КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра математичної інформатики

Звіт

до лабораторної роботи №2

з дисципліни "Інтелектуальні системи"

Гра PAC-MAN

Виконала студентка 4 курсу групи МІ

Баклан Аліса

Київ - 2023

Постановка задачі

Гра “Пекмен”. Побудувати лабіринт, реалізувати пошук шляху двома алгоритмами - A* та жадібним, написати логіку привидів.

Використані технології

Роботу було виконано мовою C++ у середовищі Visual Studio Code. Для графіки було встановлено мультимедійну бібліотеку SFML.

1. *Завантаження ресурсів*

   Необхідно завантажити бібліотеку SFML. Для цього на офіційному сайті[1] завантажуємо MinGW (SEH) останньої версії та розтискаємо архів у зручному місці.

   У середовищі створюємо теку для майбутнього проєкту.

2. *Конфігурація*

   2.1 Використання CMake.

   За потреби завантажуємо WinLibs MSVCRT на тому ж сайті та CMake[2]. Рекомендується додати два шляхи до системних змінних:

   - шлях до mingw64
   - шлях до mingw64/bin

   У Visual Studio Code встановлюємо надбудови CMake та CMake Tools. За допомогою цих інструментів створюємо CMakeLists.txt, за потреби редагуємо файл cmake-tools-kit.json для створення особистого kit.

   2.2 Вручну без CMake

   Можна зібрати проєкт користуючись лише файлами SFML, копіюючи їх в проєкт:

```
Project/
├──src
      ├──lib
      ├── include
├── main.cpp
├── bin. . .
```

   Тут папки bin, lib, include скопійовані в проєкт.

   Для зручності створюємо Makefile - (у терміналі make):

```
CXXFLAGS = -Isrc/include
LDFLAGS = -Lsrc/lib -lsfml-graphics -lsfml-window
-lsfml-system
all: compile link
compile:
    g++ $(CXXFLAGS) -c main.cpp
link:
    g++ main.o -o main $(LDFLAGS)
```

Опис алгоритмів

Жадібний алгоритм.

f = h, де:

- h - це оцінка відстані від поточної точки до кінцевої точки

Жадібний алгоритм вибирає найкращий доступний варіант на кожному кроці, без перевірки, чи веде цей вибір до найкращого загального рішення.

На кожному кроці алгоритму ми вибираємо найкращий можливий варіант - клітинку лабіринту найближчу до цілі (найменше f=h), яка є найвигіднішою на даному етапі.

Алгоритм завершується, коли досягнуто цілі в лабіринті. Жадібний алгоритм може завершитися з локально оптимальним рішенням, яке не обов'язково є глобально оптимальним.

*Алгоритм A**.

f = g + h, де:

- g - це вага шляху від початкової точки до поточної точки,
- h - це оцінка відстані від поточної точки до кінцевої точки

На відміну від жадібного, цей алгоритм A* гарантує знаходження найкоротшого шляху, враховуючи як вартість проходження через різні точки, так і прогнозовану відстань до кінцевої точки.

Виконання роботи, огляд коду

1. *Створення лабіринтів*

   Лабіринти реалізовано як двовимірний масив 0 і 1, де 1 - стінка, 0 - вільний шлях. Лабіринт зображується за цим масивом, тому не треба

додаткових файлів з картинками.

```
std::vector<std::vector<int>> mash_m = {
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1},
    {1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1},
    {1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
    {1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
};
std::vector<int> mash_p = {1,1,10, 28, 10, 27};
std::pair<std::vector<std::vector<int>>, std::vector<int>> mash = {mash_m, mash_p};
```

2. *Керування Pac-Man*

Керування Пекменом відбувається вручну стрілочками, Пекмен зображається кружечком.

```
//Pac-Man aka Yellow circle
sf::CircleShape pacman(CELL_SIZE / 2);
pacman.setFillColor(sf::Color::Yellow);
pacman.setOrigin(CELL_SIZE / 2, CELL_SIZE / 2);
```

```
// Pac-man Controls
sf::Event event;
while (window.pollEvent(event)) {
    if (event.type == sf::Event::Closed) {
        window.close();
    }
    if (event.type == sf::Event::KeyPressed) {
        if (event.key.code == sf::Keyboard::Up && maze[pacmanY - 1][pacmanX] == 0) {
            pacmanY--;
        } else if (event.key.code == sf::Keyboard::Down && maze[pacmanY + 1][pacmanX] == 0) {
            pacmanY++;
        } else if (event.key.code == sf::Keyboard::Left && maze[pacmanY][pacmanX - 1] == 0) {
            pacmanX--;
        } else if (event.key.code == sf::Keyboard::Right && maze[pacmanY][pacmanX + 1] == 0) {
            pacmanX++;
        }
    }
}
```

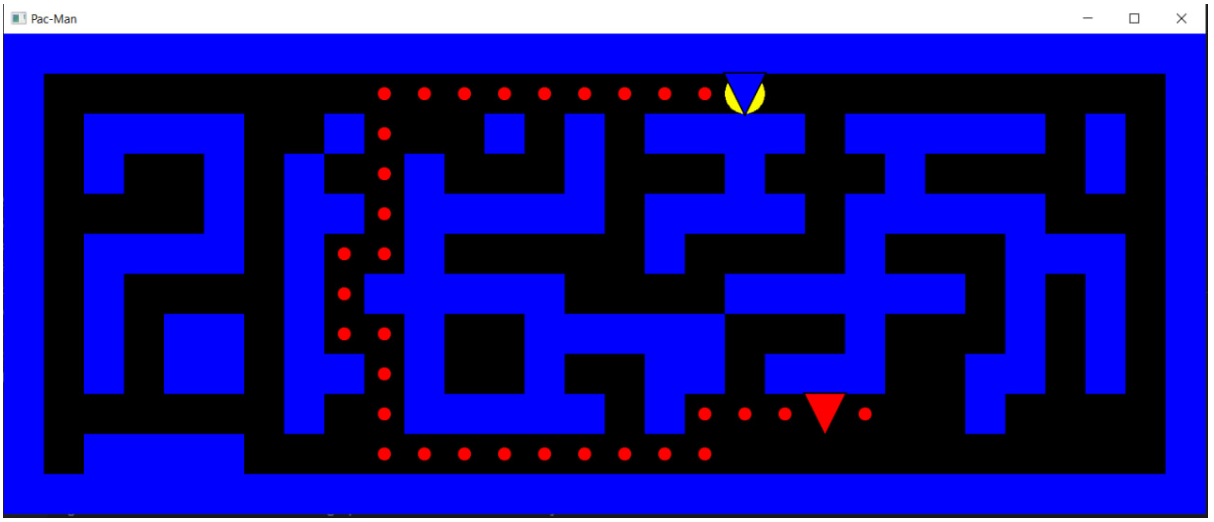
3. *Логіка привидів*

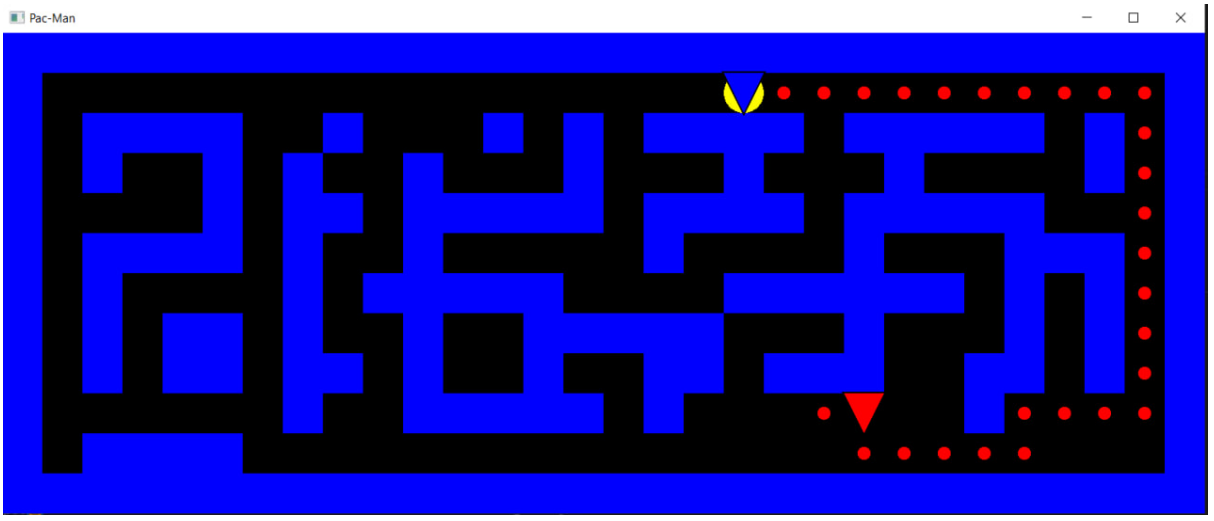
Привиди намагаються наздогнати Пекмена. У них є певна швидкість, що задає кількість ітерацій за певний проміжок часу. Два привиди обчислюють шлях жадібним та A* алгоритмом, та рухаються за обчисленою ними траєкторією, поки Пекмен не

змінить координати.

```
while (window.isOpen()) {

    sf::Time elapsed = clock.restart();
    deltaTime += elapsed.asSeconds();

    while (deltaTime >= (1.0f / GHOST_SPEED)) {

        std::pair<int, int> ghost1Directions;
        std::pair<int, int> ghost2Directions;

        if (pacmanX != prevPacmanX || pacmanY != prevPacmanY){//if Pac-Man moved
            prevPacmanX = pacmanX;
            prevPacmanY = pacmanY;
            step = 1;
            //Greedy Search for RED
            pathGhost1 = findPathGreedy(maze, {ghost1.getY(), ghost1.getX()}, {pacmanY, pacmanX});
            //A* for Green
            pathGhost2 = findPathAStar(maze, {ghost2.getY(), ghost2.getX()}, {pacmanY, pacmanX});
        }
        else step++;//if pac-man stayed in place continue on prev path
```

Обчислення нової траєкторії на кожному кроці може призвести до зациклення:





При обчисленні нового шляху на кожному кроці червоний привид “зациклюється”, не доходячи до Пекмена. Тому нова траєкторія обчислюється тільки тоді, коли Пекмен поворушиться.

Траєкторії обох привидів позначаються крапками.

```
// Draw the path of RED
for (const auto& point : pathGhost1) {
    sf::CircleShape pathDot(CELL_SIZE / 6);
    pathDot.setFillColor(sf::Color::Red);
    pathDot.setOrigin(pathDot.getRadius(), pathDot.getRadius());
    pathDot.setPosition(point.second * CELL_SIZE + CELL_SIZE / 2 - 4, point.first * CELL_SIZE + CELL_SIZE / 2 - 4);
    window.draw(pathDot);
}

// Draw the path of Green
for (const auto& point : pathGhost2) {
    sf::CircleShape pathDot(CELL_SIZE / 6);
    pathDot.setFillColor(sf::Color::Green);
    pathDot.setOrigin(pathDot.getRadius(), pathDot.getRadius());
    pathDot.setPosition(point.second * CELL_SIZE + CELL_SIZE / 2 + 4, point.first * CELL_SIZE + CELL_SIZE / 2 + 4);
    window.draw(pathDot);
}
```

4. *Алгоритми знаходження шляху*

Алгоритми реалізовані у файлі search.h.

```
struct Node {
    int x, y;
    int cost; //(heuristic)
};

bool operator<(const Node& a, const Node& b) {
    return a.cost > b.cost;
}

//Greedy algorithm
//each move greedy, returs first found path
std::vector<std::pair<int, int>> findPathGreedy(const
std::vector<std::vector<int>>& maze, std::pair<int, int> start,
std::pair<int, int> destination) {
    int numRows = maze.size();
    int numCols = maze[0].size();
    std::vector<std::pair<int, int>> directions = {{-1, 0}, {1,
0}, {0, -1}, {0, 1}};

    std::priority_queue<Node> frontier;
    frontier.push({start.first, start.second, 0});

    std::vector<std::vector<bool>> visited(numRows,
std::vector<bool>(numCols, false));
```

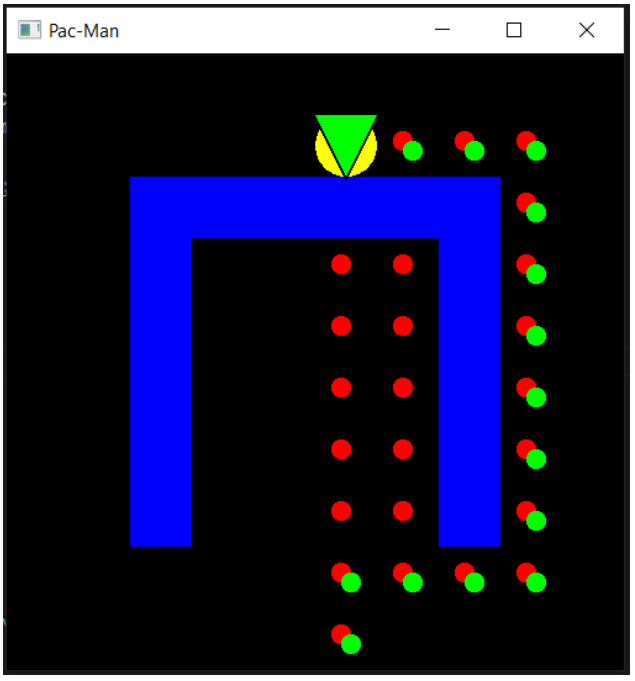
```
    std::vector<std::vector<std::pair<int, int>> >
cameFrom(numRows, std::vector<std::pair<int, int>>(numCols, {-1,
-1}));

    while (!frontier.empty()) {
        Node current = frontier.top();
        frontier.pop();

        if (current.x == destination.first && current.y ==
destination.second) {//if reached finish returning path
            std::vector<std::pair<int, int>> path;
            std::pair<int, int> currentPoint =
{destination.first, destination.second};
            while (currentPoint != start) {
                path.push_back(currentPoint);
                currentPoint =
cameFrom[currentPoint.first][currentPoint.second];
            }
            path.push_back(start);
            std::reverse(path.begin(), path.end());
            return path;
        }

        visited[current.x][current.y] = true;

        for (const auto& direction : directions) {
            int newX = current.x + direction.first;
            int newY = current.y + direction.second;

            if (newX >= 0 && newX < numRows && newY >= 0 && newY
< numCols && maze[newX][newY] == 0 && !visited[newX][newY]) {
                int heuristic = std::abs(newX -
destination.first) + std::abs(newY - destination.second);
                frontier.push({newX, newY, heuristic});
                cameFrom[newX][newY] = {current.x, current.y};
            }
        }
    }

    // No path found
    return {};
}
```

Алгоритм A* схожий за реалізацією, але має відмінну евристику:

```
// A* algorithm
std::vector<std::pair<int, int>> findPathAStar(const
std::vector<std::vector<int>>& maze, std::pair<int, int> start,
std::pair<int, int> destination) {
    //. . .
if (newCost < costSoFar[newX][newY]) {
                    costSoFar[newX][newY] = newCost;
  int heuristic = newCost + std::abs(newX - destination.first) +
std::abs(newY - destination.second);  // A* heuristic
    //. . .
                    }
```

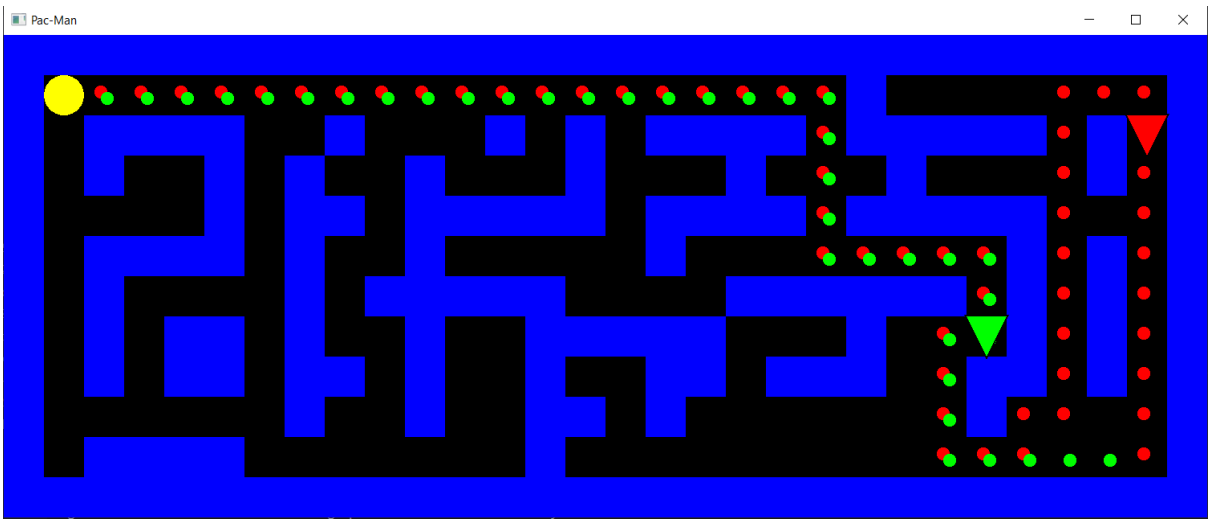
Маємо такий результат:

- Червоний - жадібний алгоритм
- Зелений - A*



Бачимо що жадібний алгоритм (червоний привид), на відміну від A* (зелений привид), не завжди знаходить найкоротший шлях.

Обидва алгоритми є достатньо швидкими, оскільки часто не аналізують весь лабіринт, як стандартні пошуки в ширину/глибину.



Посилання

1.https://www.sfml-dev.org/download.php
2.https://cmake.org/download/
3. Код: https://github.com/alya1b/Pac_Man