Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

**Лабораторна робота № 1**

з дисципліни «Програмування інтелектуальних інформаційних систем »

Тема: «Пошук шляху»

Виконав: Перевірив:

студент групи ІТ-04 вик. кафедри ІПІ

Коновальчук Андрій Баришич Лука Маріянович

Дата здачі \_\_\_\_\_\_\_\_\_\_\_\_

Захищено з балом \_\_\_\_\_\_\_\_\_\_\_

Київ 2022

# Лабораторна №1

*Тема:*

Пошук шляху

*Мета:*

Розробити програмне рішення основних завдань з пошуку для гри Pacman

*Завдання:*

1. Алгоритм Лі (Q2, замість BFS)

2. А\* пошук:

2.1 Мангеттенський шлях (Q4)

2.2 Звести А\* до жадібного алгоритму (Q3, Q8)

2.3 Зібрати всі монети (Q7)

2.4 Шлях по кутам (Q6)

# Опис програмного коду

Програмний код на GitHub: <https://github.com/KonovalchukA-IT04/PiisLabs/tree/master/lab1>

Програмний код реалізований на базі проекту Berkeley C188 Pacman, де наперед підготована гра, візуалізація, а також архітектура, яка дозволяє розширити і доповнити код. Тож завдання лабораторної роботи зводилось до написання алгоритмів пошуку та доповненні методів в класах проблем. Весь код в рамках даної лабораторної написаний у файлах *search.py* та *searchAgents.py*.

1. *Алгоритм Лі*

Алгорит Лі також називають “хвильовим алгоритмом”, і він заснований на BFS алгоритмі (пошук в ширину). Курс Berkeley передбачав написання власного BFS алгоритму, для проходження лабіринту.



Рисунок 1.1 – Алгоритм Лі. Великий лабіринт

На рисунку бачимо візуалізацію алгоритму. Як видно з рисунка алгоритм обійшов майже весь лабіринт, розкривши 620 вузлів і знайшов єдину відповідь. Через невелику кількість вузлів пошук тривав незначну кількість часу.

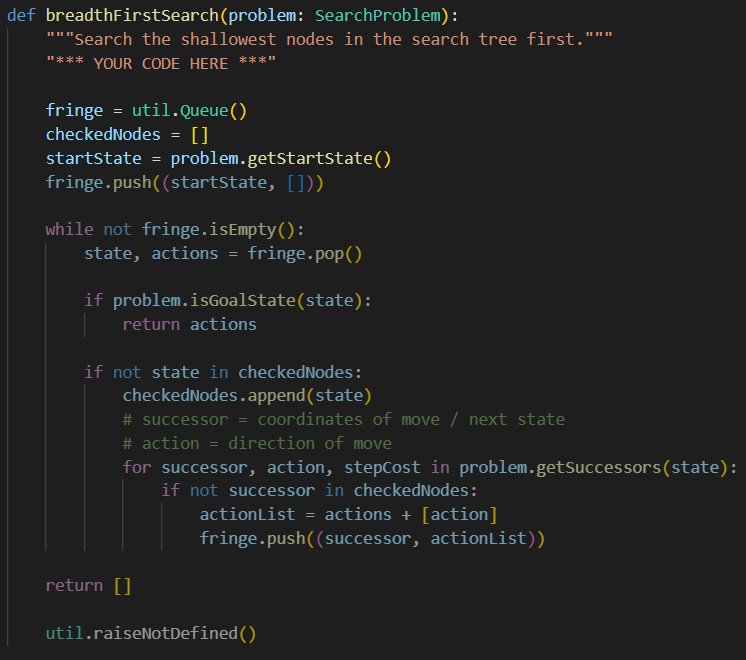


Рисунок 1.2 – Код алгоритму

Алгоритм реалізований з використанням структури даних “Queue” (черга). Всі необхідні методи прописані для нас: “getStartState()” – повертає початковий стан для проблеми пошуку; “isGoalState(state)” – повертає True тоді і тільки тоді, якщо стан є дійсним цільовим станом; “getSuccessors(state)” – для заданого стану повертає successor, action, stepCost, де “successor” є саксесором в поточному стані, “action” – дія, необхідна для досягнення цього стану, а “stepCost” – додаткові витрати на розширення до цього саксесора. Структура коду схожа до всіх алгоритмів, і буде відрізнятися декількома рядками і структурами даних.

Отже, ми створюємо чергу, куди проштовхуємо початковий стан і список дій. Також створюємо список відвіданих вузлів. І допоки черга не буде пуста, повторяємо наступне: дістаємо з черги стан і дії; перевіряємо на цільовий стан; якщо стан не в списку станів, то отримуємо наступний стан, дії, і ціну кроку (яку не використовуємо); якщо наступний стан не в списку відвіданих вузлів, то доповнюємо список дій і проштовхуємо в чергу нові стан і список дій.

1. *Алгоритм А\**

2.1. Мангеттенський шлях

Алгоритм пошуку А\* з мангеттенською евристикою в рази ефективніший за алгоритм Лі, котрий так чи інакше знаходить найкоротшу відстань, проте витрачає на це більше часу та пам’яті.



Рисунок 2.1.1 – Алгоритм А\* (мангеттенський шлях). Великий лабіринт

Як видно з рисунка, порівняно з Лі, алгоритм менше заходив у глухі кути, і розкрив 549 вузлів. Час, знову ж таки, оцінити не можливо, але у складніших проблемах алгоритм в рази швидший.



Рисунок 2.1.2 – Код алгоритму

Якщо описувати алгоритм на основі опису попереднього, то у нас структура даних тепер “PriorityQueue” (черга з пріорітетом). Евристика попередньо прописана:

*“xy1 = position*

*xy2 = problem.goal*

*return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])”*

В чергу тепер додається і розглядається ціна кроку і евристика. На кожній ітерації ми отримуємо евристику в даному стані, а також зважуємо вагу шляху. Після початкового додавання в чергу, ми проштовхуємо наступний стан, список дій, ціну кроку, і також суму ваги шляху і поточну евристику.

2.2. Звести А\* до жадібного алгоритму

Це стає можливим, якщо прибрати евристику, і лише враховувати вагу шляху. Це незначно спрощує наш код, проте при пошуку шляху в лабіринті, за наявною у нас картою (“великий лабіринт”), алгоритм робить ідентичне рішення, як за BFS. Проте це пов’язано лише з тим, що до цілі один єдиний вірний шлях.

Але якщо забігти трішки наперед і розглянути іншу проблему – проблему обходу всіх точок, то за діями Пакмена можна побачити притаманні жадібному алгоритму дії – він поїдає найближчі точки, і не йде по найкоротшому шляху.



Рисунок 2.2.1 – UCS. Поїдання найближчих точок

Даний алгоритм називається Uniform Search Cost (“пошук за критерієм вартості”), і дійсно відрізняється від попереднього лише відсутністю евристики.

Для вирішення проблеми поїдання всіх (найближчих) точок ми доповнили декілька методів: доповнили “isGoalState”, де повертаємо булове значення на присутність їжі на координатах

*“if self.food[x][y]:*

*return True*

*return False”;*

і викликаємо у методі проблеми “findPathToClosestDot” алгоритм UCS.

2.3 Зібрати всі монети

Для даної проблеми ми маємо лише прописати евристику для відповідного агента. Евристика є відстанню до найдальшої їжі на карті.

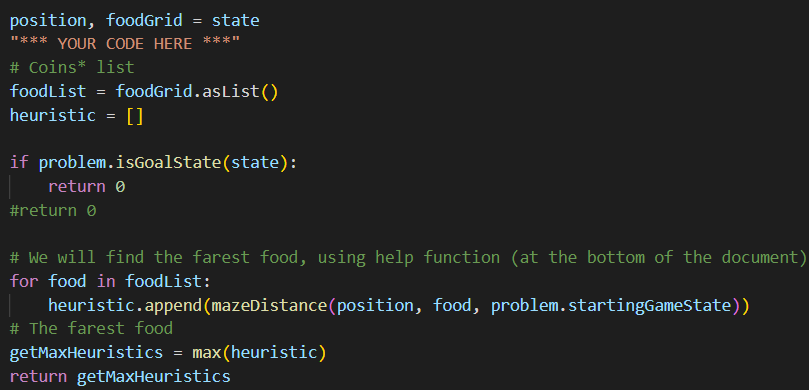


Рисунок 2.3.1 – Евристика

Тестова карта містить всього 13 точок, бо більша кількість займе в більшу часу на пошук. Коли Пакмен почне рухатись, в консоль виведеться довжина шляху (60 вузлів), час пошуку (8.1 секунд), кількість відкритих вузлів (4137):



Рисунок 2.3.2 – Карта та процес пошуку

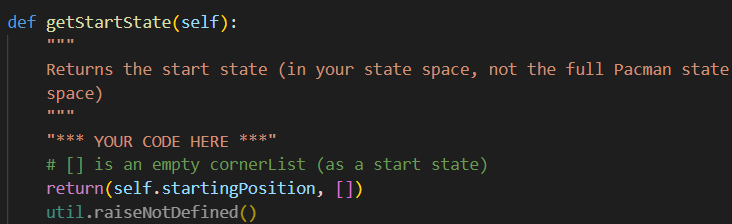


Рисунок 2.3.3 – Рух по найоптимальнішому шляху та результати оцінки

За даними із сайту проекту, наш алгоритм максимально ефективний, адже відкрив менше 7000 вузлів, і процес зайняв менше 13 секунд.

2.4 Шлях по кутам

Для вирішення цієї проблеми ми маємо переписати всі попередні методи і прописати евристику.



Рисунко 2.4.1 – Метод “getStartState”

В даному методі просто повертаємо початкову позицію Пакмена.

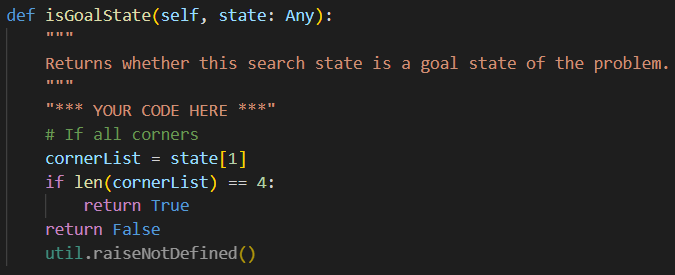


Рисунок 2.4.2 – Метод “isGoalState”

В даному методі перевіряємо величину списку відвіданих кутків (з’їдених точок), і якщо величина дорівнює 4, то це цільовий результат.

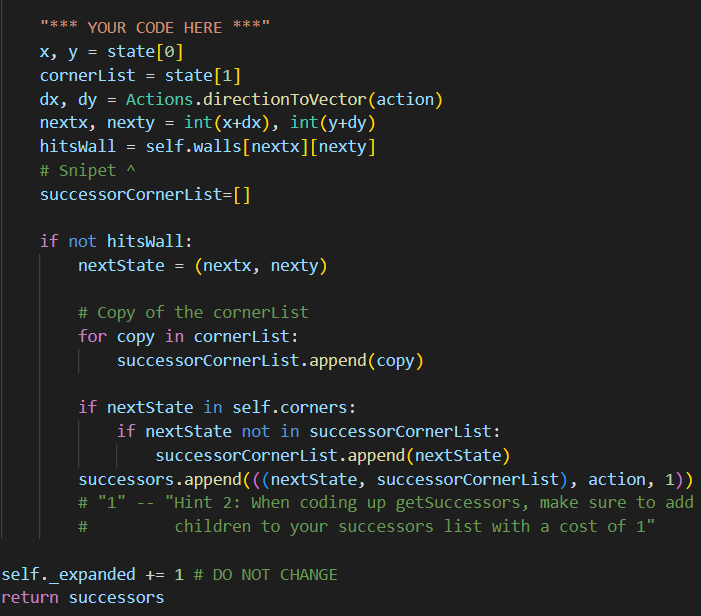


Рисунок 2.4.3 – Метод “getSuccessors”

Частина коду є шаблоном, в якому описується доступ до стін та списку кутків. Як описувалось раніше даний метод повертає саксесори – наступні стани. Важливо також не забути за вказівку із завдання курсу – “додати нащадка у список з ціною кроку 1”. Крім наступного стану в список додається також список кутків, які ми перевіряємо попереднім методом.

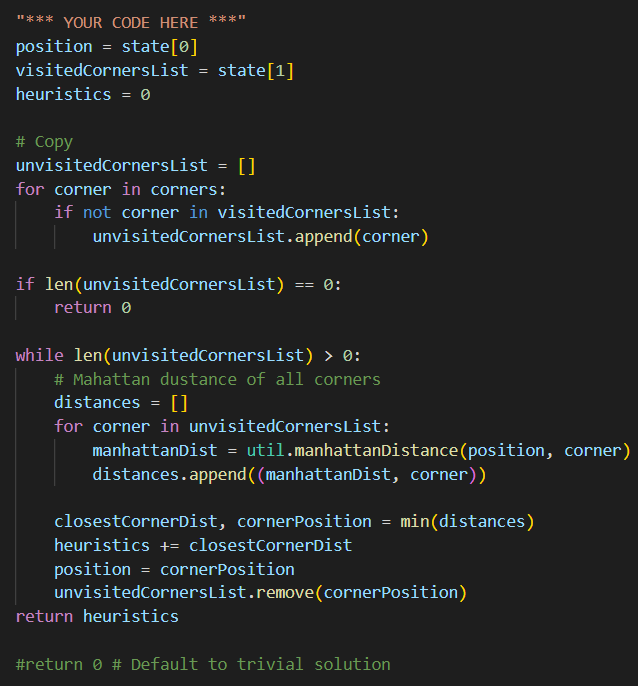


Рисунок 2.4.4 – Евристика

В евристиці перевіряються невідвідані кутки, та за мангеттенською відстанню шукається найближчий шлях до них.

Приклад вирішення проблеми пошуку шляху через всі кути на тестовій карті (2 кути уже відвідано):



Рисунок 2.4.5 – Проблема 4-ох кутів. Карта “середні кути”

# Висновок

В ході даної лабораторної роботи були вирішені проблеми пошуку в лабіринті, проходження через всі точки та проходження через всі кути.

Було реалізовано наступні алгоритми: алгоритм Лі, A\* (мангеттенський шлях), Uniform Cost Search (A\* зведений до жадібного алгоритму).

Лабораторна робота виконана на основі проекту Berkeley C188 Pacman. Розібрано структуру та архітектуру проекту, і доповнені наступні файли: search.py, searchAgents.py.

В звіті наявні описи алгоритмів, а також скріншоти з кодом та роботою алгоритмів.