ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA VẬT LÝ

HOÀNG HỮU HIẾU - 20002125
NGUYỄN TRIỆU VƯƠNG - 20002182
NGUYỄN ANH TUẤN - 20002176

MÔ PHỎNG TRÒ CHƠI RẮN TỰ ĐỘNG TÌM ĐƯỜNG ĂN MỒI SỬ DỤNG PRIORITY QUEUE TRONG THUẬT TOÁN A*

Báo cáo kết thúc học phần Cấu trúc dữ liệu và giải thuật
Ngành Kỹ thuật Điện tử và Tin học

Hà Nội - 2023

Mục lục

MỞ ĐẦU

Trò chơi rắn là một trò chơi cổ điển đã được yêu thích trong nhiều thập kỷ. Mục tiêu của trò chơi là điều khiển một con rắn ăn các mồi xuất hiện trên màn hình để trở nên dài hơn và tránh va chạm vào biên trò chơi hoặc chính bản thân của mình. Trong trò chơi này, chúng ta sẽ tập trung vào việc tạo ra một trò chơi rắn tự động tìm đường ăn mồi sử dụng thuật toán A*.

Thuật toán A* là một trong những thuật toán tìm đường ngắn nhất phổ biến nhất và priority queue là một cấu trúc dữ liệu quan trọng để triển khai thuật toán này. Bằng cách kết hợp hai phương pháp này, chúng ta sẽ tạo ra một trò chơi rắn thông minh và có khả năng tự động tìm đường đến mồi. Trong đề tài này, chúng ta sẽ tìm hiểu chi tiết về thuật toán A*, cũng như cách triển khai chúng trong trò chơi rắn.

Nội dung bài báo cáo này được chia làm 2 chương:

- Chương 1: Tìm hiểu thuật toán A* trong trò chơi rắn tự động ăn mồi
- Chương 2: Mô phỏng trò chơi rắn tự động ăn mồi

Chương 1: Tìm hiểu các thuật toán trong trò chơi rắn tự động ăn mồi

1.1 Thuật toán tìm kiếm A*

A* là thuật toán tìm kiếm trong đồ thị, thuật toán sẽ tìm một đường tối ưu nhất từ vị trí ban đầu đến vị trí đích. Thuật toán sử dụng "đánh giá heuristic" để sắp xếp từng loại từng điểm đến để đến vị trí tốt nhất. Thuật toán sẽ duyệt qua các vị trí đích đích theo đánh giá của heuristic này.

Công thức của thuật toán A* được biểu diễn như sau:

1. Khởi tạo một tập đóng (closed set) và một tập mở (open set), bao gồm các đỉnh đã duyệt và các đỉnh chưa duyệt.
2. Khởi tạo một đỉnh đầu tiên và đưa nó vào tập mở.
3. Trong mỗi vòng lặp, chọn đỉnh trong tập mở có giá trị f(g + h) nhỏ nhất và đưa nó vào tập đóng. Ở đây, g là chi phí từ đỉnh bắt đầu đến đỉnh hiện tại, h là hàm heuristic ước tính khoảng cách từ đỉnh hiện tại đến đỉnh đích.

   f(n) = g(n) + h(n)
4. Nếu đỉnh đó là đỉnh đích, trả về đường đi từ đỉnh bắt đầu đến đỉnh đích.
5. Nếu không, duyệt các đỉnh kề của đỉnh hiện tại và cập nhật giá trị g và f cho mỗi đỉnh kề.

   g(k) = g(n) + c(n,k)

   f(k) = g(k) + h(k)

   Trong đó, c(n,k) là chi phí từ đỉnh n đến đỉnh k.
6. Nếu đỉnh kề chưa được duyệt, đưa nó vào tập mở. Nếu đã được duyệt và có giá trị f tốt hơn, cập nhật giá trị f và đưa nó vào tập mở.
7. Lặp lại các bước 3-6 cho đến khi tìm được đỉnh đích hoặc tập mở trống.
8. Nếu không tìm được đường đi từ đỉnh bắt đầu đến đỉnh đích, trả về null.

Thuật toán A* là một thuật toán tìm kiếm đường đi trong đồ thị hoặc lưới dựa trên hàm heuristic để ước lượng chi phí di chuyển từ một điểm xuất phát đến một điểm đích. Độ phức tạp của thuật toán A* phụ thuộc vào các yếu tố như kích thước của đồ thị, chất lượng của hàm heuristic và cách mà nó được triển khai.

1. Độ phức tạp thời gian:

- Trường hợp tốt nhất: Nếu hàm heuristic cho kết quả chính xác ngay từ đầu, thuật toán A* chỉ cần duyệt qua một số lượng rất nhỏ các nút trên đồ thị, và do đó, độ phức tạp thời gian là gần như O(1).

- Trường hợp trung bình: Độ phức tạp thời gian trung bình của thuật toán A* được xác định bởi số lượng nút và cạnh trong đồ thị. Trong trường hợp trung bình, độ phức tạp thời gian của thuật toán A* là O(b^d), trong đó b là số lượng các nút trên mỗi nút và d là độ sâu của đường đi tìm được.
- Trường hợp xấu nhất: Trong trường hợp xấu nhất, thuật toán A* có thể phải duyệt qua tất cả các nút trên đồ thị để tìm được đường đi tối ưu. Độ phức tạp thời gian trong trường hợp xấu nhất là O(b^d), trong đó b là số lượng các nút trên mỗi nút và d là độ sâu của đường đi tối ưu.

2. Độ phức tạp không gian:
- Độ phức tạp không gian của thuật toán A* phụ thuộc vào cách triển khai và cấu trúc dữ liệu được sử dụng để lưu trữ các nút trong quá trình tìm kiếm. Thuật toán A* có thể sử dụng một hàng đợi ưu tiên (priority queue) để lưu trữ các nút theo thứ tự ưu tiên dựa trên giá trị heuristic và chi phí đi từ điểm xuất phát. Độ phức tạp không gian trong trường hợp này là O(b^d), trong đó b là số lượng các nút trên mỗi nút và d là độ sâu của đường đi tìm được.

Tóm lại, độ phức tạp của thuật toán A* trong trường hợp tốt nhất, trung bình và xấu nhất theo thời gian và không gian là:

- Thời gian:
  + Tốt nhất: O(1)
  + Trung bình: O(b^d)
  + Xấu nhất: O(b^d)

- Không gian:
  + Tốt nhất, trung bình, xấu nhất: O(b^d)

1.2 Thuật toán Priority Queue

Priority Queue là một cấu trúc dữ liệu trong đó các phần tử được lưu trữ dưới dạng danh sách tuần tự, nhưng được sắp xếp theo thứ tự ưu tiên. Trong đó, phần tử ưu tiên cao hơn được xử lý trước.

Priority Queue có 2 thao tác quan trọng là chèn phần tử và lấy phần tử có ưu tiên cao nhất. Khi chèn một phần tử vào Priority Queue, phần tử này sẽ được sắp xếp đúng vị trí của nó trong danh sách theo tiêu chí ưu tiên được định nghĩa trước đó. Khi lấy phần tử có ưu tiên cao nhất, phần tử đầu tiên trong danh sách (tức là phần tử ở đầu hàng đợi) sẽ được trả về và xóa khỏi danh sách.

Việc sử dụng Priority Queue rất hữu ích trong nhiều ứng dụng, bao gồm các thuật toán tìm kiếm đường đi ngắn nhất, sắp xếp dữ liệu, xử lý các sự kiện ưu tiên trong hệ thống,...

Dưới đây là mã Java thuật toán Priority Queue:

```
import java.util.ArrayList;
import java.util.List;

public class PriorityQueue<E extends Comparable<E>> {
    private List<E> queue;  // List to store the elements of the priority
queue

    public PriorityQueue() {
        queue = new ArrayList<>();  // Initialize the queue as an empty
ArrayList
    }

    public void add(E element) {
        queue.add(element);  // Add the element to the end of the queue
        int i = queue.size() - 1;  // Index of the newly added element
        // Perform "up-heap" operation to maintain the min-heap property
        while (i > 0 && queue.get(i).compareTo(queue.get(parent(i))) < 0) {
            swap(i, parent(i));  // Swap the element with its parent if it is
smaller
            i = parent(i);  // Move up to the parent index
        }
    }

    public E poll() {
        if (queue.isEmpty()) {
            return null;  // If the queue is empty, return null
        }
        E element = queue.get(0);  // Get the root element (minimum element)
        queue.set(0, queue.get(queue.size() - 1));  // Replace the root with
the last element
        queue.remove(queue.size() - 1);  // Remove the last element from the
queue
        minHeapify(0);  // Perform "down-heap" operation to maintain the
min-heap property
        return element;  // Return the removed element (minimum element)
    }

    public boolean contains(E element) {
        return queue.contains(element);  // Check if the queue contains the
specified element
    }

    public boolean isEmpty() {
```

```
        return queue.isEmpty();  // Check if the queue is empty
    }

    private void minHeapify(int i) {
        int left = left(i);  // Index of the left child
        int right = right(i);  // Index of the right child
        int smallest = i;  // Assume the current node is the smallest
        // Compare the left child with the current node and update the
smallest index if necessary
        if (left < queue.size() &&
queue.get(left).compareTo(queue.get(smallest)) < 0) {
            smallest = left;
        }
        // Compare the right child with the current node and update the
smallest index if necessary
        if (right < queue.size() &&
queue.get(right).compareTo(queue.get(smallest)) < 0) {
            smallest = right;
        }
        if (smallest != i) {
            swap(i, smallest);  // Swap the current node with the smallest
child
            minHeapify(smallest);  // Recursively perform "down-heap"
operation on the affected child
        }
    }

    private void swap(int i, int j) {
        E temp = queue.get(i);  // Temporary variable to hold the element at
index i
        queue.set(i, queue.get(j));  // Replace element at index i with
element at index j
        queue.set(j, temp);  // Replace element at index j with the temporary
variable
    }

    private int parent(int i) {
        return (i - 1) / 2;  // Calculate the index of the parent node
    }

    private int left(int i) {
        return 2 * i + 1;  // Calculate the index of the left child node
    }

    private int right(int i) {
        return 2 * i + 2;  // Calculate the index of the right child node
    }
}
```

1.3 Thuật toán Priority Queue trong thuật toán A* vào trò chơi rắn tự động ăn mồi

Trong trò chơi rắn tự động ăn mồi, Priority Queue và thuật toán A* được sử dụng để tìm đường đi tối ưu từ vị trí hiện tại của rắn đến mồi, tránh các chướng ngại vật và các phần thân của rắn.

Cụ thể, ứng dụng Priority Queue trong trò chơi rắn tự động ăn mồi như sau:

1. Tạo một Priority Queue với độ ưu tiên dựa trên khoảng cách từ các ô đến mồi.
2. Thêm ô hiện tại của rắn vào hàng đợi ưu tiên với độ ưu tiên là khoảng cách từ ô hiện tại đến mồi.
3. Lấy ô đầu tiên từ hàng đợi ưu tiên và kiểm tra xem có phải là ô mồi không. Nếu đúng, rắn sẽ di chuyển đến ô đó và ăn mồi. Nếu không, rắn sẽ đi đến ô đó và thêm các ô xung quanh vào hàng đợi ưu tiên với độ ưu tiên được tính toán dựa trên khoảng cách từ các ô đó đến mồi. Các ô đã xét sẽ được đánh dấu để tránh xét lại ở lần sau.
4. Lặp lại bước 3 cho đến khi rắn đến được ô mồi hoặc không còn ô nào trong hàng đợi ưu tiên.

Thuật toán A* được sử dụng để tính toán độ ưu tiên cho các ô được thêm vào hàng đợi ưu tiên. Thuật toán A* sử dụng hàm ước lượng (heuristic function) để dự đoán khoảng cách từ một ô bất kỳ đến mồi. Hàm ước lượng được tính toán bằng cách tính khoảng cách Manhattan (hoặc Euclidean) từ ô đó đến ô mồi.

Tóm lại, trong trò chơi rắn tự động ăn mồi, Priority Queue và thuật toán A* được sử dụng để tìm đường đi tối ưu từ vị trí hiện tại của rắn đến mồi, tránh các chướng ngại vật và các phần thân của rắn.

1.4 Thiết kế và triển khai thuật toán A* cho trò chơi rắn săn mồi

Dưới đây là một thiết kế và triển khai thuật toán A* cho trò chơi rắn săn mồi, trong đó sử dụng Priority Queue để xác định gCost (chi phí thực tế từ điểm bắt đầu đến điểm hiện tại).

Đầu tiên, ta cần một cấu trúc dữ liệu để đại diện cho một điểm trong trò chơi. Trong trường hợp này, ta gọi nó là Node. Mỗi Node sẽ có các thuộc tính như vị trí (x, y), gCost, hCost, fCost và parent (để theo dõi đường dẫn tốt nhất).

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

```
// Class representing a node in the grid
class Node {
    private int x; // x-coordinate
    private int y; // y-coordinate
    private int gCost; // cost from start node to this node
    private int hCost; // heuristic cost from this node to goal node
    private int fCost; // total cost (gCost + hCost)
    private Node parent; // parent node in the path

    // Constructor
    public Node(int x, int y) {
        this.x = x;
        this.y = y;
        this.gCost = 0;
        this.hCost = 0;
        this.fCost = 0;
        this.parent = null;
    }

    // Getter methods
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getGCost() {
        return gCost;
    }

    // Setter methods
    public void setGCost(int gCost) {
        this.gCost = gCost;
    }

    public int getHCost() {
        return hCost;
    }

    public void setHCost(int hCost) {
        this.hCost = hCost;
    }

    public int getFCost() {
        return fCost;
```

```
    }

    public void setFCost(int fCost) {
        this.fCost = fCost;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    // Override the equals method to compare two nodes
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Node other = (Node) obj;
        return x == other.x && y == other.y;
    }
}
```

Sau đó, chúng ta cần triển khai thuật toán A* sử dụng Priority Queue, trong đó sử dụng đoạn code Priority Queue ở mục 1.2 để xây dựng.

```
public class AStar {
    public List<Node> aStar(Node start, Node goal) {
        // Create a priority queue to store nodes based on their fCost
(lowest fCost comes first)
        PriorityQueue<Node> openSet = new PriorityQueue<>((n1, n2) ->
n1.getFCost() - n2.getFCost());

        // Add the start node to the open set
        openSet.add(start);

        // Create a set to store the visited nodes
        Set<Node> closedSet = new HashSet<>();

        // Continue the search until the open set is empty (no more nodes to
explore)
        while (!openSet.isEmpty()) {
```

```
            // Get the node with the lowest fCost from the open set
            Node current = openSet.poll();

            // Check if the current node is the goal node
            if (current.equals(goal)) {
                // Reached the goal

                // Create a list to store the path from start to goal
                List<Node> path = new ArrayList<>();

                // Reconstruct the path by following the parent pointers from
the goal node to the start node
                while (current != null) {
                    path.add(current);
                    current = current.getParent();
                }

                // Return the reversed path (from start to goal)
                return path;
            }

            // Add the current node to the closed set (visited nodes)
            closedSet.add(current);

            // Traverse neighboring nodes
            for (Node neighbor : getNeighbors(current)) {
                // Skip neighbors that are already in the closed set
                if (closedSet.contains(neighbor)) {
                    continue;
                }

                // Calculate the new gCost for the neighbor
                int newGCost = current.getGCost() +
calculateDistance(current, neighbor);

                // Check if the new gCost is lower than the neighbor's
current gCost or if the neighbor is not in the open set
                if (newGCost < neighbor.getGCost() ||
!openSet.contains(neighbor)) {
                    // Update the neighbor's gCost, hCost, fCost, parent
                    neighbor.setGCost(newGCost);
                    neighbor.setHCost(calculateDistance(neighbor, goal));
                    neighbor.setFCost(neighbor.getGCost() +
neighbor.getHCost());
                    neighbor.setParent(current);

                    // Add the neighbor to the open set if it's not already
present
                    if (!openSet.contains(neighbor)) {
```

```
                        openSet.add(neighbor);
                    }
                }
            }
        }

        // No path found
        return null;
    }

    // Get the neighboring nodes of a given node
    private List<Node> getNeighbors(Node node) {
        // Create a list to store the neighboring nodes
        List<Node> neighbors = new ArrayList<>();

        // Extract the coordinates of the current node
        int x = node.getX();
        int y = node.getY();

        // Create the four neighboring nodes (left, right, top, bottom)
        Node leftNeighbor = new Node(x - 1, y);
        neighbors.add(leftNeighbor);

        Node rightNeighbor = new Node(x + 1, y);
        neighbors.add(rightNeighbor);

        Node topNeighbor = new Node(x, y - 1);
        neighbors.add(topNeighbor);

        Node bottomNeighbor = new Node(x, y + 1);
        neighbors.add(bottomNeighbor);

        return neighbors;
    }

    // Calculate the Manhattan distance between two nodes
    private int calculateDistance(Node node1, Node node2) {
        int distance = Math.abs(node1.getX() - node2.getX()) +
Math.abs(node1.getY() - node2.getY());
        return distance;
    }
}
```

Trong mã trên, chúng ta sử dụng một hàm ‘get_neighbors(node)’ để lấy danh sách các điểm lân cận của một Node và một hàm ‘calculate_distance(node1, node2)’ để tính toán khoảng cách giữa hai Node.

Trong ví dụ trên, hàm ‘get_neighbors(node)’ trả về danh sách các điểm lân cận của một Node. Trong trường hợp này, chúng ta giả sử rằng mỗi Node chỉ có thể di chuyển sang trái, phải, trên hoặc dưới. Do đó, chúng ta tạo ra các Node mới tương ứng với các vị trí lân cận và thêm chúng vào danh sách neighbors.

Hàm ‘calculate_distance(node1, node2)’ tính toán khoảng cách giữa hai Node sử dụng khoảng cách Manhattan. Trong trường hợp này, chúng ta tính tổng các khoảng cách theo chiều ngang và chiều dọc giữa hai Node.

Lưu ý rằng mã trên chỉ là một ví dụ cơ bản để minh họa cách triển khai hàm get_neighbors và calculate_distance trong trò chơi rắn săn mồi. Bạn cần tùy chỉnh và thích ứng mã để phù hợp với cấu trúc và quy tắc của trò chơi cụ thể của bạn.

Trong đoạn mã nguồn lớp *AStarAIPanel.java*, hàm aStar() đã cung cấp sử dụng hàng đợi ưu tiên chưa được sắp xếp (**Unsorted Priority Queue**). Điều này là do lớp *PriorityQueue* trong Java triển khai hàng đợi ưu tiên bằng cách sử dụng cấu trúc dữ liệu heap. Các phần tử của hàng đợi ưu tiên được sắp xếp theo trật tự tự nhiên của chúng hoặc bởi một bộ so sánh được cung cấp tại thời điểm xây dựng hàng đợi. Trong trường hợp này, không có bộ so sánh nào được cung cấp, vì vậy thứ tự tự nhiên của lớp *Node* được sử dụng. Phương thức *poll()* truy xuất và loại bỏ phần đầu của hàng đợi này, đây là phần tử nhỏ nhất theo thứ tự đã chỉ định. Vì các phần tử không được sắp xếp theo bất kỳ thứ tự cụ thể nào trong hàng đợi nên nó là hàng đợi ưu tiên chưa sắp xếp.

Trong trường hợp **Unsorted Priority Queue**, độ phức tạp chung của các thao tác là:

- Chèn (Insert) là O(1)
- Loại bỏ phần tử có ưu tiên cao nhất (Remove-Max): O(log n)

Chương 2: Mô phỏng trò chơi rắn tự động ăn mồi

2.1 Các chú ý về dự án đã làm

Mã nguồn và dữ liệu có thể được lấy từ đây. Trong đó, toàn bộ mã nguồn của dự án được clone về từ đây, giao diện và chế độ đi xuyên biên trò chơi được tham khảo từ dự án đã được học ở môn "Lập trình hướng đối tượng", từ đó loại bỏ đi các class không cần thiết và phát triển thêm chế độ đi xuyên biên mỗi khi bị kẹt.

Lưu ý: trong mã nguồn, tất cả các class đã được gắn comment docstring java giải thích bằng tiếng Anh cho toàn bộ code, class quan trọng nhất là 'AStarAIPanel' đã được giải thích kỹ lưỡng trước mỗi function.

2.2 Chức năng, cách sử dụng

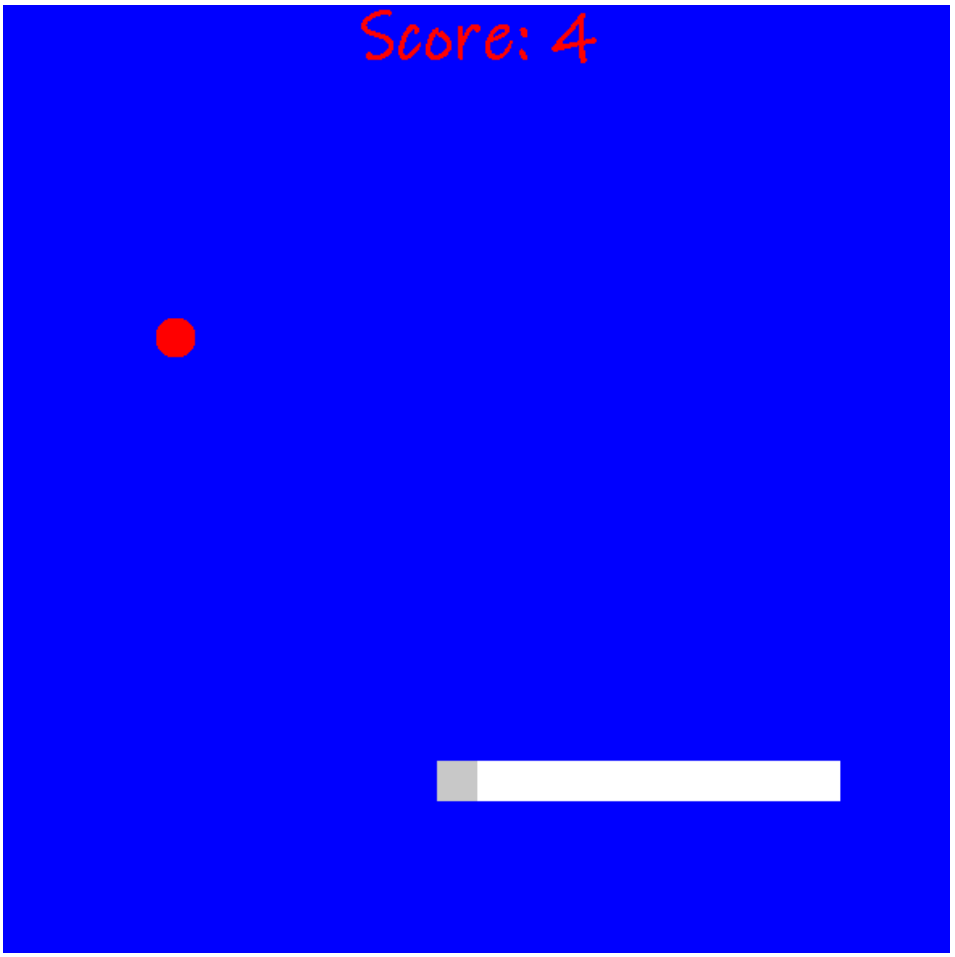
Người chơi clone repo về, sau đó vào src/App.java và chạy. Khi bắt đầu chạy, cửa sổ giao diện chính hiện lên, người chơi chỉ cần nhấn vào nút "Play AI" màu xanh lục, chuyển sang cửa sổ mà ở đó rắn tự chuyển động đến mồi, người chơi chỉ cần xem rắn đi tự động ăn được bao điểm, cho đến khi rắn cắn thân nó, nghĩa là trò chơi kết thúc, một cửa sổ Game Over hiện ra, cho phép người chơi nhấn "Play Again?" để chơi lại.

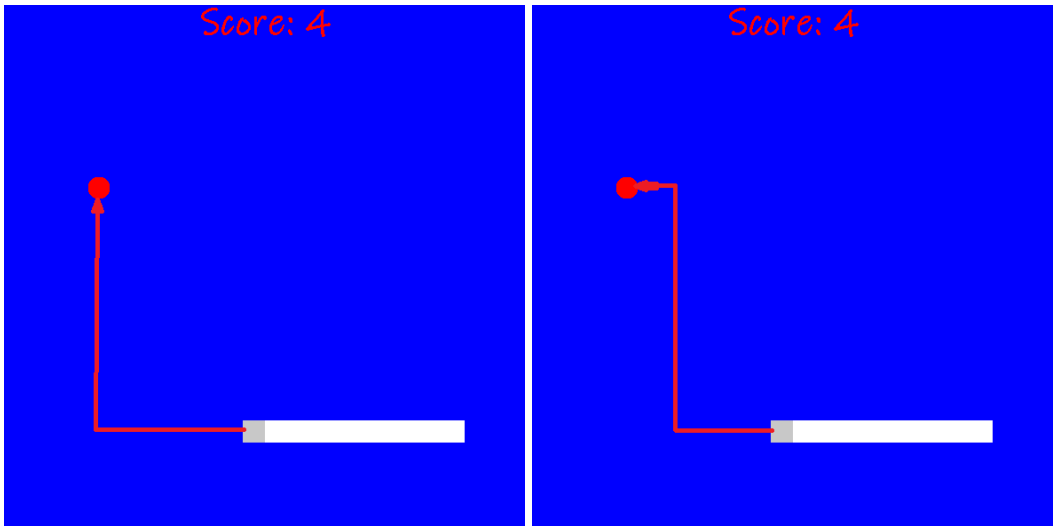
2.3 Mô phỏng trò chơi rắn tự động ăn mồi sử dụng thuật toán A*

- ***Giao diện bắt đầu:***

Ở đây có 2 chế độ rắn săn mồi tự động: Single AI và A* AI (tức rắn tự động săn mồi chỉ dùng thuật toán A*).

- ***Giao diện vào game:***





Hình minh hoạ đường đi của rắn của 2 chế độ Single AI (trái) và A* AI (phải)

Ở đây đã bổ sung thêm chế độ đi xuyên biên trò chơi, dưới đây là mã java về chế độ xuyên biên:

```
// Check if the snake goes through the wall and appears on the other side
    public void checkCollisions() {
```

```
        for(int i = bodyParts; i>0; i--) {
               if((x[0] == x[i]) && (y[0] == y[i])) { // Check if the snake's head
touches the body or not
                   running = false;
               }
           }
           for (int i = 0; i < bodyParts; i++) {
               if (x[i] < 0) {
                   x[i] = SCREEN_WIDTH; //Warp to right
               } else if (x[i] == SCREEN_WIDTH) {
                   x[i] = 0; //Warp to left
               }

               if (y[i] < 0) {
                   y[i] = SCREEN_HEIGHT; //Warp to bottom
               } else if (y[i] == SCREEN_HEIGHT) {
                   y[i] = 0; //Warp to top
               }
           }
           if (!running) {
               timer.stop();
           }
       }
```

- ***Giao diện trò chơi kết thúc:***

Đánh giá: Thuật toán được dùng để điều khiển con rắn đến một đoạn nào đó thân rắn dài ra, do con rắn được điều khiển sao cho chi phí thấp, luôn hướng đến mục tiêu một cách có phần "greedy" mà đi vào đúng chỗ nguy hiểm, bị kẹp vào thân rắn dẫn đến đâm vào thân hoặc cố gắng vùng vẫy thoát không kịp.

Dưới đây là bảng đánh giá kết quả sau 20 lần chạy thử:

STT	Single AI	A* AI
1	33	66
2	23	53
3	13	82
4	26	81
5	34	59
6	34	73
7	10	43
8	3	57
9	27	76
10	30	61
11	14	56
12	17	61
13	39	51
14	13	85
15	27	75
16	53	44
17	19	68
18	29	52
19	21	45
20	23	94

2.4 Đánh giá hiệu quả trò chơi rắn tự động ăn mồi sử dụng thuật toán A*

Đánh giá hiệu quả của trò chơi rắn tự động ăn mồi sử dụng thuật toán A* phụ thuộc vào nhiều yếu tố, bao gồm triển khai cụ thể của thuật toán, cách xử lý trạng thái, cấu trúc dữ liệu và chiến lược tìm kiếm.

Thuật toán A* (A-star) được sử dụng rộng rãi trong các bài toán tìm kiếm đường đi ngắn nhất trên đồ thị. Trong trò chơi rắn, A* có thể được áp dụng để tìm đường đi tối ưu từ vị trí hiện tại của rắn đến mồi. Đánh giá hiệu quả của thuật toán A* trong trò chơi rắn có thể được xem xét từ các khía cạnh sau:

1. Tốc độ tìm kiếm: Thuật toán A* có khả năng tìm kiếm đường đi tối ưu từ điểm A đến điểm B. Tuy nhiên, tốc độ tìm kiếm của A* có thể bị ảnh hưởng bởi kích thước của lưới và số lượng trạng thái có thể xảy ra trong trò chơi rắn. Trong trường hợp có quá nhiều trạng thái hoặc đường đi phức tạp, A* có thể mất nhiều thời gian để tìm kiếm đường đi tối ưu.

2. Tính tối ưu: A* tìm kiếm đường đi tối ưu bằng cách sử dụng hàm đánh giá (heuristic) để ước lượng chi phí còn lại từ điểm hiện tại đến điểm đích. Tuy nhiên, độ tối ưu của A* phụ thuộc vào chất lượng của hàm đánh giá được sử dụng. Nếu hàm đánh giá không đủ chính xác hoặc không phù hợp với bài toán, A* có thể không tìm được đường đi tối ưu.

3. Khả năng xử lý trạng thái: Trong trò chơi rắn, tốc độ xử lý trạng thái của thuật toán A* cũng quan trọng. Mỗi lần tìm kiếm đường đi, A* cần xem xét các trạng thái tiềm năng và tính toán chi phí từ điểm hiện tại đến các trạng thái đó. Việc xử lý trạng thái có thể trở nên đáng kể nếu kích thước của lưới lớn hoặc có quá nhiều trạng thái khả dĩ.

4. Tương tác với môi trường: Trò chơi rắn là một môi trường động, trong đó mồi di chuyển và rắn phải đáp ứng nhanh chóng để không va chạm. A* có thể gặp khó khăn trong việc tìm kiếm đường đi tối ưu khi môi trường thay đổi nhanh chóng. Cần có cơ chế cập nhật và tái tính toán đường đi khi mồi di chuyển hoặc rắn thay đổi vị trí.

Tổng quan, hiệu quả của trò chơi rắn tự động ăn mồi sử dụng thuật toán A* phụ thuộc vào nhiều yếu tố, và việc tối ưu hóa triển khai của thuật toán cũng quan trọng. Để đạt hiệu quả tốt, có thể tối ưu hóa hàm đánh giá (heuristic) và cải tiến chiến lược tìm kiếm.

KẾT LUẬN

Nhận xét về 2 chế độ, ta thấy chế độ SingleAI không dùng Priority Queue để tính gCost, còn AStar thì có, nên ta sẽ thấy là rắn trong chế độ SingleAI hướng đến mồi một cách "tham lam" hơn so với chế độ AStarAI, nên tần số ghi điểm cao của chế độ SingleAI là thấp hơn so với chế độ AStarAI.

Có thể nói về hiệu quả của việc sử dụng thuật toán A* trong việc giúp rắn tự động tìm đường ăn mồi. Thuật toán A* đã cho thấy sự ưu việt trong việc tìm kiếm đường đi ngắn nhất và giúp trò chơi trở nên thú vị hơn.

Xác suất rắn đi xuyên biên ở mức thấp, cần kiên nhẫn chạy thử khoảng 10 lần mới xuất hiện tình huống rắn đi xuyên biên ở một thời điểm nào đó thân rắn dài.

Dự án có ưu điểm như: Sử dụng thuật toán A* tìm đến vị trí nhanh nhất đến vị trí mồi. Thêm thành công chế độ rắn đi qua biên trò chơi và xuất hiện ở biên bên kia, góp phần giảm đi xác suất game over mỗi khi bị kẹp giữa thân rắn và biên của hộp. Sau khi rắn xuyên biên, rắn vẫn có thể đi ăn mồi an toàn mặc dù có sự cố rắn chỉ đi tới điểm UNIT_SIZE -1 tại 4 biên

Bên cạnh đó tồn tại một số nhược điểm như: Tính tương đối khó điều chỉnh: Thuật toán A* thuộc loại đường đi ngắn nhất và không dễ dàng thay đổi được thuật toán và sửa đổi, điều này làm cho trò chơi rắn săn mồi tự động dùng thuật toán A* cũng khó được thay đổi. Do tìm đường tốt nhất đến mồi nó xác suất cao khiến con rắn đi vào đường cùng và cắn vào đuôi/thân chính mình. Sau khi thêm chức năng xuyên biên, vẫn còn tồn tại sự cố ở tại dãy UNIT_SIZE ở 4 thành biên, khi rắn chỉ khi đi tới điểm UNIT_SIZE - 1 là đã sang biên bên kia.

Từ những ưu điểm nhược điểm trên, có những hướng phát triển/cải thiện để cải thiện trò chơi trong đó có tích hợp mô hình Reinforcement Learning (học tăng cường), ở đó rắn "học hỏi kinh nghiệm" từ những thất bại trước mà trở nên "khôn ngoan hơn" để ghi điểm cao hơn. Đồng thời chỉnh sửa lại lỗi 4 biên để không mắc phải sự cố nhược điểm trên, thêm một vài chướng ngại vật trong trò chơi để điều khiển rắn tự động tuỳ cơ ứng biến.

Tài liệu tham khảo

1. Giải thuật tìm kiếm A*: https://en.wikipedia.org/wiki/A*_search_algorithm ,
2. Giải thuật Priority Queue: https://en.wikipedia.org/wiki/Priority_queue .
3. Github project: https://github.com/BToopS2/AI-Snake_Project .
4. Các project tham khảo cho dự án:
   https://github.com/NguyenTrieuVuong/SnakeGame ,
   https://github.com/Yellowatch/Java-Snake-Game ,
   https://github.com/GreenSlime96/PathFinding .
5. Slide báo cáo:
   https://www.canva.com/design/DAFhUExvYTs/Um8cAUvG-JQiFjfGus2M1A/view?utm_content=DAFhUExvYTs&utm_campaign=designshare&utm_medium=link&utm_source=publishsharelink .