УДК 519.62

DOi 10.30837/ bi.2021.2(97).06

І.В. Кириченко[1], В.Д. Рошка[2]

[1] кандидат технічних наук, старший викладач, кафедра програмної інженерії, Харківський Національний Університет Радіоелектроніки, Харків, Україна, iryna.kyrychenko@nure.ua, ORCID iD: 0000-0002-7686-6439

[2] студент бакалаврату, кафедра програмної інженерії, Харківський Національний Університет Радіоелектроніки, Харків, Україна, veronika.roshka@nure.ua, ORCID iD: 0000-0002-1704-9519

ПОРІВНЯННЯ ЕФЕКТИВНОСТІ АЛГОРИТМІВ ПОШУКУ ШЛЯХУ ПРИ РОЗРОБЦІ ІГРОВОГО ШТУЧНОГО ІНТЕЛЕКТУ

З точки зору ігор справжній ШІ далеко виходить за рамки вимог розважального програмного проекту. В іграх така міць не потрібна. Ігровий ШІ не повинен бути наділений почуттями та самосвідомістю, йому немає необхідності вчитися чогось за межами рамок ігрового процесу. Справжня мета ШІ в іграх полягає в імітації розумної поведінки та у наданні гравцеві переконливого, правдоподібного завдання. Щоб штучний інтелект міг приймати осмислені рішення, йому необхідно будь-яким чином сприймати середовище, в якому він знаходиться. У найпростіших системах таке сприйняття може обмежуватися простою перевіркою положення об'єкта гравця. У складних системах потрібно визначати основні показники і якості ігрового світу, наприклад можливі маршрути для пересування, наявність природних укриттів на території, області конфліктів.

При цьому розробникам необхідно вигадувати спосіб виявлення та визначення основних властивостей ігрового світу, важливих для ШІ системи. Наприклад, укриття на місцевості можуть бути визначені дизайнерами рівнів або заздалегідь обчислені при завантаженні або компіляції карти рівня. Деякі елементи необхідно обчислювати на льоту, наприклад, карти конфліктів і найближчі загрози.

Пошук шляху або pathfinder — визначення комп'ютером найоптимальнішого маршруту між двома точками. З ним часто можна зіткнутися при розробці ігор, наприклад, якщо в них є вороги, що вільно пересуваються. Потрібно не просто прагнемо знайти найкоротшу відстань, а також необхідно враховувати і тривалість руху. Для пошуку цього шляху можна використовувати алгоритм пошуку за графом, який застосовується, якщо карта є графом. A* часто використовується як алгоритм пошуку за графом. Пошук в ширину — це найпростіший алгоритм пошуку за графом, тому в цій статті почнемо розбиратися з нього і поступово перейдемо до A*.

Карти, на яких шляхи прокладаються за допомогою алгоритму «Зіткнутися і повернути», дозволяють пристосуватися до карт, що змінюються. Але у стратегічних іграх гравці не можуть чекати, поки їхні війська розберуться з прокладанням маршрутів. Крім того, карти шляхів можуть бути дуже великими, і на вибір правильного шляху на таких картах витрачатиметься дуже багато ресурсів. У таких ситуаціях на допомогу приходить алгоритм пошуку шляхів.

ІГРОВИЙ ШТУЧНИЙ ІНТЕЛЕКТ, ПОШУК ШЛЯХУ, ПОШУК В ШИРИНУ, АЛГОРИТМ ДЕЙКСТРИ, ЕВРИСТИЧНИЙ АЛГОРИТМ, ЖАДКОВИЙ ПОШУК, АЛГОРИТМ A*

Iryna Kyrychenko, Veronika Roshka. Comparison of the efficiency of path finding algorithms in the development of game artificial intelligence. In terms of games, true AI goes far beyond the requirements of an entertainment software project. In games, this power is not needed. Game AI does not have to be endowed with feelings and self-awareness; it does not need to learn anything outside of the gameplay. The real purpose of AI in games is to mimic intelligent behavior and to present the player with a compelling, believable challenge. In order for artificial intelligence to make meaningful decisions, it needs to somehow perceive the environment in which it is located. In simple systems, this perception can be limited to simply checking the position of the player's object. In more complex systems, it is required to determine the main characteristics and properties of the game world, for example, possible routes for movement, the presence of natural shelters on the ground, and areas of conflict.

At the same time, developers need to come up with a way to identify and define the main properties of the game world that are important for the AI system. For example, terrain cover can be predefined by the level designers or calculated in advance when loading or compiling a level map. Some elements need to be calculated on the fly, such as conflict maps and nearby threats.

Pathfinder – the computer determines the best route between two points. You can often encounter it when developing games, for example, if there are free-moving enemies in them. It is necessary not only to find the shortest distance, but also to take into account the duration of the movement. To find this path, you can use the graph search algorithm, which is applicable if the map is a graph. A * is often used as a graph search algorithm. Breadth First Search is the simplest graph search algorithm, so in this article we will start to understand it and gradually move on to A*.

Bump-and-turn maps allow you to adapt to changing maps. But in strategy games, players can't wait for their troops to figure out the routes. In addition, path maps can be very large, and choosing the right path on such maps will consume a lot of resources. In such situations, the pathfinding algorithm comes to the rescue.

GAME ARTIFICIAL INTELLIGENCE, WAY SEARCH, BREADTH FIRST SEARCH, DIJKSTRA'S ALGORITHM, HEURISTIC ALGORITHM, GREEDY SEARCH, A * ALGORITHM

Кириченко И. В., Рошка В.Д. Сравнение эффективности алгоритмов поиска пути при разработке игрового искусственного интеллекта. С точки зрения игр подлинный ИИ далеко выходит за рамки требований развлекательного программного проекта. В играх такая мощь не нужна. Игровой ИИ не должен быть наделен чувствами и самосознанием, ему нет необходимости обучаться чему-либо за пределами рамок игрового процесса. Подлинная цель ИИ в играх состоит в имитации разумного поведения и в предоставлении игроку убедительной, правдоподобной задачи. Чтобы искусственный интеллект мог принимать осмысленные решения, ему необходимо каким-либо образом воспринимать среду, в которой он находится. В простых системах такое восприятие может ограничиваться простой проверкой положения объекта игрока. В более сложных системах требуется определять основные характеристики и свойства игрового мира, например возможные маршруты для передвижения, наличие естественных укрытий на местности, области конфликтов.

При этом разработчикам необходимо придумывать способ выявления и определения основных свойств игрового мира, важных для системы ИИ. Например, укрытия на местности могут быть заранее определены дизайнерами уровней или заранее вычислены при загрузке или компиляции карты уровня. Некоторые элементы необходимо вычислять на лету, например карты конфликтов и ближайшие угрозы.

Поиск пути или pathfinder — определение компьютером самого оптимального маршрута между двумя точками. С ним часто можно столкнуться при разработке игр, например, если в них есть свободно передвигающиеся враги. Необходимо не просто найти кратчайшее расстояние, также нужно учесть и длительность движения. Для поиска этого пути можно использовать алгоритм поиска по графу, который применим, если карта представляет собой граф. A* часто используется в качестве алгоритма поиска по графу. Поиск в ширину — это простейший из алгоритмов поиска по графу, поэтому в этой статье начнем разбираться с него и постепенно перейдём к A*.

Карты, на которых пути прокладываются с помощью алгоритма «Столкнуться и повернуть», позволяют приспособиться к изменяющимся картам. Но в стратегических играх игроки не могут ждать, пока их войска разберутся с прокладыванием маршрутов. Кроме того, карты путей могут быть очень большими, и на выбор правильного пути на таких картах будет расходоваться очень много ресурсов. В таких ситуациях на помощь приходит алгоритм поиска путей.

ИГРОВОЙ ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, ПОИСК ПУТИ, ПОИСК В ШИРИНУ, АЛГОРИТМ ДЕЙКСТРЫ, ЭВРИСТИЧЕСКИЙ АЛГОРИТМ, ЖАДНЫЙ ПОИСК, АЛГОРИТМ A*

Вступ

Комп'ютерним NPC не можна просто сказати: «Гей ти, іди до вежі!», як людям. Їм потрібно прописати точний набір команд (іди один метр вліво, потім метр вперед і т.д.) або вказати шлях з проміжних точок, щоб вони успішно дісталися точки А до точки В. Саме знаходженням цих проміжних точок і займається алгоритм пошуку шляху. Але цим завдання не обмежуються. Додатково він може визначати відстань до мети і те, чи взагалі можливо дістатися до якоїсь точки.

Є такий вид ігор-головоломок – лабіринт Мінотавра. Гравця вміщують на сітчасте поле зі стінками. Одна із клітин поля є виходом, а на іншій стоїть монстр. Завдання гравця – дістатися до виходу раніше, ніж монстр добереться до нього. Гравець і монстр ходять по черзі на скільки клітин, але монстр завжди переміщується швидше гравця.

Гравець би постійно програвав, але монстр має мінус: він хоч і швидкий, але тупий. Він завжди намагається дістатися гравця найкоротшим способом і не звертає уваги на те, що в лабіринті є стіни. Якщо монстру на заваді зустрічається стіна, він просто стоїть і сумно зітхає перед нею, не намагаючись обійти.

У таких іграх дурість глупість виправдана, оскільки вона є частиною геймплей головоломки. В інших іграх користувач очікує, що ворог почне шукати шляхи обходу, а не застрягатиме в камінні та деревах.

Завдання пошуку шляху складається з двох етапів: адаптування ігрового світу до математичної моделі та пошук у цій моделі шляху між двома точками.

1. Адаптування ігрового світу

Комп'ютер не може просто подивитися на камінь та сказати, що це камінь. Тому йому треба описати ігровий світ у вигляді чисел і вибрати набір ознак, якими буде визначатися, що між двома точками можна пройти.

Більшість алгоритмів будується на графах. Тому шляхи проходу ігровим світом перетворюють на графи. Як точки-кружочки зазвичай беруть мінімальну площу світу, в якій пересування від одного краю до другого відбувається миттєво (або за досить маленький час, щоб вважати його миттєвим). Інакше висловлюючись, у іграх ці кружечки — це позиції персонажа. Ребра ж позначають той факт, що персонаж із першого шматка світу може перейти на другий. Приклад зображено на рисунку 1.

Одна з найпростіших варіацій графа — ігрова сітка, де кружальцями є клітини, а ребра проводяться між будь-якими сусідніми клітинами, вільними від перешкод.

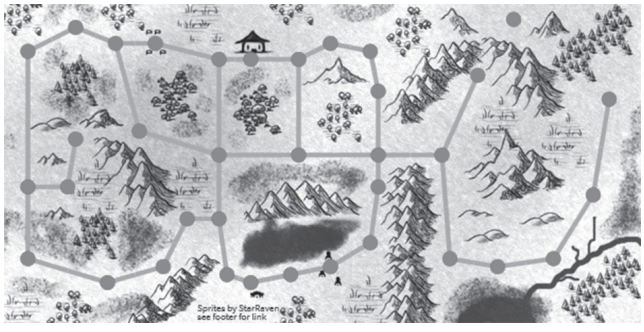


Рис. 1. Приклад побудови графа

Найпростіше розібратися в цьому на прикладі покрокових ігор, у яких сітку видно неозброєним оком, як Sneaky-Sneaky або Return of the Necrodancer (див. рис. 2).

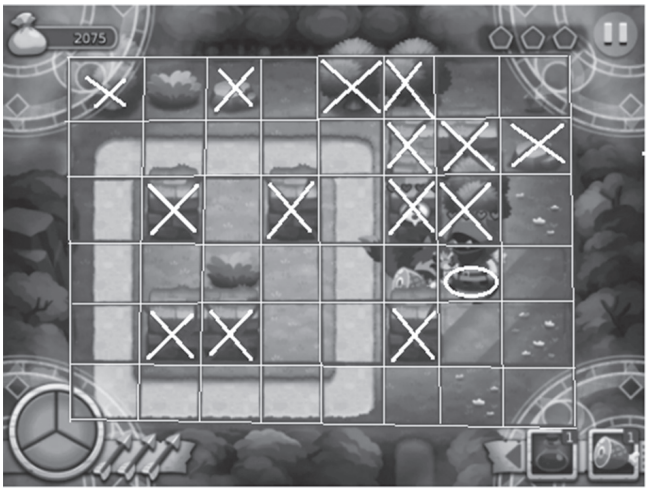


Рис. 2. Зразкова сітка у грі Sneaky-Sneaky

Клітини, на яких стоять непереборні перешкоди, позначаються хрестиком, а місця, що проходять – порожнечею. Або якщо говорити мовою програмування, клітини позначаються 0 або 1, де 0 означає, що пройти не можна, а 1 – що пройти можна. Більш складні ігри на кшталт Heroes of Might and Magic III відрізняються лише тим, що їхня сітка малюється більш довільно, а клітини, які розташовані на різних типах землі, забирають різну кількість кроків.

При роботі з тривимірними світами можна використовувати двомірну карту проекції ландшафту із зазначенням висот у точках. І тут кожному осередку написана її висота, а прохідність визначається через різницю висот сусідніх клітин.

Найчастіше адаптація ігрового світу – це найскладніша частина, тому що алгоритми пошуку оптимальних маршрутів винайдено давно, можна навіть знайти готовий код для них. Тому розробники впроваджують один з алгоритмів і потім експериментують з різними моделями та варіаціями порівнянь двох шляхів, або комбінують різні алгоритми у спробах досягти найкращого та найменш витратного результату.

2. Алгоритми пошуку шляху

Як би людина шукала вихід, якби її раптово висадили посеред лабіринту? Наприклад, він міг би скористатися правилом «однієї руки», відзначати обстеженні шляхи мотузкою або крейдою, доки не знайде вихід. Більшість алгоритмів діють за схожою схемою, за винятком того, що комп'ютер може розмножуватися та «обмацувати» одночасно кілька шляхів. Розрізняються алгоритми точністю і швидкістю отримання результату.

Пошук у ширину виконує дослідження поступово в усіх напрямках. Приклад дивись на рис. 3*а*. Це неймовірно корисний алгоритм, як для звичайного пошуку шляху, але й процедурної генерації карт, пошуку шляхів течії, карт відстаней та інших типів аналізу карт.

Алгоритм Дейкстри (також званий пошуком із рівномірною вартістю) дозволяє нам ставити пріоритети дослідження шляхів. Замість рівномірного дослідження всіх можливих шляхів він віддає перевагу шляхам із низькою вартістю. Приклад дивись на рис. 3*б*. Ми можемо задати зменшені витрати, щоб алгоритм рухався дорогами, підвищену вартість, щоб уникати лісів та ворогів, та багато іншого. Коли вартість руху може бути різною, ми використовуємо його замість пошуку у ширину.

A* – це модифікація алгоритму Дейкстри, оптимізована для єдиної кінцевої точки. Алгоритм Дейкстри може знаходити шляхи до всіх точок, A * знаходить шлях до однієї точки. Він віддає пріоритет шляхам, що ведуть ближче до мети (див. рис. 3*в*).

а *б* *в*

Рис. 3 (*а*, *б*, *в*). Графічне представлення роботи алгоритмів

Почнемо з найпростішого — пошуку у ширину, і будемо додавати функції, поступово перетворюючи його на A*.

3. Пошук у ширину

Ключова ідея всіх цих алгоритмів полягає в тому, що ми відстежуємо стан кільця, що розширюється, яке називається кордоном. У сітці цей процес іноді називається заливкою (flood fill), але та сама техніка застосовна і для карт без сіток.

Щоб це реалізувати, необхідно повторити ці кроки, поки кордон не стане порожнім:

1) Вибираємо та видаляємо крапку з кордону.

2) Позначаємо точку як відвідану, щоб знати, що не потрібно обробляти її повторно.

3) Розширюємо кордон, дивлячись на сусідів. Усіх сусідів, яких ми ще не бачили, додаємо до кордону.

Алгоритм описується лише в десяти рядках коду на Python:

```
frontier = Queue()
frontier.put(start )
visited = {}
visited[start] = True

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in visited:
            frontier.put(next)
            visited[next] = True
```

У цьому вся циклі полягає вся сутність алгоритмів пошуку за графом цієї статті, зокрема і A*. Але як знайти найкоротший шлях? Цикл насправді не створює шляхів, він просто каже нам, як відвідати всі точки на карті. Так вийшло тому, що пошук у ширину можна використовувати для набагато більшого, ніж просто пошук шляхів. У цій статті показується, як він застосовується в іграх tower defense, але його також можна використовувати в картах відстаней, у процедурній генерації карт та багато іншого. Однак тут ми хочемо використовувати його для пошуку шляхів, тому давайте змінимо цикл так, щоб відстежувати, звідки ми прийшли для кожної відвіданої точки, і перейменуємо visited в came_from:

```
frontier = Queue()
frontier.put(start )
came_from = {}
came_from[start] = None

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in came_from:
            frontier.put(next)
            came_from[next] = current
```

Тепер came_from для кожної точки вказує місце, з якого ми прийшли. Нам цього достатньо, щоб відтворити цілий шлях. Подивіться, як стрілки показують зворотний шлях до початкової позиції на рис. 4.

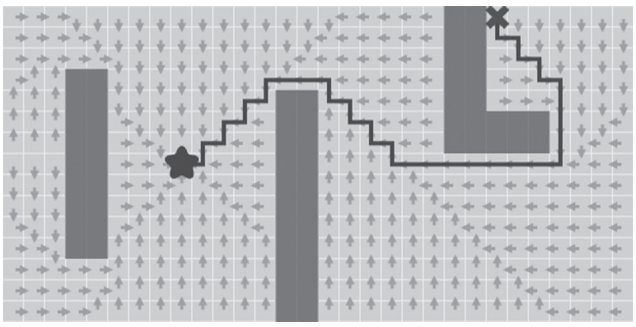


Рис. 4. Приклад реалізації

Код відтворення шляхів простий: слідуємо за стрілками назад від мети на початок. Шлях – це послідовність ребер, але іноді простіше зберігати лише вузли:

```
current = goal
path = [current]
while current != start:
    current = came_from[current]
    path.append(current)
path.append(start) # optional
path.reverse() # optional
```

Такий найпростіший алгоритм пошуку шляхів. Він працює не тільки в сітках, як показано вище, а й у будь-якій структурі графів. У підземеллі точки графа можуть бути кімнатами, а ребра – дверима між ними. У платформері вузли графа можуть бути локаціями, а ребра – можливими діями: переміститися вліво, вправо, підстрибнути, стрибнути вниз. У цілому нині можна сприймати граф як і дії, що змінюють стан.

Пошук у ширину на кожному N циклі знаходить всі точки, яких можна дійти за N кроків. Щоб отримати шлях, потрібно запам'ятовувати як перевірені точки, а ще й послідовність точок до них. Або хоча б попередню точку, з якої ми прийшли на цю, щоб кроками відновити маршрут.

Пошук у ширину добре працює, якщо перехід між сусідніми точками завжди займає однакову кількість часу (або іншого параметра, яким ми намагаємося мінімізувати шлях). Але якщо це не так, то він перетворюється на алгоритм Дейкстри.

4. Ранній вихід

Ми знайшли шляхи з однієї точки до всіх інших точок. Часто нам не потрібні всі шляхи, нам просто потрібен шлях між двома точками. Ми можемо припинити розширювати кордон, як тільки знайдемо нашу мету. Подивіться, як межа перестає розширюватись після знаходження мети (див. рис. 5).

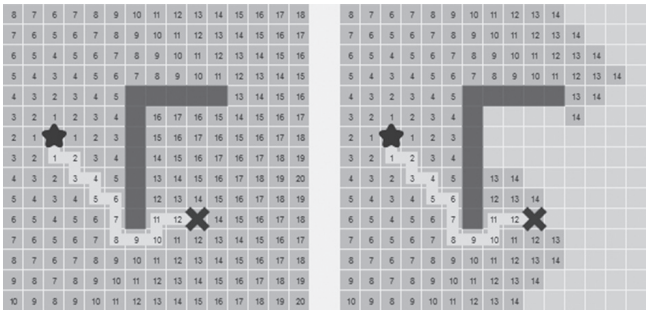


Рис. 5. Перевага раннього виходу

Код досить прямолінійний:

```
frontier = Queue()
frontier.put(start )
came_from = {}
came_from[start] = None
while not frontier.empty():
    current = frontier.get()
    if current == goal:
        break
    for next in graph.neighbors(current):
        if next not in came_from:
            frontier.put(next)
            came_from[next] = current
```

5. Вартість переміщення. Алгоритм Дейкстри

Алгоритм Дейкстри працює з моделями, де відстані між точками можуть бути різними. Наприклад, у Heroes of Might and Magic III прохід по снігу витрачає на 50% більше очок переміщення, так що умовно можна порахувати, що прохід по снігу займає 1.5 кроку замість 1 по звичайній землі. У цьому випадку обхід снігу може бути швидшим, ніж прохід безпосередньо по ньому, хоч візуально і буде пройдено більшу відстань.

Ще одним прикладом є діагональний рух у сітці, який коштує більше, ніж рух по осях. Нам потрібно, щоби пошук шляху враховував цю вартість. Давайте порівняємо кількість кроків від початку (ліворуч) з відстанню від початку (справа), зображених на рис. 6.

Рис. 6. Порівняння кількості кроків із відстанню

Для цього нам потрібен алгоритм Дейкстри (також званий пошуком із рівномірною вартістю). Від пошуку у ширину алгоритм Дейкстри відрізняє пара моментів. Окрім збереження вже перевірених точок, зберігається ще й кількість кроків, витрачених на те, щоби дістатися до них. Крапки виключаються з подальшої перевірки не тоді, коли були вже перевірені, а лише у випадку, якщо попередній знайдений шлях до неї займав меншу кількість кроків. Крапки у списку розглядаються не по порядку, спочатку вибираються ті, до яких менше йти. Нам потрібно відслідковувати вартість руху, тому додаємо нову змінну cost_so_far, щоб слідкувати за загальною вартістю руху початкової точки. Оцінюючи точок нам потрібно враховувати вартість пересування. Давайте перетворимо нашу чергу на чергу з пріоритетами. Менш очевидно те, що у нас може вийти так, що одна точка відвідується кілька разів із різною вартістю, тому потрібно трохи поміняти логіку. Замість додавання точки до кордону у випадку, коли точку жодного разу не відвідували, ми додаємо її, якщо новий шлях до точки кращий, ніж найкращий попередній шлях.

```
  frontier = PriorityQueue()
  frontier.put(start, 0)
  came_from = {}
  cost_so_far = {}
  came_from[start] = None
  cost_so_far[start] = 0

  while not frontier.empty():
    current = frontier.get()
    if current == goal:
      break
    for next in graph.neighbors(current):
      new_cost = cost_so_far[current] +
graph.cost(current, next)
      if next not in cost_so_far or new_cost <
cost_so_far[next]:
        cost_so_far[next] = new_cost
        priority = new_cost
        frontier.put(next, priority)
        came_from[next] = current
```

Використання черги з пріоритетами замість звичайної черги змінює спосіб розширення кордону. Контурні лінії дозволяють це побачити.На рисунку 7 представлені контури, ліворуч – пошук у ширину, праворуч – алгоритм Дейкстри. Кордон розширюється повільніше через ліси, і пошук найкоротшого шляху виконується навколо центрального лісу, а не через нього:

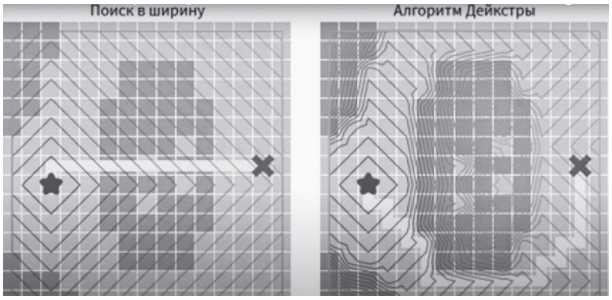


Рис. 7. Розширення кордону

Вартість руху, що відрізняється від 1, дозволяє нам досліджувати цікавіші графи, а не лише сітки.

6. Евристичний пошук

У пошуку в ширину та алгоритмі Дейкстри кордон розширюється у всіх напрямках. Це логічний вибір, якщо ви шукаєте шлях до всіх точок або до безлічі точок. Однак, зазвичай пошук виконується тільки для однієї точки. Давайте зробимо так, щоб межа розширювалася до мети більше, ніж у інших напрямках. По-перше, визначимо евристичну функцію, яка повідомляє нам, наскільки ми близькі до мети:

```
def heuristic(a, b):
  # Manhattan distance on a square grid
  return abs(a.x - b.x) + abs(a.y - b.y)
```

В алгоритмі Дейкстри порядку черги з пріоритетами ми використовували відстань від початку. У жадібному пошуку за першим найкращим збігом для порядку черги з пріоритетами ми замість цього використовуємо оцінену відстань до мети. Крапка, найближча до мети, буде вивчена першою. У коді використовується черга з пріоритетами пошуку в ширину, але не застосовується cost_so_far з алгоритму Дейкстри:

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
  current = frontier.get()

  if current == goal:
    break

  for next in graph.neighbors(current):
    if next not in came_from:
      priority = heuristic(goal, next)
      frontier.put(next, priority)
      came_from[next] = current
```

Результат виконання представлений на рис. 8, 9.

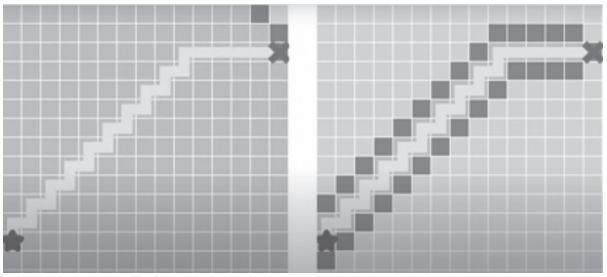


Рис. 8. Результат виконання коду

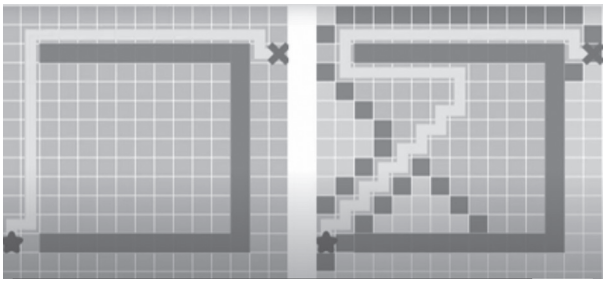


Рис. 9. Результат на більш складній карті

Ці шляхи не є найкоротшими. Отже, цей алгоритм працює швидше, коли перешкод не дуже багато, але шляхи не надто оптимальні. Чи можна це виправити? Звичайно.

7. Алгоритм A*

Алгоритм Дейкстри гарний у пошуку найкоротшого шляху, але він витрачає час на дослідження всіх напрямків, навіть безперспективних. Жадібний пошук досліджує перспективні напрями, але може знайти найкоротший шлях. Алгоритм A* використовує і справжнє відстань від початку, і оцінене відстань до мети.

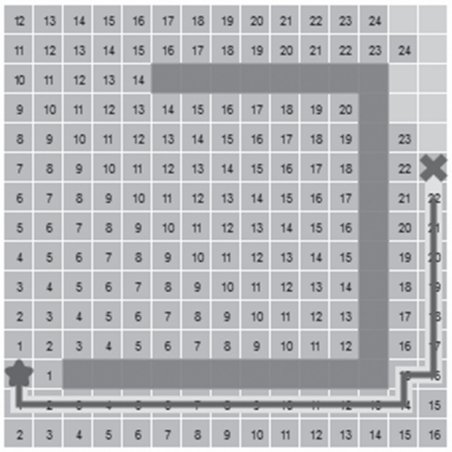
Код дуже схожий на алгоритм Дейкстри:

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
  current = frontier.get()

  if current == goal:
    break

  for next in graph.neighbors(current):
    new_cost = cost_so_far[current] + graph.cost(current, next)
    if next not in cost_so_far or new_cost < cost_so_far[next]:
      cost_so_far[next] = new_cost
      priority = new_cost + heuristic(goal, next)
      frontier.put(next, priority)
      came_from[next] = current
```

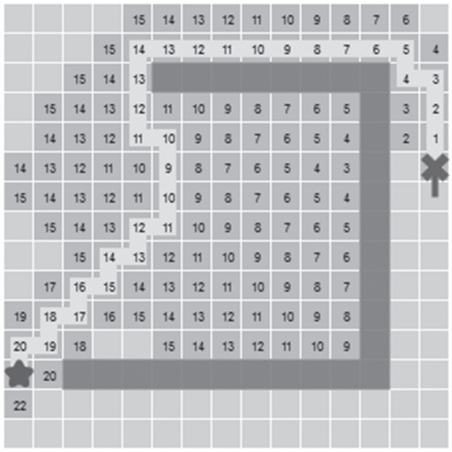
Порівняємо алгоритми. Алгоритм Дейкстри обчислює відстань від початкової точки (див. рис. 10а). Жадібний пошук за першим найкращим збігом оцінює відстань до точки мети (див. рис. 10*б*). A* використовує суму цих двох відстаней(див. рис. 10*в*).

Як видно з ілюстрацій, жадібний пошук знаходить правильну відповідь, A* теж його знаходить, досліджуючи ту саму область. Коли жадібний пошук по першому найкращому знаходить неправильну відповідь (довший шлях), A* знаходить правильний, як і алгоритм Дейкстри, але все одно досліджує менше, ніж алгоритм Дейкстри.

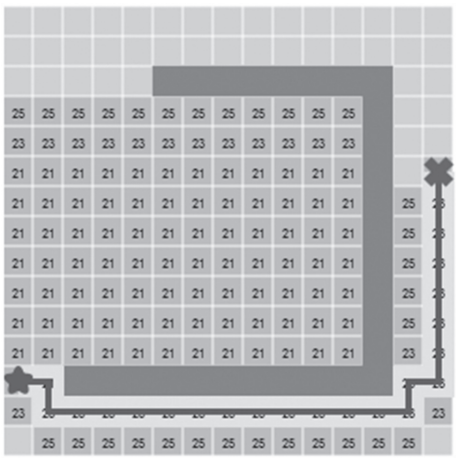
A* бере найкраще від двох алгоритмів. Оскільки евристика не оцінює відстані повторно, A* не використовує евристику для пошуку відповіді. Він знаходить оптимальний шлях, як алгоритм Дейкстри.



а) Алгоритм Дейкстри



б) Жадібний пошук



в) A* пошук

Рис. 10 (а, б, в) – Порівняння алгоритмів

A* використовує евристику для зміни порядку вузлів, щоб підвищити ймовірність раннього знаходження вузла мети.

Алгоритм A* намагається всидіти на обох стільцях одразу: знайти найкоротшу відстань, але зробити це за менший час, ніж алгоритм Дейкстри. Тому наступна точка на розгляд вибирається за мінімальною сумою відстаней до початку та до кінця шляху. Це єдина відмінність від попередніх алгоритмів.

Висновки

З вищесказаного виникає питання. Який алгоритм варто використовувати для пошуку шляхів на карті?

Якщо вам потрібно знайти шляхи з або до всіх точок, використовуйте пошук у ширину або алгоритму Дейкстри. Використовуйте пошук у ширину, якщо вартість руху однакова. Використовуйте алгоритм Дейкстри, якщо вартість руху змінюється.

Якщо потрібно знайти шляхи до однієї точки, використовуйте жадібний пошук за найкращим першим або A *. Найчастіше варто віддати перевагу A*. Коли є спокуса використовувати жадібний пошук, подумайте над застосуванням A* з «неприпустимою» евристикою.

А що щодо оптимальних шляхів? Пошук у ширину та алгоритм Дейкстри гарантовано знайдуть найкоротший шлях по графу. Жадібний пошук не обов'язково його знайде. A* гарантовано знайде найкоротший шлях, якщо евристика ніколи не більша за справжню відстань. Коли евристика стає меншою, A* перетворюється на алгоритм Дейкстри. Коли евристика стає більшою, A* перетворюється на жадібний пошук за найкращим першим збігом.

А як щодо продуктивності? Найкраще усунути непотрібні точки графа. Якщо ви використовуєте сітку, прочитайте це. Зменшення розміру графа допомагає всім алгоритмам пошуку за графами. Після цього використовуйте найпростіший з можливих алгоритмів. Прості черги виконуються швидше. Жадібний пошук зазвичай виконується швидше, ніж алгоритм Дейкстри, але не забезпечує оптимальних шляхів. Для більшості завдань пошуку шляхів оптимальним вибором є A*.

А що щодо використання не на картах? У статті використовувалися карти тому, що простіше пояснити роботу алгоритму. Однак ці алгоритми пошуку за графами можна використовувати на будь-яких графах, не тільки на ігрових картах, у статті представлений код алгоритму у вигляді, що не залежить від двовимірних сіток. Вартість руху на картах перетворюється на довільні ваги ребер графа. Евристики перенести на довільні карти непросто, необхідно створювати евристику кожного типу графа. Для плоских карток хорошим вибором будуть відстані, тому тут ми використовували їх.

Не забувайте, що пошук за графами – це лише одна частина того, що вам потрібне. Сам по собі A* не обробляє такі аспекти, як спільний рух, переміщення перешкод, зміна карти, оцінка небезпечних областей, формації, радіуси повороту, розміри об'єктів, анімацію, згладжування шляхів та багато іншого.

Є й інші алгоритми, що дають переваги за різних умов. Самі алгоритми пошуку шляху досить прості, але вони працюють для сферичних ідеальних умов у вакуумі. У реальних іграх є багато нюансів, які можуть ускладнити алгоритм. Наприклад, може бути кілька варіантів переміщень, повороти, перешкоди можуть рухатись, можуть бути вузькі місця, в яких ворогам бажано не товпитися. Кожна така умова робить рух цікавішим, але й уповільнює роботу алгоритму. Через це доводиться щось спрощувати або викручуватись по-іншому для оптимізації роботи гри.

Список літератури:

[1] Grid pathfinding optimizations // URL: https://www.redblobgames.com/ pathfinding/grids/algorithms.html (date of access: 11.01.2022).

[2] Grids and Graphs // URL: https://www.redblobgames.com/ pathfinding/ grids/graphs.html (date of access: 11.01.2022).

[3] Введення в алгоритм A* // URL: https://habr.com/ru/post/331192/ (date of access: 11.01.2022).

[4] Пошук шляху, або як вороги в іграх знаходять дорогу // URL: https://dtf.ru/gamedev/709133-poisk-puti-ili-kak-vragi-v-igrah-nahodyat-dorogu (date of access: 11.01.2022).

[5] Map representations // URL: http://theory.stanford.edu/~amitp/Game Programming/MapRepresentations.html (date of access: 11.01.2022).

[6] Algorithm A * Implementation // URL: https://habr.com/ru/post/331220/ (date of access: 11.01.2022).

[7] Admissible heuristic // URL: https://en.wikipedia.org/wiki/Admissible _heuristic (date of access: 11.01.2022).

[8] Створення штучного інтелекту для ігор – від проектування до оптимізації // URL: https://habr.com/ru/company/intel/blog/265679/ (date of access: 11.01.2022).

[9] Як створити ігровий ШІ: гайд для початківців // URL: https://habr.com/ru/company/pixonic/blog/428892/ (date of access: 11.01.2022).

[10] Як створити ігровий ШІ: гайд для початківців // URL: https://habr.com/ru/post/420219/ (date of access: 11.01.2022).

[11] Не зовсім людина: штучний інтелект в іграх // URL: https://skillbox.ru/media/gamedev/iskusstvennyy-intellekt-v-igrakh/ (date of access: 11.01.2022).

[12] Pathfinding Demystified (Part I): Generic Search Algorithm // URL: https://www.gabrielgambetta.com/generic-search.html (date of access: 11.01.2022).

[13] Generalized Platformer AI Pathfinding // URL: https://www.gamedev.net/articles/programming/artificial-intelligence/generalized-platformer-ai-pathfinding-r3924/ (date of access: 11.01.2022).

[14] Алгоритм Дейкстри. Пошук оптимальних маршрутів на графі // URL: https://habr.com/ru/post/111361/ (date of access: 11.01.2022).

Надійшла до редколегії 14.09.2021