

13. Алгоритм пошуку A\* [Електронний ресурс] – Режим доступу до ресурсу:https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC\_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83\_A\*

14. Акушский И.Я.. Юдицкий Д.И. Машинная арифметика в остаточных классах. – М.: Сов. Радио, 1968. – 444 с.

15. Барсов В.И., Сорока Л.С., Краснобаев В.А., Хери Али Абдуллах. Модели и методы повышения отказоустойчивости и производительности управляющих вычислительных комплексов специализированных систем управления реального времени на основе применения непозиционных кодовых структур модулярной арифметики. Монография.–Х.: УИПА, 2008. 147с.

16 Евстигнеев В.Г. Сведо – Швед В.Н., Краснобаев В.А. Арифметические алгоритмы для q – й системы исчисления/Тем. Сб. Науч. Трудов ХАИ. 1982, № 4, с. 165 – 168.

17. Барсов В.И., Фурман И.А., Малиновский М.Л., Краснобаев В.А., Шевченко В.В. Модели и методы параллельной реализации логических операций в АСУ ТП: Монография.–Х.: МОН, УИПА, 2009. 140с.

1. Гольцман В.І. «MySQL 5.0. Библиотека программиста» / Гольцман В.І. – К. : «Питер», 2010. – 253 с
2. Евстигнеев В.Г. Сведо – Швед В.Н., Краснобаев В.А. Арифметические алгоритмы для q – й системы исчисления/Тем. Сб. Науч. Трудов ХАИ. 1982, № 4, с. 165 – 168.
3. Кузін А.В «Базы данных, 5-е издание» / Кузін А.В., Левонисова С.В. – К. : «Академия», 2012. – 317 с
4. Краснобаев В.А. Искуственный интелект и система исчисления в остаточных классах / Проблемы бионики. 1987, № 39, с. 40 – 46.
5. Adobe Illustrator [Електронний ресурс] – Режим доступу до ресурсу: https://www.adobe.com/ru/products/illustrator.html.
6. Посібник користувача Photoshop [Електронний ресурс] – Режим доступу до ресурсу: https://helpx.adobe.com/ua/photoshop/user-guide.html.
7. Как работать в «Фотошоп»: инструкция для начинающих [Електронний ресурс] – Режим доступу до ресурсу: https://timeweb.com/ru/community/articles/kak-rabotat-v-photoshop.
8. Как работать в «Фотошоп»: полное руководство для начинающих [Електронний ресурс] – Режим доступу до ресурсу: https://texterra.ru/blog/kak-rabotat-v-fotoshop-polnoe-rukovodstvo-dlya-nachinayushchikh.html.