**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**

**KHOA CÔNG NGHỆ THÔNG TIN**

**BỘ MÔN MẠNG VÀ CÁC HỆ THỐNG THÔNG TIN**

---------o0o---------



**BÁO CÁO XÂY DỰNG CHƯƠNG TRÌNH**

**HỌC PHẦN TRÍ TUỆ NHÂN TẠO**

**Giảng viên hướng dẫn:**

Giảng viên*: TS.Phạm Thanh Hà*

**Sinh viên thực hiện:**

| **STT** | **Họ và tên** | **Mã sinh viên** | **Lớp** |
| --- | --- | --- | --- |
| **1** | **Nguyễn Duy Hiếu** | **191200410** | **CNTT4-K60** |

*Hà Nội, Tháng 12 Năm 2022*

I.Thuật toán tìm kiếm mù

* Bài toán triệu phú và kẻ cướp

II.Thuật toán tìm kiếm có đối thủ

* Trò chơi cờ caro

I.Thuật toán tìm kiếm mù

1.Bài toán triệu phú và kẻ cướp.

Có 3 triệu phú với 3 tên cướp và một chiếc thuyền ở bên bờ tả của một con sông.

Thuyền chỉ chở được một hoặc 2 người.

Hãy tìm cách đưa 3 triệu phú sang sông sao cho mỗi bờ sông số kẻ cướp không được lớn hơn số triệu phú.

1. **Mô tả thuật toán.**

Dùng một bộ 3 số (a,b,k) để biểu diễn các trạng thái

   a - số triệu phú, b - số kẻ cướp ở bờ tả

k=true thuyền đang ở bờ tả, ngược lại k=false

**Không gian trạng thái được xác định như sau:**

TTĐ:(3,3,true)

TTKT:(0,0,false)

Tập các toán tử bao gồm các toán tử:

+Thuyền chở 1 kẻ cướp

+Thuyền chở 1 triệu phú

+Thuyền chở 1 triệu phú, 1 kẻ cướp

+Thuyền chở 2 kẻ cướp

+Thuyền chở 2 triệu phú

Tại mỗi bước ta chọn trạng thái để phát triển là trạng thái được sinh ra trước các trạng thái chờ phát triển khác

Sử dụng danh sách L để lưu các trạng thái được sinh ra và chờ được phát triển

Mục tiêu của tìm kiếm là tìm đường đi từ trạng thái đầu đến trạng thái đích nên ta cần lưu vết của đường đi, ta có thể dùng hàm father để lưu lại cha của mỗi đỉnh trên đường đi, father(v) = u nếu u là cha của v

1. **Cài đặt.**

Xây dựng lớp State để lưu các trạng thái của bài toán:

**public class** State {

**public int millionaires**;

**public int robbers**;

**public boolean isBoatOnLeft**;

**public** State() {

**this**.**millionaires** = 3;

**this**.**robbers** = 3;

**this**.**isBoatOnLeft** = **true**;

}

**public** State(**int** millionaires, **int** robbers, **boolean** isBoatOnLeft) {

**this**.**millionaires** = millionaires;

**this**.**robbers** = robbers;

**this**.**isBoatOnLeft** = isBoatOnLeft;

}

@Override

**public** String toString() {

**return "("** + **this**.**millionaires** + **", "**

+ **this**.**robbers** + **", "**

+ **this**.**isBoatOnLeft** + **")"**;

}

@Override

**public boolean** equals(Object o) {

**if** (**this** == o) **return true**;

**if** (o == **null** || getClass() != o.getClass()) **return false**;

State state = (State) o;

**return millionaires** == state.**millionaires** && **robbers** == state.**robbers** && **isBoatOnLeft** == state.**isBoatOnLeft**;

}

@Override

**public int** hashCode() {

**return** Objects.*hash*(**millionaires**, **robbers**, **isBoatOnLeft**);

}

}

Xây dựng lớp Main để cài đặt thuật toán tìm kiếm:

**public class** Main{

**private static** State *state* = **new** State();

**private static** State *firstState* = **new** State(3, 3, **true**);

**private static** State *finishState* = **new** State(0, 0, **false**);

**private static** Stack<State> *stateStack* = **new** Stack<>();

**private static** PriorityQueue<State> *priorityQueue*;

**private static** Map<State, Integer> *checkMap* = **new** HashMap<>();

**private static** Map<State, State> *familyMap* = **new** HashMap<>();

**private static** Stack<State> *stack* = **new** Stack<>();

**private static int**[][] *instance* = {{1, 0}, {0, 1}, {1, 1}, {2, 0}, {0, 2}};

**public static void** Work3() {

*stateStack*.push(*firstState*);

*checkMap*.put(*state*, 1);

**while** (*state* != *finishState*) {

*priorityQueue* = **new** PriorityQueue<>(**new** Comparator<State>() {

@Override

**public int** compare(State o1, State o2) {

**if** (o1.**isBoatOnLeft** && o2.**isBoatOnLeft**) {

**return** *sum*(o1) < *sum*(o2) ? 1 : o1.**millionaires** > o2.**millionaires** ? 1 : -1;

}

**if** (!o1.**isBoatOnLeft** && !o2.**isBoatOnLeft**) {

**return** o1.**robbers** < o2.**robbers** ? 1 : *sum*(o1) < *sum*(o2) ? 1 : -1;

}

**return** 0;

}

});

*state* = *stateStack*.pop();

System.***out***.print(*state* + **": "**);

**if** (*state*.equals(*finishState*)) {

*Track*();

**return**;

}

**for** (**int**[] i : *instance*) {

**if** (*state*.**isBoatOnLeft** && *CheckLeft*(*state*, i)) {

State rs1 = *Minus*(*state*, i);

**if** (*checkMap*.containsKey(rs1) == **false**) {

*checkMap*.put(rs1, 1);

*familyMap*.put(rs1, *state*);

*priorityQueue*.add(rs1);

System.***out***.print(rs1 + **", "**);

}

}

**if** (!*state*.**isBoatOnLeft** && *CheckRight*(*state*, i)) {

State rs2 = *Plus*(*state*, i);

**if** ( *checkMap*.containsKey(rs2) == **false** ) {

*checkMap*.put(rs2, 1);

*familyMap*.put(rs2, *state*);

*priorityQueue*.add(rs2);

System.***out***.print(rs2 + **", "**);

}

}

}

System.***out***.print(**"[ "**);

State s;

**while**((s = *priorityQueue*.poll()) != **null**) {

*stateStack*.push(s);

System.***out***.print(s + **" "**);

}

System.***out***.print(**"]"**);

System.***out***.println();

}

}

**private static int** sum(State s) {

**return** s.**millionaires** + s.**robbers**;

}

**private static void** Track() {

System.***out***.println();

System.***out***.println(**"Tracking: "**);

State child = *finishState*;

*stack*.push(child);

**while** (!child.equals(*firstState*)) {

State father = *familyMap*.get(child);

*stack*.push(father);

child = father;

}

System.***out***.printf(**""** + *stack*.peek());

*stack*.pop();

**while** (!*stack*.isEmpty()) {

System.***out***.printf(**" --> "** + *stack*.peek());

*stack*.pop();

}

}

**private static boolean** CheckLeft(State s, **int**[] pair) {

**return** (s.**millionaires** >= pair[0] && s.**robbers** >= pair[1]

&& s.**millionaires** - pair[0] >= s.**robbers** - pair[1]);

}

**private static boolean** CheckRight(State s, **int**[] pair) {

**return** (s.**millionaires** + pair[0] <= 3 && s.**robbers** + pair[1] <= 3

&& s.**millionaires** + pair[0] >= s.**robbers** + pair[1]);

}

**private static** State Minus(State s, **int**[] pair) {

**return new** State(s.**millionaires** - pair[0], s.**robbers** - pair[1], !s.**isBoatOnLeft**);

}

**private static** State Plus(State s, **int**[] pair) {

**return new** State(s.**millionaires** + pair[0], s.**robbers** + pair[1], !s.**isBoatOnLeft**);

}

}

Chạy chương trình bằng lớp Run:

**public class** Run {

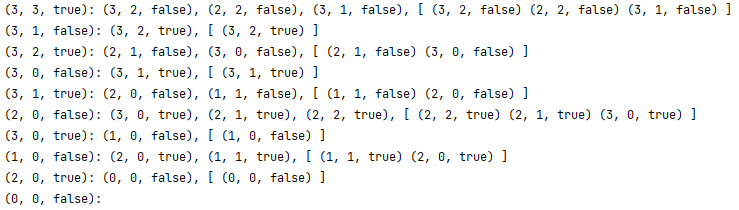
**public static void** main(String[] args) {

Main.*Work3*();

}

}

1. **Kết quả thực hiện**







II.Thuật toán tìm kiếm có đối thủ

1.Trò chơi cờ caro.

1. **Mô tả thuật toán.**

**Sử dụng chiến lược minimax để tìm ra nước đi tốt nhất.**

- Quá trình chơi cờ là quá trình Trắng và Đen thay phiên nhau đưa ra quyết định, thực hiện một trong số các nước đi hợp lệ. Trên cây trò chơi, quá trình đó sẽ tạo ra đường đi từ gốc tới lá

- Giả sử tới một thời điểm nào đó, đường đi đã dẫn tới đỉnh u. Nếu u là đỉnh Trắng thì Trắng cần chọn đi tới một trong các đỉnh Đen v là con u. Tại đỉnh Đen v mà Trắng vừa chọn. Đen sẽ phải chọn đi tới một trong các đỉnh trắng w là con của v.

- Quá trình trên sẽ dừng lại khi đạt tới một đỉnh là lá của cây

- Giả sử Trắng cần tìm nước đi tại đỉnh u. Nước đi tối ưu cho Trắng là nước đi dẫn tới đỉnh con v, v là đỉnh tốt nhất (cho Trắng) trong số các đỉnh con của u

- Ta cần giả thiết rằng, đến lượt đối thủ chọn nước đi từ v, Đen cũng sẽ chọn nước đi tốt nhất cho anh ta

- Như vậy để chọn nước đi tối ưu cho Trắng tại đỉnh u, ta cần phải xác định giá trị các đỉnh của cây trò chơi gốc u

- Chú ý giá trị của các đỉnh lá (đỉnh ứng với các trạng thái kết thúc) là giá trị của hàm kết cuộc

- Đỉnh có giá trị càng lớn càng tốt cho Trắng, đỉnh có có trị càng nhỏ càng tốt cho Đen. Để xác định giá trị các định của cây trò chơi gốc u, ta đi từ mức thấp nhất lên gốc u.

- Giả sử v là đỉnh trong của cây và giá trị các đỉnh con của nó đã được xác định. Khi đó nếu v là đỉnh Trắng thì giá trị của nó được xác định là giá trị lớn nhất trong các giá trị của các đỉnh con. Nếu v là đỉnh Đen thì giá trị của nó là giá trị nhỏ nhất trong các giá trị của các đỉnh con

- **Hạn chế không gian tìm kiếm**, cụ thể khi cần xác định nước đi cho trắng tại u ta chỉ xem xét cây trò chơi gốc u tới độ cao h nào đó

- Vẫn sử dụng chiến lược MINIMAX để tìm kiếm nước đi

- Tuy nhiên các lá của cây trò chơi hạn chế này có thể không phải là trạng thái kết thúc cho nên hàm kết cuộc không dùng được, thay vào đó ta sử dụng ***hàm đánh giá*** cho các lá của cây trò chơi hạn chế này

Hàm đánh giá eval ứng với mỗi trạng thái u của trò chơi với một giá trị eval(u), giá trị này là sự đánh giá “độ lợi thế” của trạng thái u.

Trạng thái u càng lợi cho Trắng nếu eval(u) là số dương càng lớn

Trạng thái u càng thuận lợi cho Đen nếu eval(u) là số âm càng nhỏ

Nếu eval(u) ≈ 0 thì không có lợi thế cho ai cả

Nếu eval(u) = +∞ thì Trắng thắng và eval(u) = -∞ thì Đen thắng

Chất lượng của chương trình chơi cơ phụ thuộc rất nhiều vào hàm đánh giá

**Trong trò chơi cờ caro này ta sẽ sử dụng hàm lượng giá eval() được quy ước như sau:**

Sau đây là quy ước hàm lượng giá đối với bên Max, bên Min cũng vậy nhưng thêm dấu “-” cho mỗi giá trị đạt được.

1.

Trả về 100000 nếu bên Max thắng.

Trả về 10000 đối với:

Trạng thái dãy 4 mở 2 đầu (openFour).

Hai dãy 4 mở 1 đầu (halfOpenFour).

2.

Trả về 5000 đối với 1 dãy 3 mở hai đầu + 1 dãy 4 mở 1 đầu.

Trả về 3000 đối với 2 dãy 3 mở hai đầu.

Trả về 1000 đối với 1 dãy 3 mở hai đầu + 1 dãy 3 mở một đầu.

3.

Trả về 500 đối với 1 dãy 4 mở một đầu.

Trả về 200 đối với 1 dãy 3 mở hai đầu.

Trả về 100 đối với 2 dãy 2 mở hai đầu.

4.

Trả về 50 đối với trạng thái chỉ có 1 dãy 3 mở một đầu.

Trả về 10 đối với trạng thái có 1 cặp dãy 2 mở một đầu.

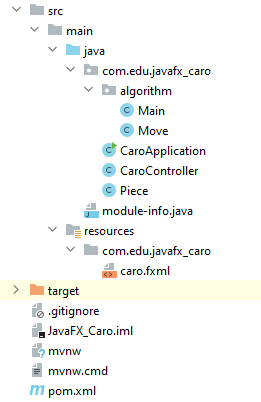
Trả về 5 đối với 1 dãy 2 mở hai đầu.

Trả về 3 đối với 1 dãy 2 mở một đầu.

Trả về 0 điểm cho các trạng thái còn lại và trạng thái hòa.

1. **Cài đặt**

Sử dụng Java FX để xây dựng đồ họa cho trò chơi cờ caro.



Xây dựng lớp Move để định nghĩa một nước đi:

**public class** Move {

**public int x**;

**public int y**;

**public boolean isHuman**;

**public** Move() {};

**public** Move(**int** x, **int** y) {

**this**.**x** = x;

**this**.**y** = y;

}

**public** Move(**int** x, **int** y, **boolean** isHuman) {

**this**.**x** = x;

**this**.**y** = y;

**this**.**isHuman** = isHuman;

}

**public** Object clone() **throws** CloneNotSupportedException {

**return super**.clone();

}

@Override

**public boolean** equals(Object o) {

**if** (**this** == o) **return true**;

**if** (o == **null** || getClass() != o.getClass()) **return false**;

Move move = (Move) o;

**return x** == move.**x** && **y** == move.**y**;

}

@Override

**public int** hashCode() {

**return** Objects.*hash*(**x**, **y**);

}

@Override

**public** String toString() {

**return "Move{"** +

**"x="** + **x** +

**", y="** + **y** +

**'}'**;

}

}

Xây dựng lớp Main chứa hàm miniMax:

**public class** Main {

**public final static int *BOARD\_SIZE*** = 11;

**public final static int *DEPTH*** = 2;

**public int**[][] **board**;

**public** List<Move> **listMove** = **new** LinkedList<>();

**public** List<Integer>[] **arrayListHorizontal** = **new** ArrayList[***BOARD\_SIZE***];

**public** List<Integer>[] **arrayListVertical** = **new** ArrayList[***BOARD\_SIZE***];

**public** List<Integer>[] **arrayListDiagonal** = **new** ArrayList[2 \* ***BOARD\_SIZE*** - 1];

**public** List<Integer>[] **arrayListReverseDiagonal** = **new** ArrayList[2 \* ***BOARD\_SIZE*** - 1];

**public boolean isHuman**;

**public** Main() {

**board** = **new int**[***BOARD\_SIZE***][***BOARD\_SIZE***];

**for** (**int** i = 0; i < ***BOARD\_SIZE***; i++) {

**arrayListHorizontal**[i] = **new** ArrayList<>();

}

**for** (**int** i = 0; i < ***BOARD\_SIZE***; i++) {

**arrayListVertical**[i] = **new** ArrayList<>();

}

**for** (**int** i = 0; i < 2 \* ***BOARD\_SIZE*** - 1; i++) {

**arrayListDiagonal**[i] = **new** ArrayList<>();

}

**for** (**int** i = 0; i < 2 \* ***BOARD\_SIZE*** - 1; i++) {

**arrayListReverseDiagonal**[i] = **new** ArrayList<>();

}

convertHorizontal(**board**, ***BOARD\_SIZE***);

convertVerticalToHorizontal(**board**, ***BOARD\_SIZE***);

convertDiagonalToHorizontal(**board**, ***BOARD\_SIZE***);

convertReverseDiagonalToHorizontal(**board**, ***BOARD\_SIZE***);

}

**private void** convertHorizontal(**int**[][] arr, **int** n) {

**for** (**int** i = 0; i < n; i++)

**arrayListHorizontal**[i].clear();

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < n; j++) {

**arrayListHorizontal**[i].add(arr[i][j]);

}

}

}

**private void** convertVerticalToHorizontal(**int**[][] arr, **int** n) {

**for** (**int** i = 0; i < n; i++)

**arrayListVertical**[i].clear();

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < n; j++) {

**arrayListVertical**[i].add(arr[j][i]);

}

}

}

**private void** convertDiagonalToHorizontal(**int**[][] arr, **int** n) {

**for** (**int** i = 0; i < 2 \* n - 1; i++)

**arrayListDiagonal**[i].clear();

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = i; j >= 0; j--) {

**arrayListDiagonal**[i].add(arr[j][i - j]);

}

}

**for** (**int** i = n; i < 2 \* n - 1; i++) {

**for** (**int** j = n - 1; j >= i + 1 - n; j--) {

**arrayListDiagonal**[i].add(arr[j][i - j]);

}

}

}

**private void** convertReverseDiagonalToHorizontal(**int**[][] arr, **int** n) {

**for** (**int** i = 0; i < 2 \* n - 1; i++)

**arrayListReverseDiagonal**[i].clear();

**for** (**int** i = n - 1; i >= 0; i--) {

**for** (**int** j = i; j <= n - 1; j++) {

**arrayListReverseDiagonal**[n - 1 - i].add(arr[j][j - i]);

}

}

**for** (**int** i = 1; i <= n - 1; i++) {

**for** (**int** j = i; j <= n - 1; j++) {

**arrayListReverseDiagonal**[n - 1 + i].add(arr[j - i][j]);

}

}

}

*// ---------------*

**public** Move bestMove() {

**int** bestScore = -10000000;

Move move = **null**;

**for** (Move m : generate(**this**.**board**)) {

**if** (**board**[m.**x**][m.**y**] == 0) {

**board**[m.**x**][m.**y**] = -1;

**int** score = miniMax(**board**, ***DEPTH***, **false**, -10000000);

score += miniMax(**board**, ***DEPTH*** - 1, **false**, -10000000);

score += miniMax(**board**, ***DEPTH*** - 2, **false**, -10000000);

score /= 3;

**board**[m.**x**][m.**y**] = 0;

**if** (score > bestScore) {

bestScore = score;

move = **new** Move(m.**x**, m.**y**);

}

}

}

**board**[move.**x**][move.**y**] = -1;

**isHuman** = **true**;

**listMove**.add(**new** Move(move.**x**, move.**y**, **false**));

System.***out***.println(**"Score: "** + bestScore);

**return** move;

}

**public int** miniMax(**int**[][] board, **int** depth, Boolean isMaximizing, **int** alpha) {

**int** result = checkWin();

**if** (result == -1) {

**return** 100000;

}

**if** (result == 1) {

**return** -100000;

}

**if** (result == 2) {

**return** 0;

}

**if** (depth == 0) {

**return** -eval(board);

}

**if** (isMaximizing) {

**int** bestScore = -1000000;

**for** (Move m : generate(board)) {

**if** (board[m.**x**][m.**y**] == 0) {

board[m.**x**][m.**y**] = -1;

**int** score = miniMax(board, depth - 1, **false**, bestScore);

board[m.**x**][m.**y**] = 0;

*// if (alpha < score) {*

*// return bestScore;*

*// }*

bestScore = Math.*max*(score, bestScore);

}

}

**return** bestScore;

} **else** {

**int** bestScore = 1000000;

**for** (Move m : generate(board))

**if** (board[m.**x**][m.**y**] == 0) {

board[m.**x**][m.**y**] = 1;

**int** score = miniMax(board, depth - 1, **true**, bestScore);

board[m.**x**][m.**y**] = 0;

*// if (alpha > score) {*

*// return bestScore;*

*// }*

bestScore = Math.*min*(score, bestScore);

}

**return** bestScore;

}

}

**public void** addHumanMove(**int** x, **int** y) {

**this**.**board**[x][y] = 1;

**this**.**listMove**.add(**new** Move(x, y, **true**));

}

**private** List<Move> generate(**int**[][] board) {

List<Move> moves = **new** ArrayList<>();

**if** (**listMove**.size() == 0) {

moves.add(**new** Move(Math.*round*(***BOARD\_SIZE*** / 2), Math.*round*(***BOARD\_SIZE*** / 2), **false**));

**return** moves;

}

**int** left = 100;

**int** right = -100;

**int** top = 100;

**int** bottom = -100;

**for** (Move move : **listMove**) {

**if** (move.**isHuman**) {

left = left > move.**y** ? move.**y** : left;

right = right < move.**y** ? move.**y** : right;

top = top > move.**x** ? move.**x** : top;

bottom = bottom < move.**x** ? move.**x** : bottom;

}

}

**int** LEFT = left - 2 >= 0 ? left - 1 : 0;

**int** RIGHT = right + 2 <= ***BOARD\_SIZE*** - 1 ? right + 2 : ***BOARD\_SIZE*** - 1;

**int** TOP = top - 2 >= 0 ? top - 1 : 0;

**int** BOTTOM = bottom + 2 <= ***BOARD\_SIZE*** - 1 ? bottom + 2 : ***BOARD\_SIZE*** - 1;

**for** (**int** i = TOP; i <= BOTTOM; i++) {

**for** (**int** j = LEFT; j <= RIGHT; j++) {

**if** (board[i][j] == 0) {

moves.add(**new** Move(i, j));

}

}

}

**if** (moves.size() != 0)

**return** moves;

**for** (**int** i = 0; i < ***BOARD\_SIZE***; i++) {

**for** (**int** j = 0; j < ***BOARD\_SIZE***; j++) {

**if** (board[i][j] == 0) {

moves.add(**new** Move(i, j));

}

}

}

**return** moves;

}

**public int** checkWin() {

**int** winner = 0;

convertHorizontal(**board**, ***BOARD\_SIZE***);

convertVerticalToHorizontal(**board**, ***BOARD\_SIZE***);

convertDiagonalToHorizontal(**board**, ***BOARD\_SIZE***);

convertReverseDiagonalToHorizontal(**board**, ***BOARD\_SIZE***);

**for** (List<Integer> list : **arrayListHorizontal**) {

**int** rs = checkListWin(list);

**if** (rs == -1) winner = -1;

**if** (rs == 1) winner = 1;

}

**for** (List<Integer> list : **arrayListVertical**) {

**int** rs = checkListWin(list);

**if** (rs == -1) winner = -1;

**if** (rs == 1) winner = 1;

}

**for** (List<Integer> list : **arrayListDiagonal**) {

**int** rs = checkListWin(list);

**if** (rs == -1) **return** winner = -1;

**if** (rs == 1) **return** winner = 1;

}

**for** (List<Integer> list : **arrayListReverseDiagonal**) {

**int** rs = checkListWin(list);

**if** (rs == -1) **return** winner = -1;

**if** (rs == 1) **return** winner = 1;

}

**int** openSpots = 0;

**for** (**int** i = 0; i < ***BOARD\_SIZE***; i++) {

**for** (**int** j = 0; j < ***BOARD\_SIZE***; j++) {

**if** (**board**[i][j] == 0) {

openSpots++;

}

}

}

**if** (winner == 0 && openSpots == 0) {

**return** 2;

} **else** {

**return** winner;

}

}

**public int** eval(**int**[][] board) {

**int** val = 0;

convertHorizontal(board, ***BOARD\_SIZE***);

convertVerticalToHorizontal(board, ***BOARD\_SIZE***);

convertDiagonalToHorizontal(board, ***BOARD\_SIZE***);

convertReverseDiagonalToHorizontal(board, ***BOARD\_SIZE***);

**int** blackOpen4 = 0;

**int** whiteOpen4 = 0;

**int** blackHalf4 = 0;

**int** whiteHalf4 = 0;

**int** blackOpen3 = 0;

**int** whiteOpen3 = 0;

**int** blackHalf3 = 0;

**int** whiteHalf3 = 0;

**int** blackOpen2 = 0;

**int** whiteOpen2 = 0;

**int** blackHalf2 = 0;

**int** whiteHalf2 = 0;

*// Horizontal and Vertical*

**for** (List<Integer> list : **arrayListHorizontal**) {

**int** rs = checkListOpen4(list);

**if** (rs == -1) blackOpen4++;

**if** (rs == 1) whiteOpen4++;

rs = checkListHalf4(list);

**if** (rs == -1) blackHalf4++;

**if** (rs == 1) whiteHalf4++;

rs = checkListOpen3(list);

**if** (rs == -1) blackOpen3++;

**if** (rs == 1) whiteOpen3++;

rs = checkListHalf3(list);

**if** (rs == -1) blackHalf3++;

**if** (rs == 1) whiteHalf3++;

rs = checkListOpen2(list);

**if** (rs == -1) blackOpen2++;

**if** (rs == 1) whiteOpen2++;

rs = checkListHalf2(list);

**if** (rs == -1) blackHalf2++;

**if** (rs == 1) whiteHalf2++;

}

**for** (List<Integer> list : **arrayListVertical**) {

**int** rs = checkListOpen4(list);

**if** (rs == -1) blackOpen4++;

**if** (rs == 1) whiteOpen4++;

rs = checkListHalf4(list);

**if** (rs == -1) blackHalf4++;

**if** (rs == 1) whiteHalf4++;

rs = checkListOpen3(list);

**if** (rs == -1) blackOpen3++;

**if** (rs == 1) whiteOpen3++;

rs = checkListHalf3(list);

**if** (rs == -1) blackHalf3++;

**if** (rs == 1) whiteHalf3++;

rs = checkListOpen2(list);

**if** (rs == -1) blackOpen2++;

**if** (rs == 1) whiteOpen2++;

rs = checkListHalf2(list);

**if** (rs == -1) blackHalf2++;

**if** (rs == 1) whiteHalf2++;

}

*// Diagonal and Reverse diagonal*

**for** (List<Integer> list : **arrayListDiagonal**) {

**int** rs = checkListOpen4(list);

**if** (rs == -1) blackOpen4++;

**if** (rs == 1) whiteOpen4++;

rs = checkListHalf4(list);

**if** (rs == -1) blackHalf4++;

**if** (rs == 1) whiteHalf4++;

rs = checkListOpen3(list);

**if** (rs == -1) blackOpen3++;

**if** (rs == 1) whiteOpen3++;

rs = checkListHalf3(list);

**if** (rs == -1) blackHalf3++;

**if** (rs == 1) whiteHalf3++;

rs = checkListOpen2(list);

**if** (rs == -1) blackOpen2++;

**if** (rs == 1) whiteOpen2++;

rs = checkListHalf2(list);

**if** (rs == -1) blackHalf2++;

**if** (rs == 1) whiteHalf2++;

}

**for** (List<Integer> list : **arrayListReverseDiagonal**) {

**int** rs = checkListOpen4(list);

**if** (rs == -1) blackOpen4++;

**if** (rs == 1) whiteOpen4++;

rs = checkListHalf4(list);

**if** (rs == -1) blackHalf4++;

**if** (rs == 1) whiteHalf4++;

rs = checkListOpen3(list);

**if** (rs == -1) blackOpen3++;

**if** (rs == 1) whiteOpen3++;

rs = checkListHalf3(list);

**if** (rs == -1) blackHalf3++;

**if** (rs == 1) whiteHalf3++;

rs = checkListOpen2(list);

**if** (rs == -1) blackOpen2++;

**if** (rs == 1) whiteOpen2++;

rs = checkListHalf2(list);

**if** (rs == -1) blackHalf2++;

**if** (rs == 1) whiteHalf2++;

}

*// One open four = 10000*

*// Two open half four = 10000*

*// One open three + one open half four = 5000*

*// Two open three = 3000*

*// One open three + one open half three = 1000*

*// One open half four = 500*

*// One open three = 200*

*// Two open two = 100*

*// One open half three = 50*

*// Two open half two = 10*

*// One open two = 5*

*// one open half two = 3*

**if** (blackOpen4 > 0)

val += blackOpen4 \* -10000;

**if** (blackHalf4 >= 2)

val += blackHalf4 / 2 \* -10000;

**if** (blackHalf4 > 0 && blackOpen4 > 0)

val += -5000;

**if** (blackOpen3 >= 2)

val += blackOpen3 / 2 \* -3000;

**if** (blackOpen3 > 0 && blackHalf3 > 0)

val += -1000;

**if** (blackHalf4 > 0)

val += blackHalf4 \* -500;

**if** (blackOpen3 > 0)

val += blackOpen3 \* -200;

**if** (blackOpen2 >= 2)

val += blackOpen2 / 2 \* -100;

**if** (blackHalf3 > 0)

val += blackHalf3 \* -50;

**if** (blackHalf2 >= 2)

val += blackHalf2 \* -10;

**if** (blackOpen2 > 0)

val += blackOpen2 \* -5;

**if** (blackHalf2 > 0)

val += blackHalf2 \* -3;

**if** (whiteOpen4 > 0)

val += whiteOpen4 \* 10000;

**if** (whiteHalf4 >= 2)

val += whiteHalf4 / 2 \* 10000;

**if** (whiteHalf4 > 0 && whiteOpen4 > 0)

val += 5000;

**if** (whiteOpen3 >= 2)

val += whiteOpen3 / 2 \* 3000;

**if** (whiteOpen3 > 0 && whiteHalf3 > 0)

val += 1000;

**if** (whiteHalf4 > 0)

val += whiteHalf4 \* 500;

**if** (whiteOpen3 > 0)

val += whiteOpen3 \* 200;

**if** (whiteOpen2 >= 2)

val += whiteOpen2 / 2 \* 100;

**if** (whiteHalf3 > 0)

val += whiteHalf3 \* 50;

**if** (whiteHalf2 >= 2)

val += whiteHalf2 \* 10;

**if** (whiteOpen2 > 0)

val += whiteOpen2 \* 5;

**if** (whiteHalf2 > 0)

val += whiteHalf2 \* 3;

**return** val;

}

**public int** checkListWin(List<Integer> list) {

**if** (list.size() < 5)

**return** 0;

**for** (**int** i = 0; i < list.size() - 4; i++) {

**if** (

list.get(i) == -1 &&

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == -1 &&

list.get(i + 4) == -1) {

**return** -1;

}

**if** (

list.get(i) == 1 &&

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 1 &&

list.get(i + 4) == 1) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListOpen4(List<Integer> list) {

**if** (list.size() < 6)

**return** 0;

**for** (**int** i = 0; i < list.size() - 5; i++) {

**if** (

list.get(i) == 0 &&

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == -1 &&

list.get(i + 4) == -1 &&

list.get(i + 5) == 0) {

**return** -1;

}

**if** (

list.get(i) == 0 &&

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 1 &&

list.get(i + 4) == 1 &&

list.get(i + 5) == 0) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListHalf4(List<Integer> list) {

**if** (list.size() < 5)

**return** 0;

**if** (

list.get(0) == 0 &&

list.get(1) == -1 &&

list.get(2) == -1 &&

list.get(3) == -1 &&

list.get(4) == -1) {

**return** -1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == -1 &&

list.get(list.size() - 3) == -1 &&

list.get(list.size() - 4) == -1 &&

list.get(list.size() - 5) == -1) {

**return** -1;

}

**if** (

list.get(0) == 0 &&

list.get(1) == 1 &&

list.get(2) == 1 &&

list.get(3) == 1 &&

list.get(4) == 1) {

**return** 1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == 1 &&

list.get(list.size() - 3) == 1 &&

list.get(list.size() - 4) == 1 &&

list.get(list.size() - 5) == 1) {

**return** 1;

}

**if** (list.size() == 5) {

**return** 0;

}

**for** (**int** i = 0; i < list.size() - 5; i++) {

**if** (

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == -1 &&

list.get(i + 4) == -1 &&

((list.get(i + 5) == 0 && list.get(i) == 1) || (list.get(i + 5) == 1 && list.get(i) == 0))) {

**return** -1;

}

**if** (

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 1 &&

list.get(i + 4) == 1 &&

((list.get(i + 5) == 0 && list.get(i) == -1) || (list.get(i + 5) == -1 && list.get(i) == 0))) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListOpen3(List<Integer> list) {

**if** (list.size() < 5)

**return** 0;

**for** (**int** i = 0; i < list.size() - 4; i++) {

**if** (

list.get(i) == 0 &&

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == -1 &&

list.get(i + 4) == 0) {

**return** -1;

}

**if** (

list.get(i) == 0 &&

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 1 &&

list.get(i + 4) == 0) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListHalf3(List<Integer> list) {

**if** (list.size() < 4)

**return** 0;

**if** (

list.get(0) == 0 &&

list.get(1) == -1 &&

list.get(2) == -1 &&

list.get(3) == -1) {

**return** -1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == -1 &&

list.get(list.size() - 3) == -1 &&

list.get(list.size() - 4) == -1) {

**return** -1;

}

**if** (

list.get(0) == 0 &&

list.get(1) == 1 &&

list.get(2) == 1 &&

list.get(3) == 1) {

**return** 1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == 1 &&

list.get(list.size() - 3) == 1 &&

list.get(list.size() - 4) == 1) {

**return** 1;

}

**if** (list.size() == 4) {

**return** 0;

}

**for** (**int** i = 0; i < list.size() - 4; i++) {

**if** (

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == -1 &&

((list.get(i + 4) == 0 && list.get(i) == 1) || (list.get(i + 4) == 1 && list.get(i) == 0))) {

**return** -1;

}

**if** (

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 1 &&

((list.get(i + 4) == 0 && list.get(i) == -1) || (list.get(i + 4) == -1 && list.get(i) == 0))) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListOpen2(List<Integer> list) {

**if** (list.size() < 4)

**return** 0;

**for** (**int** i = 0; i < list.size() - 3; i++) {

**if** (

list.get(i) == 0 &&

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

list.get(i + 3) == 0) {

**return** -1;

}

**if** (

list.get(i) == 0 &&

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

list.get(i + 3) == 0) {

**return** 1;

}

}

**return** 0;

}

**public int** checkListHalf2(List<Integer> list) {

**if** (list.size() < 3)

**return** 0;

**if** (

list.get(0) == 0 &&

list.get(1) == -1 &&

list.get(2) == -1) {

**return** -1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == -1 &&

list.get(list.size() - 3) == -1) {

**return** -1;

}

**if** (

list.get(0) == 0 &&

list.get(1) == 1 &&

list.get(2) == 1) {

**return** 1;

}

**if** (

list.get(list.size() - 1) == 0 &&

list.get(list.size() - 2) == 1 &&

list.get(list.size() - 3) == 1) {

**return** 1;

}

**if** (list.size() == 3) {

**return** 0;

}

**for** (**int** i = 0; i < list.size() - 3; i++) {

**if** (

list.get(i + 1) == -1 &&

list.get(i + 2) == -1 &&

((list.get(i + 3) == 0 && list.get(i) == 1) || (list.get(i + 3) == 1 && list.get(i) == 0))) {

**return** -1;

}

**if** (

list.get(i + 1) == 1 &&

list.get(i + 2) == 1 &&

((list.get(i + 3) == 0 && list.get(i) == -1) || (list.get(i + 3) == -1 && list.get(i) == 0))) {

**return** 1;

}

}

**return** 0;

}

}

Xây dựng lớp Piece dùng để hỗ trợ hiển thị đồ quân cờ trên màn hình đồ họa:

**public class** Piece {

**private double x**;

**private double y**;

**private double radius**;

**private** Circle **c**;

**public** Piece(**double** x, **double** y, **double** radius, Circle c) {

**this**.**x** = x;

**this**.**y** = y;

**this**.**radius** = radius;

**this**.**c** = c;

}

**public void** setY(**double** y) {

**this**.**y** = y;

}

**public void** setX(**double** x) {

**this**.**x** = x;

}

**public double** getY() {

**return y**;

}

**public double** getX() {

**return x**;

}

**public** Circle getCircle() {

**return c**;

}

**public void** setColor(Color color) {

**c**.setFill(color);

}

**public void** draw() {

**c**.setRadius(**radius**);

**c**.setTranslateX(**x**);

**c**.setTranslateY(**y**);

}

@Override

**public** String toString() {

**return "Piece{"** +

**"x="** + **x** +

**", y="** + **y** +

**", radius="** + **radius** +

**", c="** + **c** +

**'}'**;

}

}

Xây dựng lớp CaroController để hỗ trợ hiển thị giao diện đồ họa, sử dụng thuật toán trong lớp Main và xử lý các sự kiện chuột:

**public class** CaroController {

@FXML

**private** Label **ResultText**;

@FXML

Pane **pane**;

**private int size** = 750;

**private int spots** = 10;

**private int squareSize** = **size** / **spots**;

**private** ArrayList<Piece> **pieces**;

**private** Rectangle[][] **grid**;

**private** Main **engine**;

**boolean engineBlack** = **false**;

@FXML

**public void** initialize() {

**engine** = **new** Main();

**grid** = **new** Rectangle[**spots**][**spots**];

*// Draw chess board*

**for** (**int** i = 0; i < **size**; i += **squareSize**) {

**for** (**int** j = 0; j < **size**; j += **squareSize**) {

Rectangle r = **new** Rectangle(i, j, **squareSize**, **squareSize**);

**grid**[i / **squareSize**][j / **squareSize**] = r;

**if** (i / **squareSize** < Main.***BOARD\_SIZE*** - 1 && j / **squareSize** < Main.***BOARD\_SIZE*** - 1)

r.setFill(Color.***LIGHTPINK***);

**else**

r.setFill(Color.***WHITE***);

r.setStroke(Color.***BLACK***);

**pane**.getChildren().add(r);

}

}

*// Draw chess pieces*

**pieces** = **new** ArrayList<Piece>();

*// First move*

**engine**.**isHuman** = **true**;

**pane**.setOnMousePressed(**new** EventHandler<MouseEvent>() {

@Override

**public void** handle(MouseEvent mouseEvent) {

**int** X = (**int**) Math.*round*(mouseEvent.getX() / **squareSize**);

**int** Y = (**int**) Math.*round*(mouseEvent.getY() / **squareSize**);

System.***out***.println(**"human --------"**);

System.***out***.println(X + **", "** + Y);

Circle c = **new** Circle();

**if** (**engineBlack**) {

c.setFill(Color.***WHITE***);

c.setStroke(Color.***BLACK***);

} **else** {

c.setFill(Color.***BLACK***);

c.setStroke(Color.***WHITE***);

}

**double** radius = **squareSize** / 2.5;

**double** x = **squareSize** \* X;

**double** y = **squareSize** \* Y;

Piece p = **new** Piece(x, y, radius, c);

**pieces**.add(p);

**pane**.getChildren().add(c);

p.draw();

}

});

**pane**.setOnMouseReleased(**new** EventHandler<MouseEvent>() {

@Override

**public void** handle(MouseEvent mouseEvent) {

**int** X = (**int**) Math.*round*(mouseEvent.getX() / **squareSize**);

**int** Y = (**int**) Math.*round*(mouseEvent.getY() / **squareSize**);

**if** (**engine**.**isHuman**) {

**if** (**engine**.**board**[Y][X] == 0) {

**engine**.addHumanMove(Y, X);

*// engine.board[Y][X] = 1;*

**engine**.**isHuman** = **false**;

**if** (**engine**.checkWin() == 1) {

**ResultText**.setText(**"Human wins!"**);

**return**;

}

**if** (**engine**.checkWin() == 2) {

**ResultText**.setText(**"Tie!"**);

**return**;

}

Move aiMove = **engine**.bestMove();

System.***out***.println(**"engine --------"**);

System.***out***.println(aiMove.**y** + **", "** + aiMove.**x**);

Circle c = **new** Circle();

**if** (**engineBlack**) {

c.setFill(Color.***BLACK***);

c.setStroke(Color.***WHITE***);

} **else** {

c.setFill(Color.***WHITE***);

c.setStroke(Color.***BLACK***);

}

**double** radius = **squareSize** / 2.5;

**double** x = **squareSize** \* aiMove.**y**;

**double** y = **squareSize** \* aiMove.**x**;

Piece p = **new** Piece(x, y, radius, c);

**pieces**.add(p);

**pane**.getChildren().add(c);

p.draw();

**if** (**engine**.checkWin() == -1) {

**ResultText**.setText(**"Engine wins!"**);

**return**;

}

**if** (**engine**.checkWin() == 2) {

**ResultText**.setText(**"Tie!"**);

**return**;

}

}

}

}

});

}

@FXML

**public void** onEngineStart() {

**engineBlack** = **true**;

**engine**.**isHuman** = **false**;

Move firstMove = **engine**.bestMove();

System.***out***.println(firstMove);

Circle c = **new** Circle();

c.setFill(Color.***BLACK***);

c.setStroke(Color.***WHITE***);

**double** radius = **squareSize** / 2.5;

**double** x = **squareSize** \* firstMove.**y**;

**double** y = **squareSize** \* firstMove.**x**;

Piece p = **new** Piece(x, y, radius, c);

**pieces**.add(p);

**pane**.getChildren().add(c);

p.draw();

}

}

Xây dựng lớp CaroApplication để chạy chương trình:

**public class** CaroApplication **extends** Application {

@Override

**public void** start(Stage stage) **throws** IOException {

FXMLLoader loader = **new** FXMLLoader();

VBox root = loader.load(CaroController.**class**.getResource(**"caro.fxml"**).openStream());

**final** Button restartButton = **new** Button( **"Restart"** );

restartButton.setFont(**new** Font(30));

restartButton.setBackground(Background.*fill*(Color.***WHITE***));

restartButton.setStyle(**"-fx-border-color: blue; -fx-border-width: 3; -fx-border-radius: 10"**);

restartButton.setOnAction(actionEvent -> {

CaroApplication app = **new** CaroApplication();

**try** {

app.start(stage);

} **catch** (IOException e) {

**throw new** RuntimeException(e);

}

});

root.getChildren().add(restartButton);

Scene scene = **new** Scene(root);

stage.setScene(scene);

stage.setTitle(**"Caro game"**);

stage.show();

}

**public static void** main(String[] args) {

*launch*();

}

}

File caro.fxml để cấu hình giao diện đồ họa

*<?***xml version="1.0" encoding="UTF-8"***?>*

*<?***import javafx.scene.layout.BorderPane***?>*

*<?***import javafx.scene.layout.Pane***?>*

*<?***import javafx.scene.layout.VBox***?>*

*<?***import javafx.scene.control.Label***?>*

*<?***import javafx.scene.control.Button***?>*

*<?***import javafx.geometry.Insets***?>*

*<?***import javafx.scene.layout.GridPane***?>*

<**VBox alignment="CENTER" spacing="20.0" xmlns:fx="http://javafx.com/fxml"**

**fx:controller="com.edu.javafx\_caro.CaroController"**>

<**BorderPane**

**maxHeight="-Infinity"**

**maxWidth="-Infinity"**

**minHeight="-Infinity"**

**minWidth="-Infinity"**

**xmlns="http://javafx.com/javafx/11.0.1"**

**xmlns:fx="http://javafx.com/fxml/1"**>

<**center**>

<**Pane fx:id="pane" BorderPane.alignment="CENTER" prefWidth="750" prefHeight="750"**/>

</**center**>

</**BorderPane**>

<**padding**>

<**Insets bottom="10.0" left="40.0" right="40.0" top="40.0"**/>

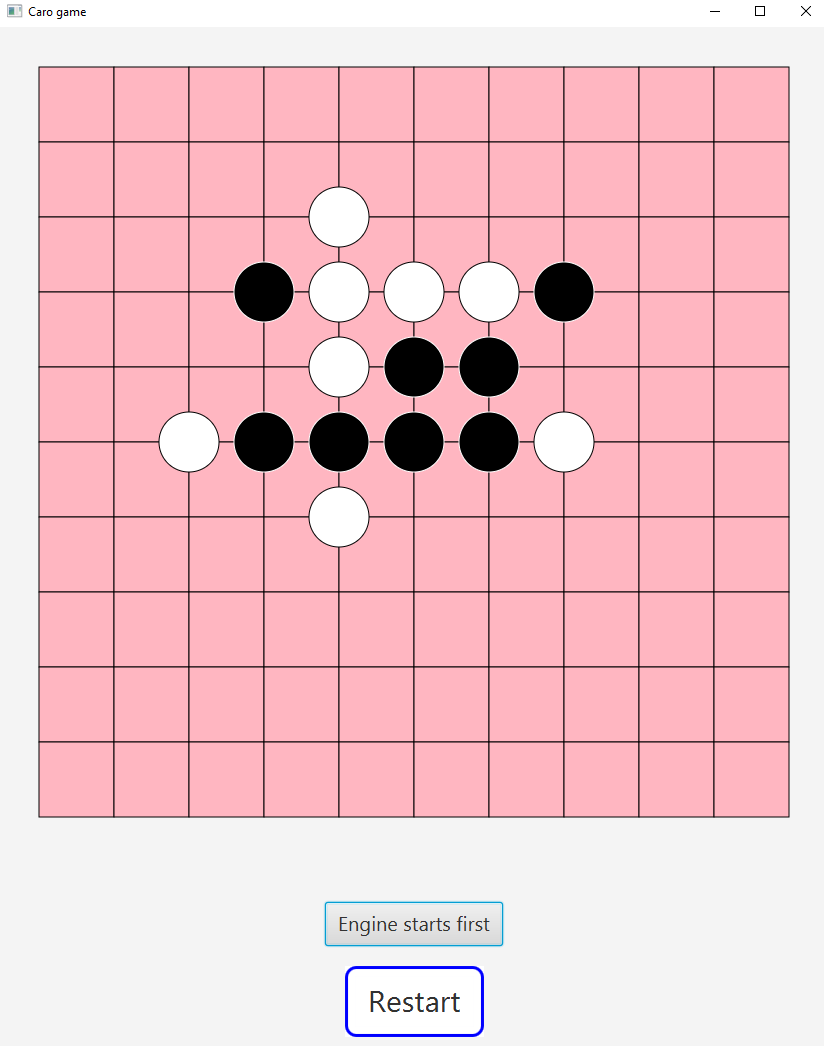
</**padding**>

<**Label fx:id="ResultText" style="-fx-font-size: 30px;"**/>

<**Button text="Engine starts first" style="-fx-font-size: 20px;" onAction="#onEngineStart"**/>

</**VBox**>

1. **Kết quả thực hiện:**



Kết quả:

