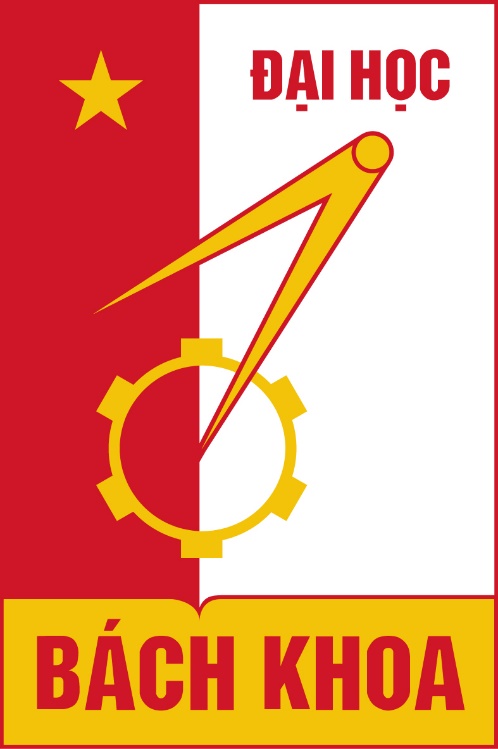
***BÁO CÁO BÀI TẬP LỚN MÔN***

Môn: Trí tuệ nhân tạo

(Đề Tài: Cờ Gánh)

*Giảng viên hướng dẫn: Nguyễn Nhật Quang*



|  |
| --- |
| Nhóm sinh viên thực hiện: Nhóm 14  Nguyễn Đức Tài MSSV: 20153264  Nguyễn Hữu Thắng MSSV: 20153517  Trần Thị Quỳnh MSSV: 20153142  Nguyễn Tiến Khương MSSV: 20152044 |

**Phụ lục**

Lời mở đầu 2

Phần 1: Giới thiệu đề tài 3

Phần 2: Ý tưởng xây dựng trò chơi 3

Phần 3: Phân tích bài toán và lựa chọn thuật toán 4

3.1 Biểu diễn bài toán dưới dạng cây trò chơi 4

3.2 Hàm đánh giá trạng thái bàn cờ 4

3.3 Thuật toán Minimax 4

3.4 Thuật toán Alpha-Beta Prunning 7

3.4 Stock Alpha-Beta 10

Phần 4: Nhận xét 10

Phần 5: Chú ý cài đặt 11

TÀI LIỆU THAM KHẢO 12

**Lời mở đầu**

Hiện nay, chúng ta nghe thấy rất nhiều về trí tuệ nhân tạo, việc triển khai triển khai các ứng dụng và xây dựng các trò chơi đồng thới áp dụng các kĩ thuật AI là một việc gần như không thể thiếu. Nhất là trong lĩnh vực phát triển game, Cờ gánh không phải là ngoại lệ. Hơn thế nữa, đây là một trò chơi dân gian có nguồn gốc ở Quảng Nam Việt Nam. Chính vì thế mà chúng em đã quyết định chọn Cờ gánh làm đề tài cho bài tập lớn của mình. Đây là tài liệu mô tả một cách cơ bản về việc xây dựng game. Trong game có sử dụng thuật toán Minimax và AlphaBeta đồng thời đề ra các cải tiến mới cho thuật toán Alpha-Beta pruning. Trong quá trình thực hiện không thể tránh khỏi những sai sót, chúng em rất mong nhận được những đóng góp của thầy giáo.

**Phần 1: Giới thiệu đề tài (Cờ Gánh).**

Trò chơi thuộc thể loại chiến lược, chiến thuật.

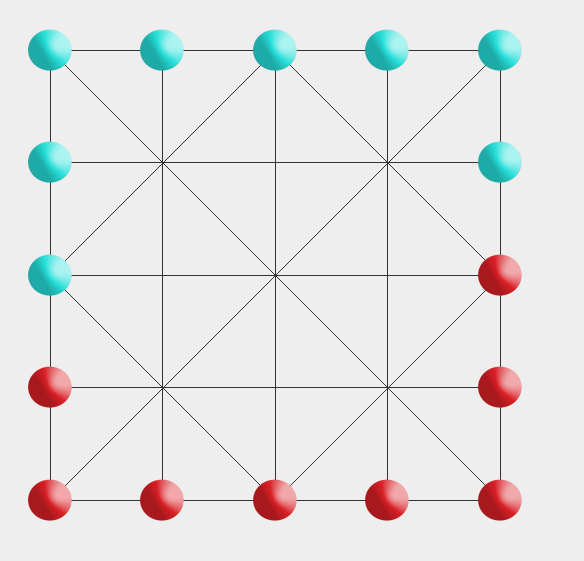
Số lượng người chơi: 2 người chơi.

Tập luật:

Mỗi người chơi có trong tay, tại xuất phát điểm, 8 quân cờ.

Quân cờ của người chơi này có đặc điểm nhận dạng khác với quân cờ của người chơi kia.

Tất cả các quân cờ của một người chơi là giống nhau và có cách di chuyển trên bàn cờ và tuân theo luật chơi giống nhau.



Bàn cờ là một bề mặt phẳng có 25 điểm nằm tại giao điểm của một lưới vuông 4 nhân 4 như trên hình vẽ bên phải.

Các đường kẻ nằm ngang, thẳng đứng hoặc đường chéo, được vẽ trên bàn cờ như hình vẽ bên phải, thể hiện các đường di chuyển được phép của các quân cờ.

Tại xuất phát điểm, các quân cờ của hai người chơi được bố trí ở các điểm nằm rìa ngoài của bàn cờ như trên hình vẽ bên phải.

Khi chơi, lần lượt mỗi người được di chuyển bất kỳ một quân cờ nào của mình đến một giao điểm lân cận của lưới vuông, theo đường nằm ngang, đường thẳng đứng hoặc đường chéo thể hiện ở trên bàn cờ, mà chưa có quân cờ nào ở đó. Nếu quân cờ của một người chơi nằm ở giữa (trung điểm) của hai quân cờ đối phương lân cận, thì hai quân cờ lân cận này của đối phương bị coi là bị "gánh" và bị đổi màu để trở thành quân cờ của người chơi. Nếu quân cờ của người chơi nằm xung quanh một quân cờ của đối phương, khiến cho nó không thể di chuyển, thì quân cờ này của đối phương bị coi là bị "vây" và bị đổi màu để trở thành quân cờ của người chơi.

Trò chơi kết thúc khi một người chơi không còn nước đi khả thi khi tới lượt mình.

**Phần 2: Ý tưởng xây dựng trò chơi.**

**Trò chơi được xây dựng dựa trên các mô hình cờ cơ bản được nhắc tới trong trò chơi cờ vua Black-Widow**

Bàn cờ được xây dựng dưới dạng một list 25 các ô cờ được đánh số từ 0-24.

Bàn cờ sẽ lưu lại trạng thái các quân cờ, các nước đi hợp lệ của mỗi phe, trạng thái của người chơi hiện tại đang điều khiển nước đi.

Một con bot được xây dựng với tác tử thông minh và thêm vào bàn cờ với khả năng theo dõi bàn cờ và trả lại nước đi khi tới lượt phe cờ được đánh dấu là bot. Ở trường hợp này mặc định bot là phe cờ màu xanh.

**Phần 3: Phân tích bài toán và lựa chọn thuật toán**

1. *Biểu diễn bài toán dưới dạng cây trò chơi(Game tree)*

Trò chơi có thể được biểu diễn dưới dạng 1 cây gồm gốc, các nút , các nhánh

-Gốc là trạng thái ban đầu của trò chơi, hay nói cách khác là trạng thái hiện tại của bàn cờ.

-Các nút (node)của cây thì có nút cha (trạng thái sinh ra trạng thái hiện tại) nút con ( trạng thái được sinh ra từ trạng thái hiện tại ,

-Nút lá là trạng thái cuối cùng của cây có thể biểu diễn trạng thái khi một trong hai phe dành được chiến thắng, hay là trạng thái cuối cùng của bàn cờ sau độ sâu đã cho trước của cây.

-Độ sâu của cây ở đây thể hiện độ sâu mà ta có thể tiên đoán được trước các nước đi tiếp theo

1. *Hàm đánh giá trạng thái bàn cờ:*

**public** **static** **boolean** isEndGame(Board board) {

**return** board.currentPlayer().getLegalMoves().isEmpty();

}

* Kiểm tra xem người chơi thắng hay thua.

**public** **static** **int** evaluate(Board board, Alliance alliance) {

**if** (*isEndGame*(board) && alliance == board.currentPlayer().getAlliance()) {

**return** -2000;

}

**if** (*isEndGame*(board) && alliance != board.currentPlayer().getAlliance()) {

**return** 2000;

}

**return** alliance.isBlue()

? (board.getBluePieces().size() - board.getRedPieces().size())

: (board.getRedPieces().size() - board.getBluePieces().size());

}

* Đưa ra đánh giá trạng thái bàn cờ đối với phe quân cờ đang xét. Kết quả trả về là chênh lệch giữa số lượng hai quân cờ của 2 người chơi. Nếu phe quân cờ đang xét bị thua điểm số trả về sẽ là -2000 và ngược lại nếu thắng sẽ trả về điểm số là 2000.

1. *Thuật toán MiniMax:*

Hai người chơi trong game được đại diện là MAX và MIN. MAX đại diện cho người chơi luôn muốn chiến thắng và cố gắng tối ưu hóa ưu thế của mình còn MIN đại diện cho người chơi cố gắng cho người MAX giành số điểm càng thấp càng tốt.

Giải thuật Minimax thể hiện bằng cách định trị các Node trên cây trò chơi:

Node thuộc lớp MAX thì gán cho nó giá trị lớn nhất của con Node đó.

Node thuộc lớp MIN thì gán cho nó giá trị nhỏ nhất của con Node đó.

Từ các giá trị này người chơi sẽ lựa chọn cho mình nước đi tiếp theo hợp lý nhất

* Đánh giá: Khi tăng độ sâu của thuật toán tăng lên tức là khi ta muốn độ thông minh của máy thì số lượng các node mà chúng ta phải xét sẽ là rất lớn cụ thê là nếu số nhánh của cây là a, depth là b thì ta cần phải duyệt a^b node.

**public** Move execute(**final** Board board) {

**final** **long** startTime = System.*currentTimeMillis*();

**int** highestSeenValue = Integer.***MIN\_VALUE***;

**int** lowestSeenValue = Integer.***MAX\_VALUE***;

**int** currentValue;

**int** moveCounter = 1;

**final** **int** numMoves = board.currentPlayer().getLegalMoves().size();

Move bestMove = Move.MoveFactory.*getNullMove*();

System.***out***.println(board.currentPlayer() + " THINKING with depth = " + **this**.searchDepth);

System.***out***.println("\tmoves! : " + board.currentPlayer().getLegalMoves());

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**final** **long** candidateMoveStartTime = System.*nanoTime*();

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

currentValue = board.currentPlayer().getAlliance().isBlue()

? min(moveTransition.getToBoard(), **this**.searchDepth - 1)

: max(moveTransition.getToBoard(), **this**.searchDepth - 1);

**if** (board.currentPlayer().getAlliance().isBlue()) {

**if** (currentValue > highestSeenValue) {

highestSeenValue = currentValue;

bestMove = move;

} **else** **if** (currentValue == highestSeenValue) {

Random random = **new** Random();

**if** ((random.nextBoolean() && move.getScore() == bestMove.getScore())|| move.getScore() > bestMove.getScore()) {

bestMove = move;

}

}

} **else** **if** (board.currentPlayer().getAlliance().isRed()) {

**if** (currentValue < lowestSeenValue) {

lowestSeenValue = currentValue;

bestMove = move;

} **else** **if** (currentValue == lowestSeenValue) {

Random random = **new** Random();

**if** ((random.nextBoolean() && move.getScore() == bestMove.getScore())|| move.getScore() < bestMove.getScore()) {

bestMove = move;

}

}

}

System.***out***.println("\t" + toString() + " analyzing move (" + moveCounter + "/" + numMoves + "):\t" + move+ "\tbest:" + bestMove + "\tscores " + currentValue + "\tt: "

+ calculateTimeTaken(candidateMoveStartTime, System.*nanoTime*()));

moveCounter++;

}

**this**.executionTime = System.*currentTimeMillis*() - startTime;

System.***out***.printf("%s SELECTS %s [ time taken = %d ms] [buttons: %d]\n", board.currentPlayer().toString(),bestMove, **this**.executionTime, count);

**return** bestMove;

}

**private** **int** min(**final** Board board, **final** **int** depth) {

**if** (depth == 0 || BoardUtils.*isEndGame*(board)) {

**this**.count++;

**return** BoardUtils.*evaluate*(board, Alliance.***BLUE***);

}

**int** lowestSeenValue = Integer.***MAX\_VALUE***;

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

**final** **int** currentValue = max(moveTransition.getToBoard(), depth - 1);

**this**.count++;

**if** (currentValue <= lowestSeenValue) {

lowestSeenValue = currentValue;

}

}

**return** lowestSeenValue;

}

**private** **int** max(**final** Board board, **final** **int** depth) {

**if** (depth == 0 || BoardUtils.*isEndGame*(board)) {

**this**.count++;

**return** BoardUtils.*evaluate*(board, Alliance.***BLUE***);

}

**int** highestSeenValue = Integer.***MIN\_VALUE***;

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

**final** **int** currentValue = min(moveTransition.getToBoard(), depth - 1);

**this**.count++;

**if** (currentValue >= highestSeenValue) {

highestSeenValue = currentValue;

}

}

**return** highestSeenValue;

}

1. *Thuật toán Alpha-Beta pruning*

Trong chiến lược MiniMax ta đã thấy ở trên thì số lượng node xét trong trường hợp tệ nhất sẽ là rất lớn Thuật toán cắt tỉa Alpha-Beta đưa ra giúp giảm số lượng node cần phải xét.

Thuật toán làm giảm số lượng node cần duyệt để làm giảm không gian duyệt Giải thuật thêm vào hai tham số Alpha và Beta. Chúng cho biết các điểm có giá trị (hàm evaluate) nằm ngoài khoảng [alpha,beta] là các điểm không cần phải xét nữa. Ta bắt đầu tại node gốc( trạng thái hiện tại) với các giá trị của alpha là –infinity và beta là +infinity. Sau đó gọi đệ quy chính nó khi ấy khoảng [alpha, beta] sẽ giảm dần.

* Giải thuật được implement dưới đây:

**public** Move execute(**final** Board board) {

**final** **long** startTime = System.*currentTimeMillis*();

Move bestMove = MoveFactory.*getNullMove*();

**int** highestSeenValue = Integer.***MIN\_VALUE***;

**int** lowestSeenValue = Integer.***MAX\_VALUE***;

**int** currentValue;

**int** moveCounter = 1;

**final** **int** numMoves = board.currentPlayer().getLegalMoves().size();

System.***out***.println(board.currentPlayer() + " THINKING with depth = " + **this**.searchDepth);

System.***out***.println("\tmoves! : " + board.currentPlayer().getLegalMoves());

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

**final** String s;

**final** **long** candidateMoveStartTime = System.*nanoTime*();

currentValue = board.currentPlayer().getAlliance().isBlue()

? (min(moveTransition.getToBoard(), **this**.searchDepth - 1, highestSeenValue, lowestSeenValue))

: (max(moveTransition.getToBoard(), **this**.searchDepth - 1, highestSeenValue, lowestSeenValue));

**if** (board.currentPlayer().getAlliance().isBlue()) {

**if** (currentValue > highestSeenValue) {

highestSeenValue = currentValue;

bestMove = move;

} **else** **if** (currentValue == highestSeenValue) {

Random random = **new** Random();

**if** ((random.nextBoolean() && move.getScore() == bestMove.getScore())|| move.getScore() > bestMove.getScore()) {

bestMove = move;

}

}

} **else** **if** (board.currentPlayer().getAlliance().isRed()) {

**if** (currentValue < lowestSeenValue) {

lowestSeenValue = currentValue;

bestMove = move;

} **else** **if** (currentValue == lowestSeenValue) {

Random random = **new** Random();

**if** ((random.nextBoolean() && move.getScore() == bestMove.getScore())|| move.getScore() < bestMove.getScore()){

bestMove = move;

}

}

}

**final** String quiescenceInfo = " [h: " + highestSeenValue + " l: " + lowestSeenValue + "]";

s = "\t" + toString() + "(" + **this**.searchDepth + "), m: (" + moveCounter + "/" + numMoves + ") " + move+ ", best: " + bestMove+ quiescenceInfo + ", t: " + calculateTimeTaken(candidateMoveStartTime, System.*nanoTime*());

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

~~setChanged~~();

~~notifyObservers~~(s);

moveCounter++;

}

**this**.executionTime = System.*currentTimeMillis*() - startTime;

System.***out***.printf( "%s SELECTS %s [time taken = %d ms], [buttons = %d]",

board.currentPlayer(), bestMove, **this**.executionTime, **this**.count);

**return** bestMove;

}

**private** **int** max(**final** Board board, **final** **int** depth, **final** **int** highest, **final** **int** lowest) {

**if** (depth == 0 || BoardUtils.*isEndGame*(board)) {

**this**.count++;

**return** BoardUtils.*evaluate*(board, Alliance.***BLUE***);

}

**int** currentHighest = highest;

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**this**.count++;

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

currentHighest = Math.*max*(currentHighest, min(moveTransition.getToBoard(), depth - 1, currentHighest, lowest));

**if** (lowest <= currentHighest)

**break**;

}

**return** currentHighest;

}

**private** **int** min(**final** Board board, **final** **int** depth, **final** **int** highest, **final** **int** lowest) {

**if** (depth == 0 || BoardUtils.*isEndGame*(board)) {

**this**.count++;

**return** BoardUtils.*evaluate*(board, Alliance.***BLUE***);

}

**int** currentLowest = lowest;

**for** (**final** Move move : board.currentPlayer().getLegalMoves()) {

**this**.count++;

**final** MoveTransition moveTransition =

board.currentPlayer().makeMove(move);

currentLowest = Math.*min*(currentLowest,

max(moveTransition.getToBoard(), depth - 1, highest, currentLowest));

**if** (currentLowest <= highest)

**break**;

}

**return** currentLowest;

}

* Đánh giá thuật toán :Trong điều kiện lí tưởng người ta tính toán được thì thuật toán alpha beta cắt tỉa chỉ phải xét số nút theo công thức
* 2\*a^(b/2)-1 nếu b chẵn
* a^((b+1)/2) nếu b lẻ

trong đó a là số nhánh trên cây , b là độ sâu của cây

các node cần phải duyệt đã giảm đi đáng kể nhưng mà chúng ta cần phải tìm ra các phương pháp cải tiến để giảm đi số lượng các node cần duyệt hơn nữa.

Chúng em đã đề ra phương án tiếp theo.

1. *Cải tiến giải thuật (StockAlphaBeta)*

Như đã đề cập ở trên, chúng em đã đưa ra phương án để giảm số lượng các node cần duyệt hơn nữa chúng em đưa ra hàm getScore() cho mỗi trạng thái. Tại mỗi node ta sẽ có thông số là Score được tính bằng số quân ăn được . Những nước cờ sẽ được sắp xếp theo Score từ cao đến thấp.

**private** Collection<Move> smartSortDown(Collection<Move> moves) {

Comparator<Move> sortByScoreDown = **new** Comparator<Move>() {

@Override

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

**return** o2.getScore() - o1.getScore();

}

};

**return** Ordering.*from*(sortByScoreDown).immutableSortedCopy(moves);

}

**Phần 4 Nhận xét**  
Chúng em đã implement thành công 3 thuật toán trên. Đồng thời đưa ra một sample để chỉ ra

performance về mặt thời gian cũng như số lượng các node cần phải duyệt của cả 3 giải thuật.

\*Hướng phát triển

Sau khi máy đưa ra trạng thái kế tiếp của bàn cờ tức là thực hiện một nước đi, máy sẽ lưu lại Path chứa tất cả các node đã duyệt trước đó để tìm ra nước đi vừa đánh. Khi ấy nếu đến phiên của người đánh. Người đánh đi nước tốt nhất của họ và nằm trong Path thì thay vì phải một lần nữa áp dụng lại giải thuật, máy chỉ cần đưa ra trạng thái trong path.

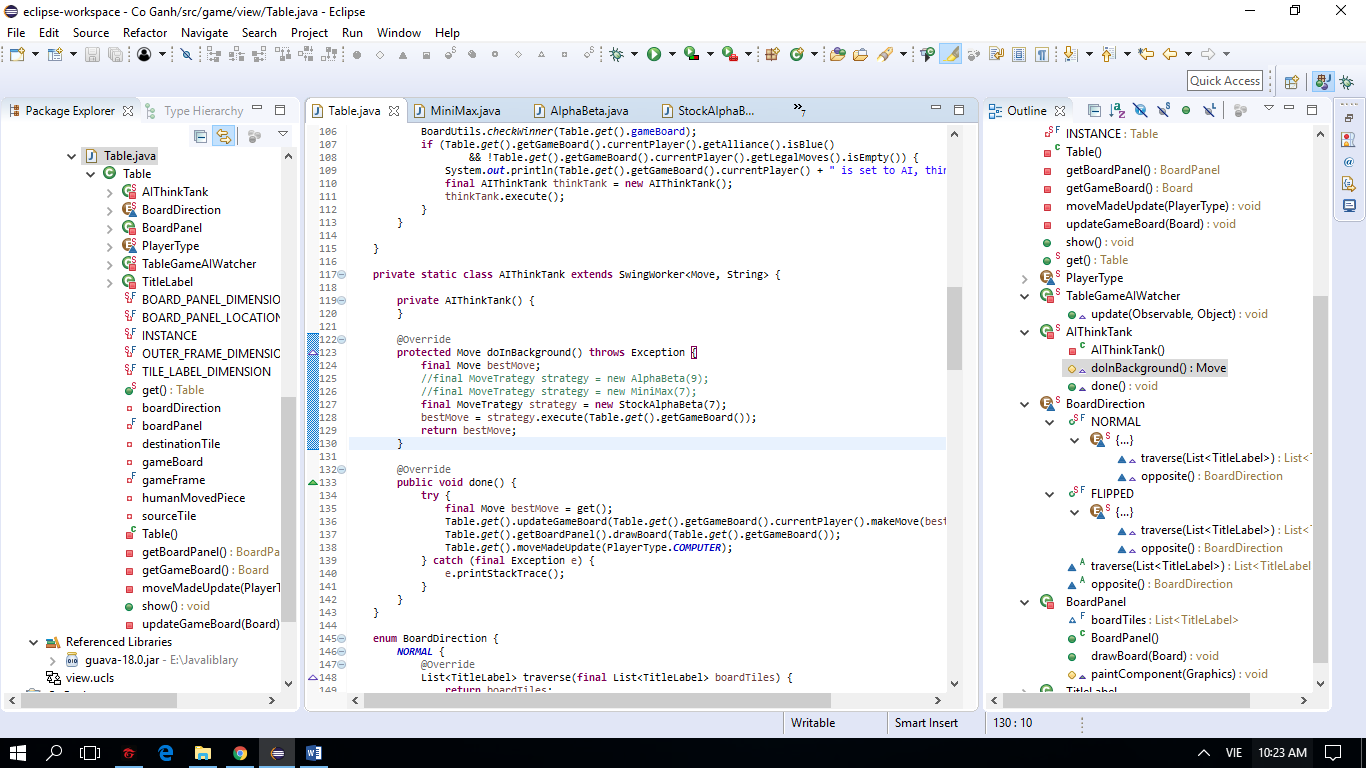
**Phần 5 Chú ý khi cài đặt.**

Chú ý khi cài đặt: Cần import đầy đủ thư viện để chạy

Môi trường phát triển : IDE: Eclipse 4; Platform JDK 9.01.

Thư viện: Jre9.01, guava 18.0;

Thư viện guava được cung cấp bởi google có thể truy cập vào trang gitthub với cụm từ guava 18.0. Thư viện này cũng được lưu đính kèm cùng với trò chơi.

Thay đổi thuật toán và độ sâu ở file Table tại class AIThinkTank

**TÀI LIỆU THAM KHẢO**

Dựa trên mô hình cờ cơ bản về các trò chơi cờ nói chung được tham khảo trò chơi cờ vua BlackWidow Chess Master:

https://www.youtube.com/playlist?list=PLOJzCFLZdG4zk5d-1\_ah2B4kqZSeIlWtt