Nine Men's Morris Project Design

by Yuxiang Feng(32431813), Yifan Wang(32459238), Tianyi Liu(27936619)

1.Team Information

1.1 Team Name and Photo

- Team Name: **G**enerative **A**dversarial **N**etwork (short: GAN)
- Team Photo

1.2 Team Membership

Name	Contact Detail	Tech Stack	Fun Fact
Yifan(Charles) Wang	ID: 32459238 Email: ywan0687@student.monash.edu	python,java,MEAN Stack, IOS app dev, Android app dev	I like to play tachibana mahjong and joined the Monash Mahjong Club in order to be able to play tachibana mahjong offline. I like to play single player games, especially JRPG. I am addicted to the persona series,

Name	Contact Detail	Tech Stack	Fun Fact
			dragon quest series, octopath series and final fantasy series. Also love soul games, platinum achievement Elden Ring, is also waiting for Bloodborne to be played on PC. Is managing a thousands of people's idol support club. 🎮
Tianyi Liu	ID: 27936619 Email: tliu135@student.monash.edu	Android dev, MEAN stack, python, git	I acquired Platinum Trophies of Elden Ring before the beginning of S1 2023. I am a dog lover who has a 4-year-old golden retriever who needs to be walked every single day. I often swim as exercise and to keep myself healthy. I have not

Name	Contact Detail	Tech Stack	Fun Fact
			learnt java systematically before this semester and I am currently taking both FIT2099 and FIT3077. 🏷️
Yuxiang Feng	ID: 32431813 Email: yfen0056@student.monash.edu	iOS dev, MEAN Stack and Spring full stack	I am an avid cyclist. I once spent 7 hours cycling around my hometown, with a total mileage of 50 kilometers. Incidentally, the average temperature that day was 38 degrees Celsius. I'm a game "collector" and I buy a lot of video games, but most of them I don't have time to play.💥

1.3 Team Schedule

1.3.1 Weekly Meeting

Team meetings are to be held twice every week, one taking place on campus at 10:00 a.m. Wednesdays and the other online at 10:00 a.m. Saturdays.

1.3.2 Workload Distribution

Since we are adopting the agile approach, no tasks are delegated to a specific team member instead we are picking tasks based on our tech skills and everyone is mandatory for contributing to every single task at hand to a degree. A detailed wiki is documented in our group git-lab repository.

1.4 Team Tech Stack and Justification

1.4.1 Tech Stack

Members of our team have rich experience in Java development, for example, Yifan and Tianyi have experience in Android development. Yuxiang has experience in Java Web development. At the same time, we are all students who love object-oriented technologies such as Java.

Therefore, our group plans to adopt a Java-based technology stack. For example, the Java language itself and the IDEA integrated development environment.

In terms of visualization technology, we initially intend to use Terminal as the main output interface.

During development, we intend to use Maven as our project management tool. As a very large package management platform, Maven can easily manage project dependencies. We can add JUnit or Spring dependencies directly in Maven, and any dependencies we may add. If there are new requirements or external dependencies in the subsequent agile development, we can integrate these dependencies into the project through Maven at any time.

We plan to use JUnit as our testing tool. JUnit provides a very simple and fast unit testing method that allows us to quickly test the functions we have completed. Guarantee rapid development, rapid testing and rapid iteration of the entire project.

We plan to introduce Spring dependencies in Sprint3. This is because Spring provides an excellent IoC container that can help us further reduce the coupling between classes. IoC is a well known and common design pattern. Although the IoC design pattern is not one of GoF23, it has been adopted by many well-known companies and open source frameworks in the world so far. We want to improve the overall scalability of the project.

We plan to introduce MVC framework in Sprint3 or Sprint4. The games we design require the player to interact with the game, and the game requires visual output. Therefore, we believe that the MVC framework can further reduce the coupling between the view layer and the control layer. When our requirements expand, we only need to modify the control layer or view layer related to this requirement. The scalability of the project will be further expanded and the maintenance cost will be reduced.

To sum up, we set 3 different stage goals for our group's delivery. In the first stage, basic functions and Terminal-based GUI implementation are completed. The second stage is to introduce the Spring IoC container to further reduce the coupling between classes. In the third stage, the MVC framework is built to further reduce the overall coupling degree of the project and reduce maintenance costs. Separation of functions and responsibilities through the MVC framework improves the scalability of the project.

We are very much looking forward to the teacher's ability to teach us the GUI and MVC framework in the course. If he can lead us to implement a simple IoC framework, it will be of great help to us. In this way, we can use our own simple IoC container to complete the management of the class in the earlier delivery.

1.4.2 Tech Justification

The reasons for choosing Maven, Spring and MVC frameworks have been explained above and will not be repeated here. Here is an explanation of our team's choice of visualization form for the project's minimum viable product. Early in our group discussion, we listed three options, Terminal, JavaFx, and Web. These three technologies have their own advantages.

The advantage of Terminal is that what you see is what you get, we can just manage our visual screens, such as chessboards and chess pieces, in the early stage of development. This form of development is the lowest cost, because we hardly need any additional learning to complete the development. But the disadvantage is also obvious, that is, terminal-based development is very primitive. We cannot perform effective visualization, and this form is very strongly coupled with game logic. We have to embed this visual part in our function code. This means that once important functions are changed in later iterations, we need to refactor the terminal part, and the maintenance cost is very high.

The advantage of JavaFx is that it is a GUI development framework mainly promoted by Oracle in this era as a Java GUI development framework, and it is still being updated with more modern documents and development tools. The downside is that we need extra study. We cannot predict the cost of such learning, which may greatly delay our development progress, or even seriously miss the delivery date.

The advantage of the Web is that it can easily complete the separation of the front and back ends and the MVC framework. Web technology is the hottest technology at the moment, and we have a lot of learning resources to find. And Web technology itself is for more intuitive development of front-end pages, so it is very friendly to our game visualization development. Its disadvantage is similar to that of JavaFx, that is, we cannot predict whether its learning cost is acceptable to us.

To sum up, our initial choice is to use Terminal as our MVP (minimum viable product) visualization technology. But this does not mean that we abandoned JavaFx or Web. On the contrary, every member of our group is a student who loves technology and has the courage to innovate and challenge. We have planned to try to learn these two techniques in our spare time, and try to make a game board demo if possible. If all goes well, it is not ruled out that we will use one of these two technologies in MVP.

2.User Stories

NOTE: As the game could result in a draw "Nine men's morris is a solved game, that is, a game whose optimal strategy has been calculated. It has been shown that with perfect play from both players, the game results in a draw." (Gasser, 1996). However, as such functionality requires the implementation of one of the advanced requirements (2.b), and our chosen implementation of the advanced requirement is (2.c), so this is not discussed in our project. [1]

- As a player
    1. I want to be able to see my available moves of a token

       So I can minimise the chance of missing a potential move.
    2. I want to see hints when I have attempted an illegal move

       So I know the reason for I cannot move to the position
    3. I want to play against real people

       So that I can enjoy the sheer happiness of victory or the bitter sorrow of losing
    4. I want to play against AI

       So that I can practice with AI until I feel comfortable playing against real people
    5. I want to see which player's turn it is in the game UI

       So that I know whose turn it is to act
    6. I want to remove one of the opponent's tokens when I form a mill

       So that I can win the game eventually

7. I want to place my tokens freely as long as the position is not taken by another token before playing out my nine tokens

   So that I can strategize when moving the tokens

8. I want to move my tokens to the adjacent position after all my tokens are placed as well as my remaining tokens on board is more than three

   So that I can attempt to create a mill and gain an advantage in the game

9. I want to move my tokens freely when I only have three tokens left

   So that I can have a better chance of winning

10. I want to auto-end my turn when I am done

    So that I can play the game more smoothly

- As a board

  1. I want to have the game board clearly distinguishing the "nodes" and "lines"

     So that the players can effortlessly recognise the position to place tokens

  2. I want to check a given player's number of remaining tokens

     So that I ensure the player's move pattern complies with the rule

  3. I want to check a given player's possible moves when the player's token is greater than three

     So that if there are no possible moves for the current player, I can declare the victory of the other player

  4. I want to ensure in a single turn there is a maximum of one token moved from the board even if a player achieved a double mill

     So that the game meets the standard rule

  5. I want to ensure a player can only place/move a single token in a single turn

     So that the game's rule is followed

  6. I want to end the game when a player's remaining token is less than three

So that I can declare the other player’s victory

7. I want to know which player has won the game

   So that I can acknowledge that player’s victory

8. I want to ensure players’ turns are alternating when ended

   So that each player can take turns playing

9. I want to ensure that a player cannot place or move a token in a position if that position is already occupied by another token

   So that the game’s rule is followed

10. I want to ensure that a player cannot take a token within a mill

    So the game’s rule is followed

- As a developer

  1. I want to add a button to toggle nine men's morris’s rules

     So that the players are clear of the game’s rules

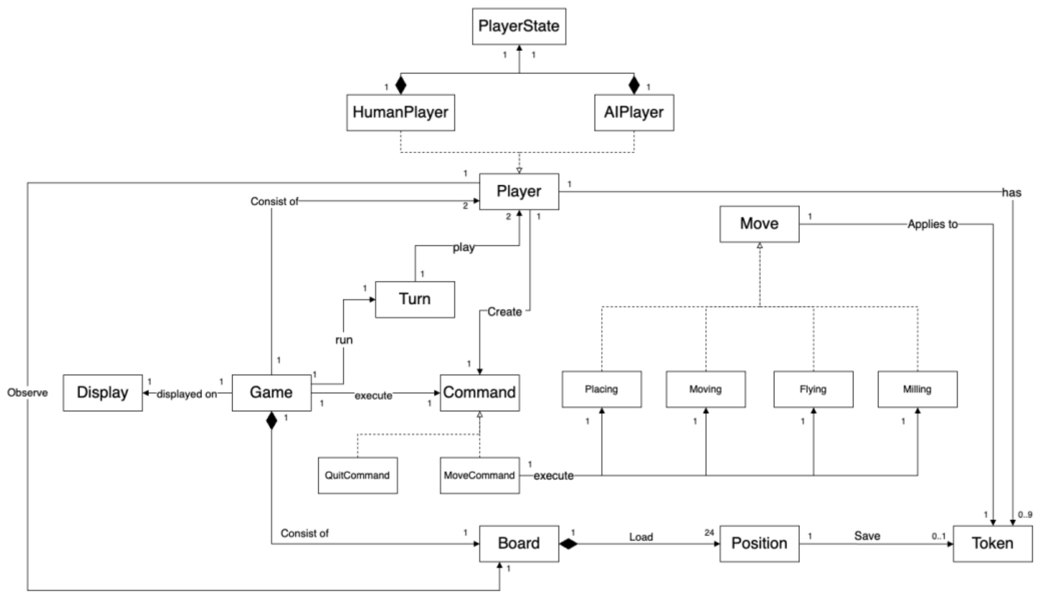
- As a game (assumption)

  1. I want to ensure when both players’ tokens are in a mill, the active player’s turn is ended without taking a token from the other player. (e.g. play A moved a token which formed a mill. However, player B’s tokens are all in a mill on the board. So Player A will not be able to remove a token from player B.)

     So the game’s rule is followed

3.Basic Architecture

3.1 Domain Model



NOTE: We choose advanced feature is 'c: A single player may play against the computer, where the computer will randomly play a move among all of the currently valid moves for the computer, or any other set of heuristics of your choice.'

3.2 Design Rationale

3.2.1 Introduction

This program is a Nine Men's Morris game running locally. As a board game, we need to display its board and pieces, and provide interactive interfaces for players. Due to the high-level feature our group chose was the AI player. Therefore, we will increase the battle mode between human players and AI players.

Next, we will introduce our design in two parts.

The first part is to explain the domain model above. Explain that the entities in the picture have those relationships.

The second part is Design Principle. In this section, we will elaborate on which principles we follow? How do we follow it? Why do we follow these principles?

The third part is Design Pattern. In this part, we will elaborate on which design patterns we plan to implement? How did we achieve them? Also illustrate the design with some pictures with more classes. And why we choose such a design patter.

3.2.2 Entity Relationship Description

3.2.2.1 Game

Game entity is the entrance of the entire game. It is responsible for controlling the overall flow of the game, such as building the game, starting the game, and ending the game.

Game needs to rely on Display for output.

The relationship between Game and Board is Composition, because without a game, the board is meaningless.

Game contains Player and Turn. But Game needs to operate Player through Turn. The specific reasons will be explained in the subsequent sections.

3.2.2.2 Display

Display is the output interface of the entire game. The content that needs to be output in the entire game needs to be output through this unified interface to maintain the formatting of the overall GUI.

3.2.2.3 Board

Board and Position are in the relationship of Composition. Because without the Board, the Position loses its meaning. A Board will load 24 Positions. The Board is observed by the Player.

3.2.2.4 Position

Position will hold Token. But the number of Tokens is limited. Therefore, Position may or may not have a Token.

3.2.2.5 Token

Tokens will be placed on the board. At the beginning of the game, each player holds 9 Tokens. Players will place the Tokens in their hands on the board in turn. During the game, the player's piece may be "milling" by another player, at which point the player will lose control of the Token.

3.2.2.6 Player

Player is controlled by Turn. When the Turn informs the Player that the player needs to play the game, the Player can perform the current round of game operations. Player needs to observe the Board and feed back the observed Board situation to PlayerState. Player can create Command.

3.2.2.7 PlayerState

PlayerState is an entity that controls player state. It will share the Board situation fed back by the player, and decide whether the Player's state needs to be changed according to the rules of the game. PlayerState must be held by Player, and a Player corresponds to a PlayerState.

3.2.2.8 Command

Command is created by the player, and the Game is responsible for executing the Command. There are two generic types of Command, one is QuitCommand and the other is MoveCommand. (There is actually an UndoCommand, but the Undo feature is not part of the advanced effects our group chose, so decided not to show it, just for illustration)
MoveCommand will contain the concrete generic type of Move.

3.2.2.9 Move

Move has four generic types, Placing, Moving, Flying and Milling. Move wraps the algorithm for the specific operation that the Token will perform. So a Move will correspond to a Token.

3.2.2.10 Turn

Turn is used to control the round in the game. Turn stores the player. Whenever the Game runs a Turn, the Turn will notify the player to play with the game. After a player has played a turn, the Turn changes who will be playing the next time.

3.2.3 Design Principles (SOLID)

3.2.3.1 Single-Responsibility Principle

After discussion and the relationship displayed by the domain model, it can be found that every entity in this project will follow the SRP.

For example, Display is only responsible for visual output and does not perform any specific logical operations. When GUI output is required in the game, Display will output the received content.

Game is only responsible for the control of the overall stage of the game, not the specific details. When after the Game builds the game board, pieces, players and Turn. Then Game will start. During the game, Game does not pay attention to the specific changes of the game. Only when the game player's state changes (to Failed), the Game's state will change, and it will change from playing to ending the game.

Turn is only responsible for controlling the turn of the game, and does not care how the player plays the game.

Player creates Command, but is not responsible for executing Command. The Player only pass Board to the PlayerState, and does not pay attention to the specific situation of the Board. Only PlayerState is responsible for the change of the player state, and only when the state changes, the player will change.

3.2.3.2 Open-Closed Principle

OCP is a principle that is very important to the overall architecture of the framework. Every team hopes that the system they design is highly scalable. And be robust enough. After we design interfaces or abstractions, these interfaces or abstractions should not be modified in subsequent development and expansion.

In our domain model, Player, Command, and Move are good examples.

Such as Player. Whether it is Human or AI, they all implement the Player interface. The Player interface defines a set of common methods, such as the play() function. The play() function needs to return Command so that Game can execute it. If we need to expand the Player in the future, such as AI players with different strategic styles. We don't need to modify the Player interface.

It is also a good example of following the OCP principle for Command and Move. For Command, if the Undo function needs to be added in the future, we only need UndoCommand to implement the Command interface. For Move, if we need to add extra type of Move, we only need to implement the Move interface.

3.2.3.3 Liskov Substitution Principle

The LSP principle means that the subclasses we extend should all be able to replace their superclasses. This is a principle that makes our program very elegant, because between different classes. It only needs to pay attention to those functions that the parent class of the other party has. In this way, the details between each class are blurred, and the scalability of the whole system is enhanced. For example, Player, Command and Move in the domain model.

For example, for Turn, it only needs to call the Player's play() function. As for whether the Player is Human or AI, this is something Turn does not need to care about. Because whether it is Human or AI, they all have the same play() function.

Another example is Command and Move. MoveCommand does not need to know which type of Move it specifically stores. Whether it is Placing or Moving, MoveCommand does not need to care. When MoveCommand needs to execute Move, it only needs to call the common execute() function of Move. Because no matter what type of Move, this function is implemented.

3.2.3.4 Interface Segregation Principle

The ISP principle states that we should not force a class to implement methods it should not implement. In specific practice, we usually split a complex and huge interface into multiple small interfaces with more specific functions. In our design, Player will implement 3 interfaces. They are `PlayerInterface`, `PollableInterface` and `ObserverInterface` respectively. These three interfaces represent three different groups of functions respectively.

There is a `play` function in PlayerInterface, which means that the class that implements this interface can play games.

There is a `poll` function in PollableInterface, which means that the class that implements this interface can be controlled by Turn.

There is a `notification` function in ObserverInterface, which means that the class that implements this interface can be notified by the class that implements ObservableInterface.
At this point, the Player completes the ISP. Because the Player completes the combination of its own expected functions by using several interfaces with different functions.

3.2.3.5 Dependency Inversion Principle

DIP principles will be used to the greatest extent in our projects. An amazing fact is that all entities in the domain model (except generics) will be interfaces or abstract classes. Using this method to build the entire framework has greatly improved the overall scalability of the framework. This is because each entity in the entire project may need to be expanded in the future. When we need to extend them, if they fail to meet the DIP principle, this will lead us to modify the source code, and further, the OCP principle is broken.

We declare functions and attributes that we think may persist for the entire project in Interface or abstract classes. And they will complete the interaction by calling functions in other Interfaces or abstract classes. At this point, the details between each class will be hidden, and they improve the overall scalability through dependency inverse.

The idea that helped us accomplish this was that instead of focusing on Nine Men's Morris when designing the overall framework, we focused on one category, 2D board games. We pay attention to the common features of these 2D board games, and design these common functions as Interface or abstract classes. When we need to implement the Nine Men’s Morris game in the next delivery, we only need to implement these Interfaces or abstract classes, and expand new functions to complete the development of the Nine Men’s Morris game.

3.2.4 Design Pattern

3.2.4.1 Dependency Injection

Although Dependency Injection is not a member of GoF23, it is still a very commonly used design pattern. In theory, other classes that a class needs to use should not be created by itself, but should come from external injection. The advantage of this is to separate the two operations of class creation and use. Reduced coupling between classes.

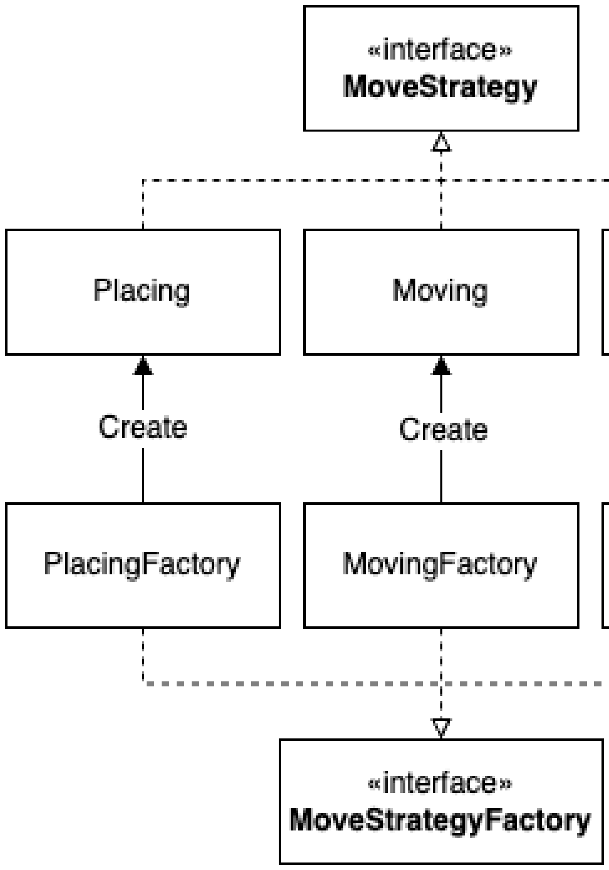
In our design, Game has the highest degree of coupling with other classes. Therefore, our Game will not create other classes during initialization, but will receive injection from external functions. (This is another reason why we expect to use IoC in Tech justification)

```
public GameFacade(Board board, Player player1, Player player2, Turn tur
n, Display display) {
        this.board = board;
        this.player1 = player1;
        this.player2 = player2;
        this.turn = turn;
        this.display = display;
}
```

3.2.4.2 Factory Method

Factory Method is a very common and efficient design pattern for creating instances. The advantage of Factory Method is that it reduces the coupling of the program. We delegate the creation of instances to a separate class. Another benefit is compliance with SRP and OCP. Since we've offloaded the creation functionality to a separate class, our maintenance costs are low. And when we need to add a new "product", we don't need to modify the source code, just create a new factory.

In our design, Move uses a factory method to create its own subclasses. More detailed blueprints are below.



3.2.4.3 Composite

The composition design pattern allows some small classes to be combined into a larger class. Use them like a class by encapsulating them.

In our domain model, Board and Position complete the combined design pattern. Position exposes a suitable interface to the Board so that it can be called by the Board. When the game needs to operate Position or Token through the Board, it can directly issue commands to the Board. The function corresponding to the interface exposed based on Position has been implemented in the Board. These functions are combined into complete functionality when we call Board. Therefore, the Game can operate the Position as if it were operating the Board.

3.2.4.4 Facade

The advantage of the appearance design pattern is that it can make the program independent of complex subsystems and provide a simple and intuitive operation interface.

In our domain model, Game implements the appearance design pattern. Game only provides `build`, `start`, `stop` and other interfaces. It gives users a simple and direct way to operate the game. The caller does not need to worry about how each subsystem works, and can complete a series of complex operations by calling these interfaces provided by Game.

3.2.4.5 Command

The command pattern is a design pattern that converts all the information about the requested content into an object. The advantage of this design is that not only can the requested information be parameterized, but the request itself can also be recorded. If we put the request into the stack, we can implement the undo operation.

In our domain model, Command implements Command Pattern. Player will create different Commands, and some Commands will pass Move as a parameter. When the Game executes the Command, the Move as a parameter will be executed.

Another advantage of this is the separation of command initiation and execution. Player creates commands but does not execute commands. Game is responsible for executing the command.

3.2.4.6 Observer

The Observer pattern creates a subscription mechanism. When the observed object changes, it can notify all objects that have subscribed to it. The advantage of this is that we establish a communication between objects, and this communication can occur synchronously with the operation. At the same time, this design also satisfies the principle of opening and closing, and we can introduce new subscriber classes without modifying the source code.

In our domain model, Player and Board complete the Observer pattern. When the pieces on the Board increase or decrease, the Board will notify the Player. Player will take actions based on specific information.

```
public interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
}
public interface Observer {
    void notification(Observable observable);
}
```

3.2.4.7 State

The State Pattern makes changes to an object's internal behavior the same as changes to its own properties. The advantage of this is that we put the specific status code in a separate class, and put this class as a separate property in another class. Another class only needs to ask the status class to know its own status. This greatly reduces the conditional code for state judgments.

In our domain model, PlayerState implements the state pattern, and each Player will save a PlayerState as one of its own properties. And take different actions according to the state given by PlayerState.

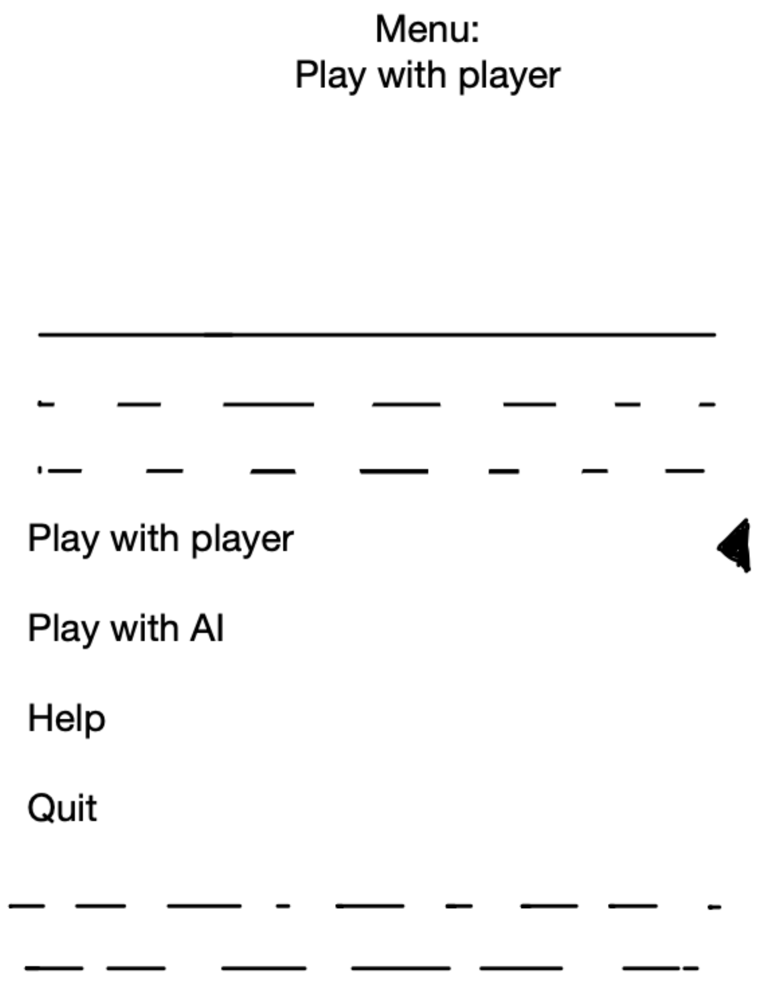
3.2.4.8 Strategy

The Strategy pattern allows us to define a set of algorithms and wrap the algorithms into self-contained classes. Multiple classes form a set of policies, which can be switched between each other.

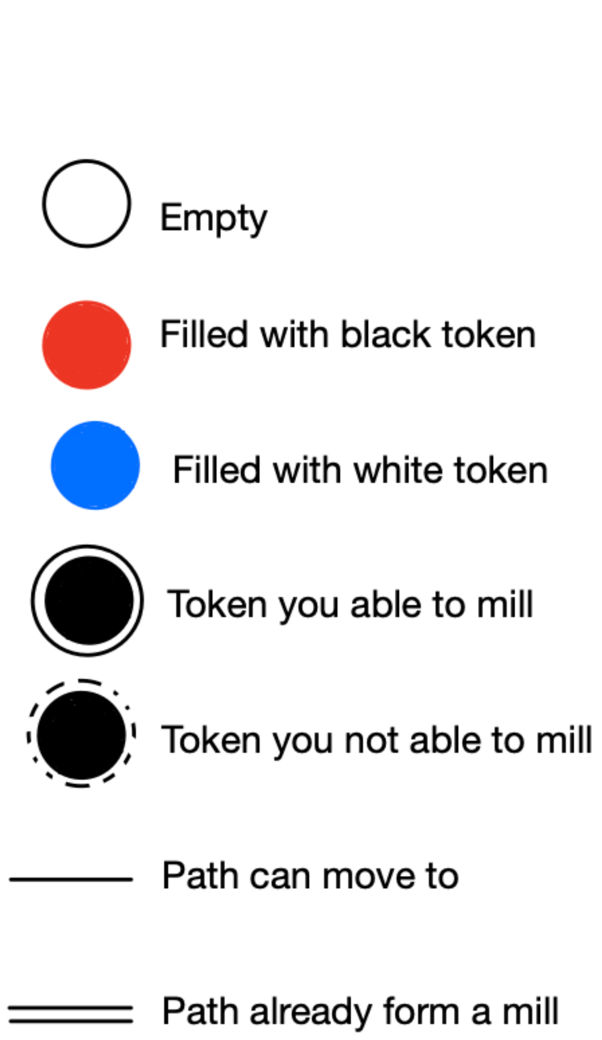
In our domain model, we use the strategy pattern on Move. Although Player only initiates one MoveCommand, we need to give MoveCommand different Move strategies according to the player's status.

4.Basic UI Design

4.1 Menu

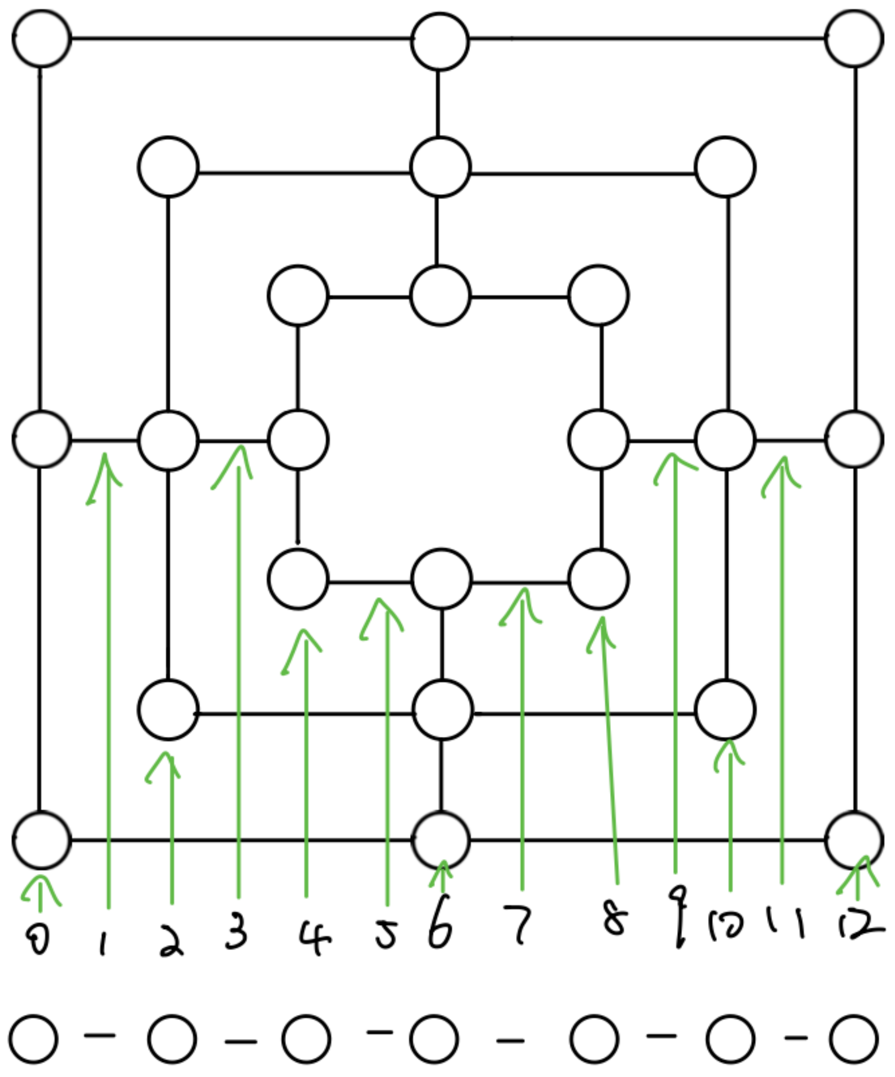


4.2 Icons

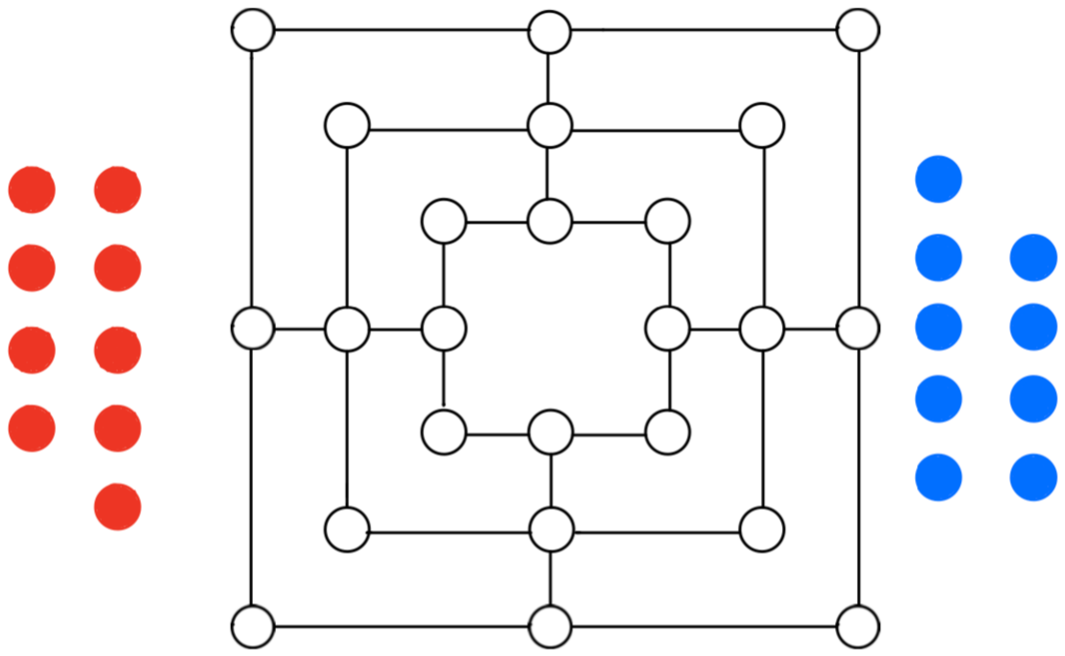

Icons
Empty
Filled with black token
Filled with white token
Token you able to mill
Token you not able to mill
Path can move to
Path already form a mill

4.3 Board Introduction

Stage one:
The whole board can be seen as a 13 *13 coordinate system, when the user input the coordinates, the token will be placed on that coordinate, error message show when there is already a token



Side choice



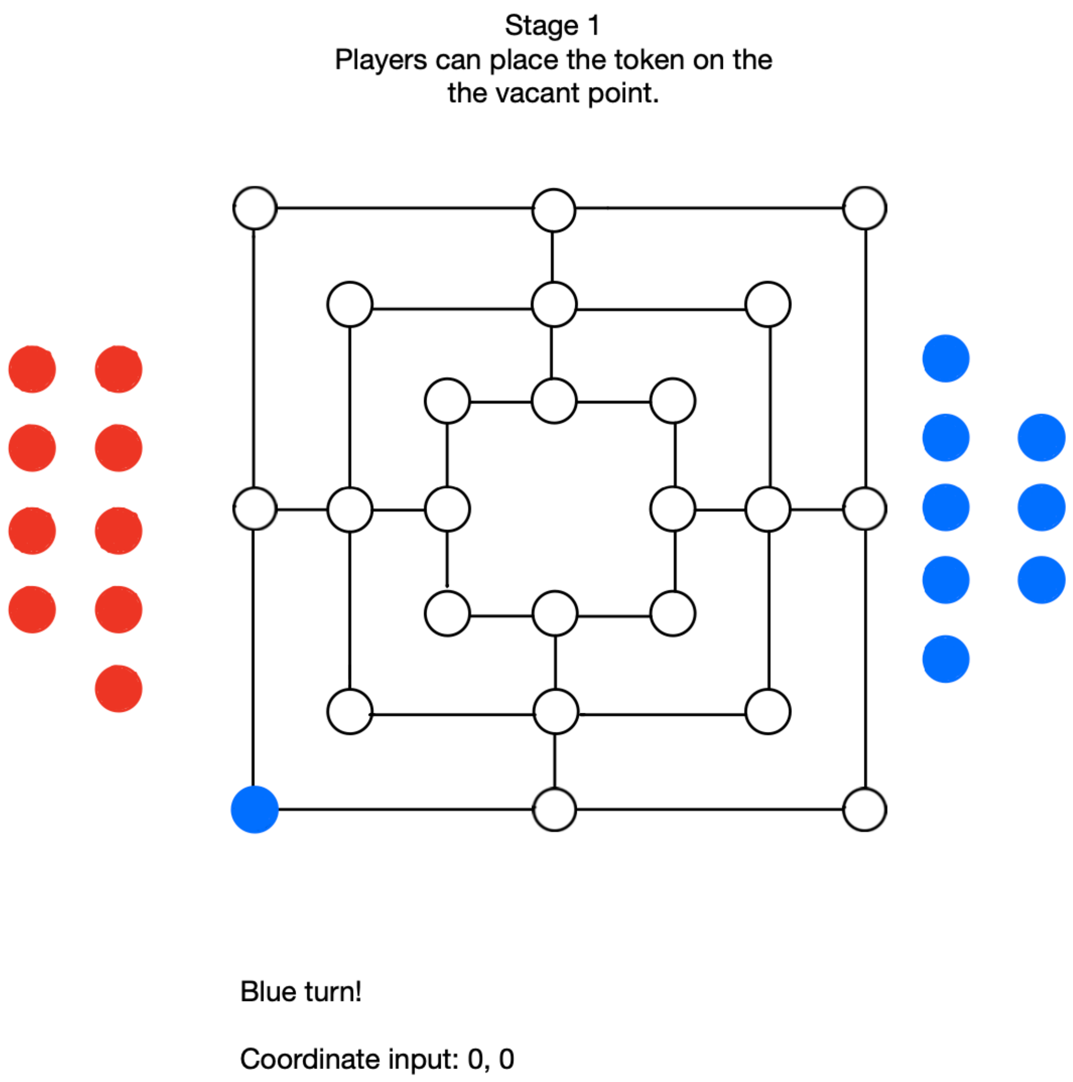
Player 1 choice:
A. Head
B. Tail

or

Head!
Player1 choice your side:
A. Blue(first)
B. Red(second)

Tail!
Player2 choice your side:
A. Blue(first)
B. Red(second)

4.4 Stage One

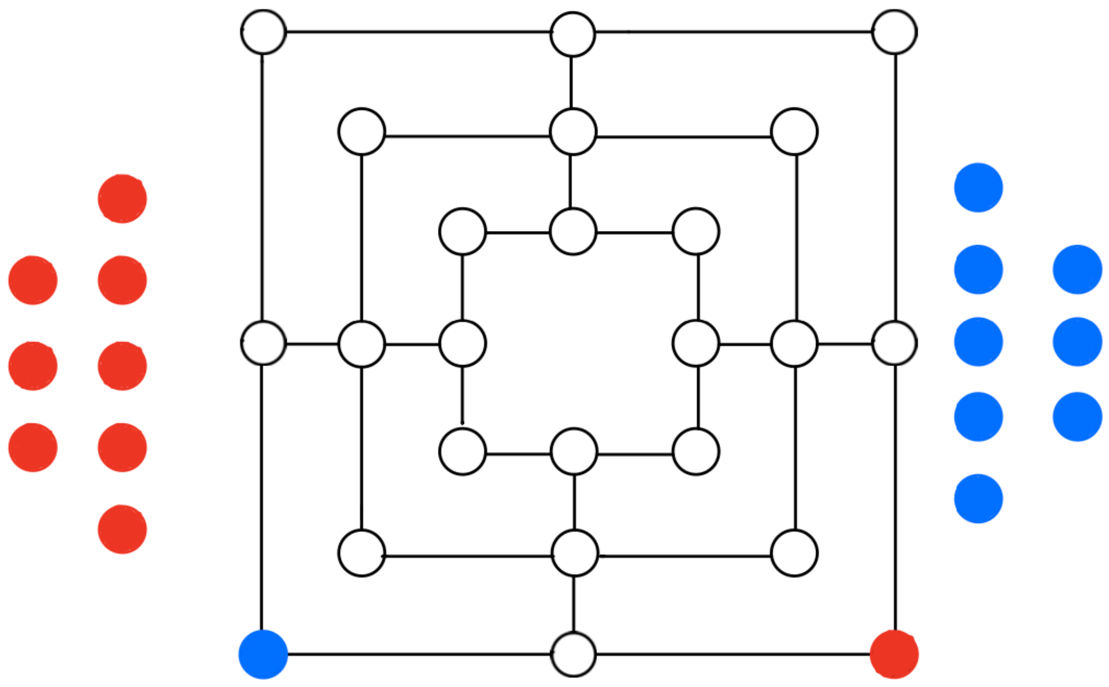


Stage 1
Players can place the token on the
the vacant point.

Blue turn!

Coordinate input: 0, 0

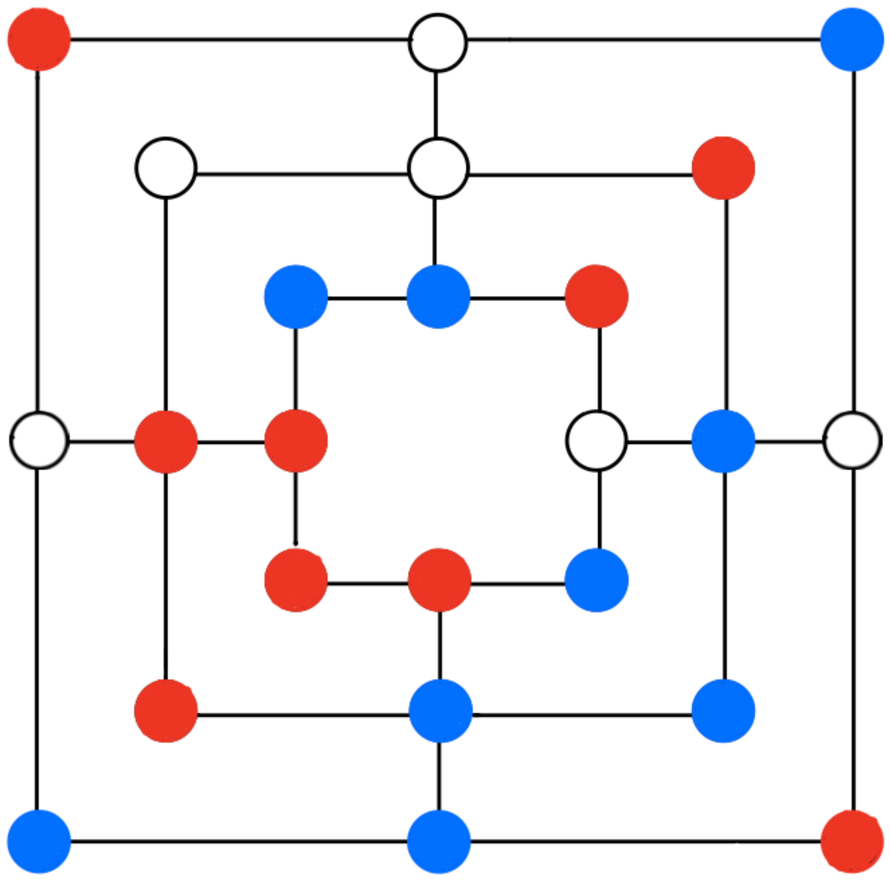
Stage 1
The other user do the same thing in this stage



Red turn!

Coordinate input: 12, 0

Stage 1
End of stage one



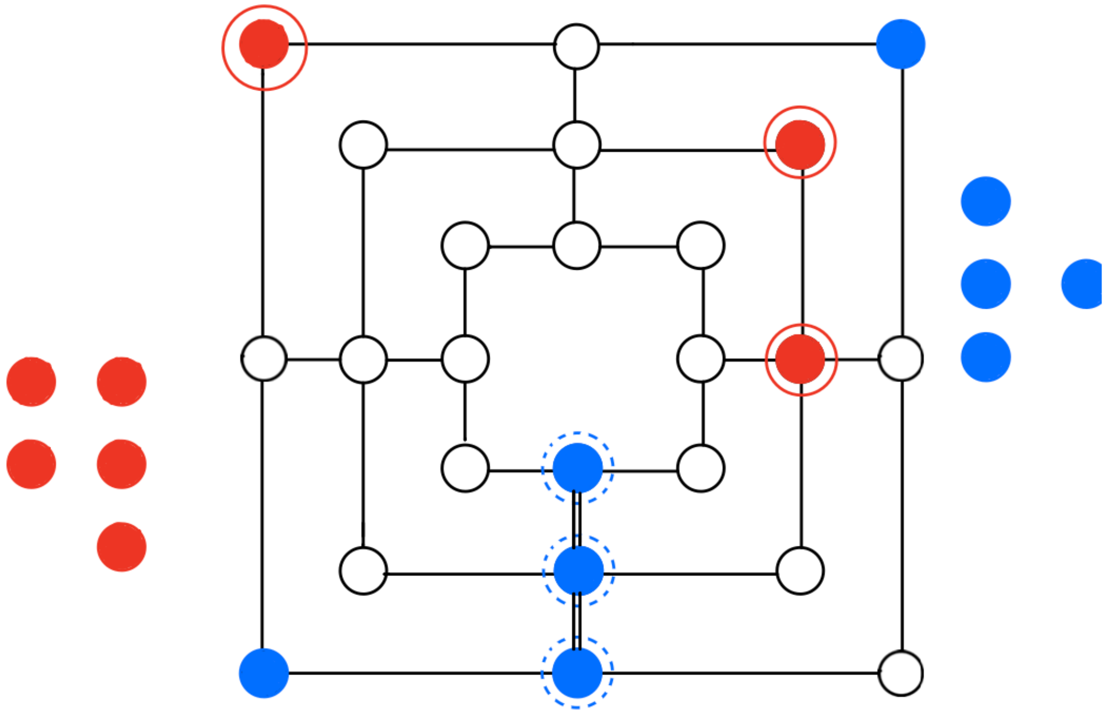
blue turn:
1. 0 ,0
2. 12, 12
3. 6, 0
4. 6, 2
5. 10, 2
6. 6, 8
7. 8, 4
8. 4, 8
9. 10, 6

Red turn:
1. 12, 0
2. 0, 12
3. 10, 10
4. 6, 4
5. 2, 2
6. 4, 4
7. 4, 6
8. 8, 8
9. 2, 6

At this points, the first stage where no token have been captured, comes to end

Stage 1: mill situation
When a mill form, will highlight the opponent token you able to taken

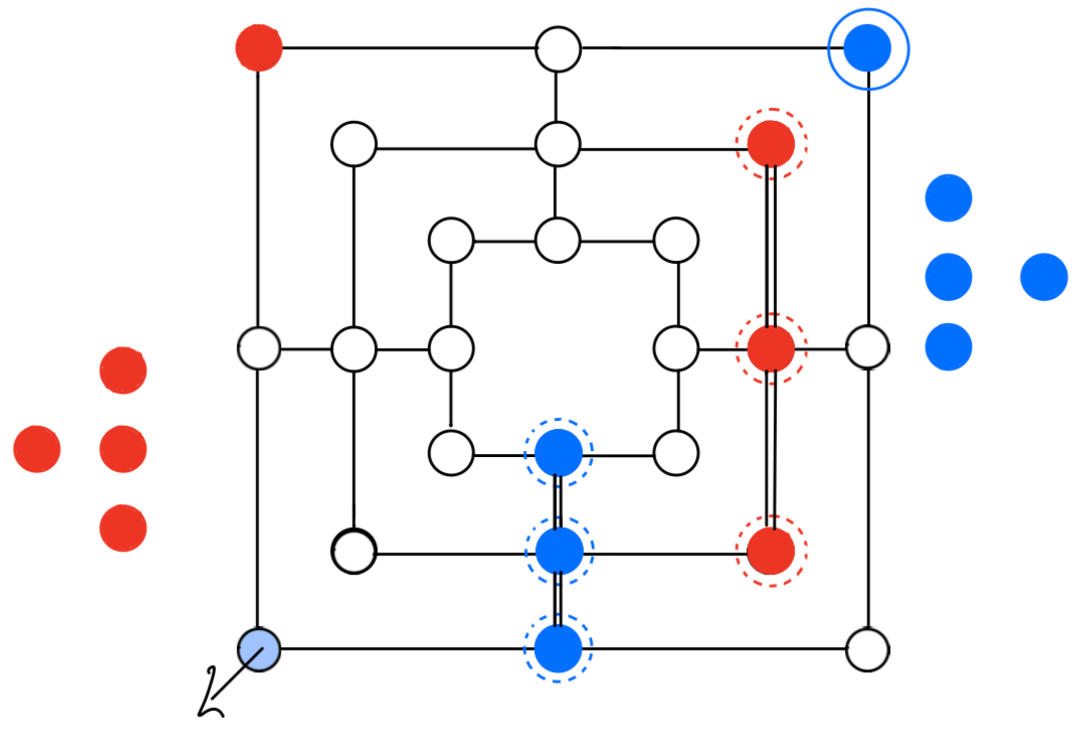
Three token stay together will form a mill, you can take a token from the board.



Blue turn!

Chose a token you want to remove: 12, 0

Stage 1: one mill for each player
You cannot take the token in the mill

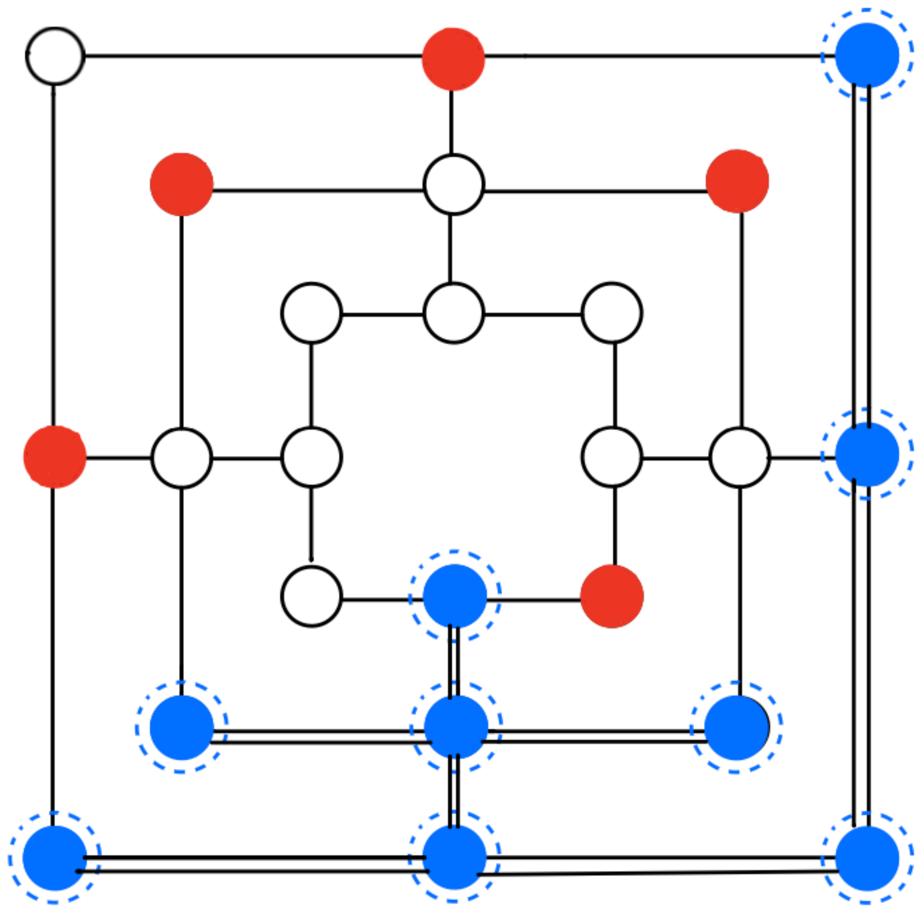


Red turn!

Chose a token you want to remove: 0, 0

4.4.1 Extreme case 1

Extreme case 1:
In stage 1(placing stage) one side can have 4 mills most.
And the other side will left 5 token least

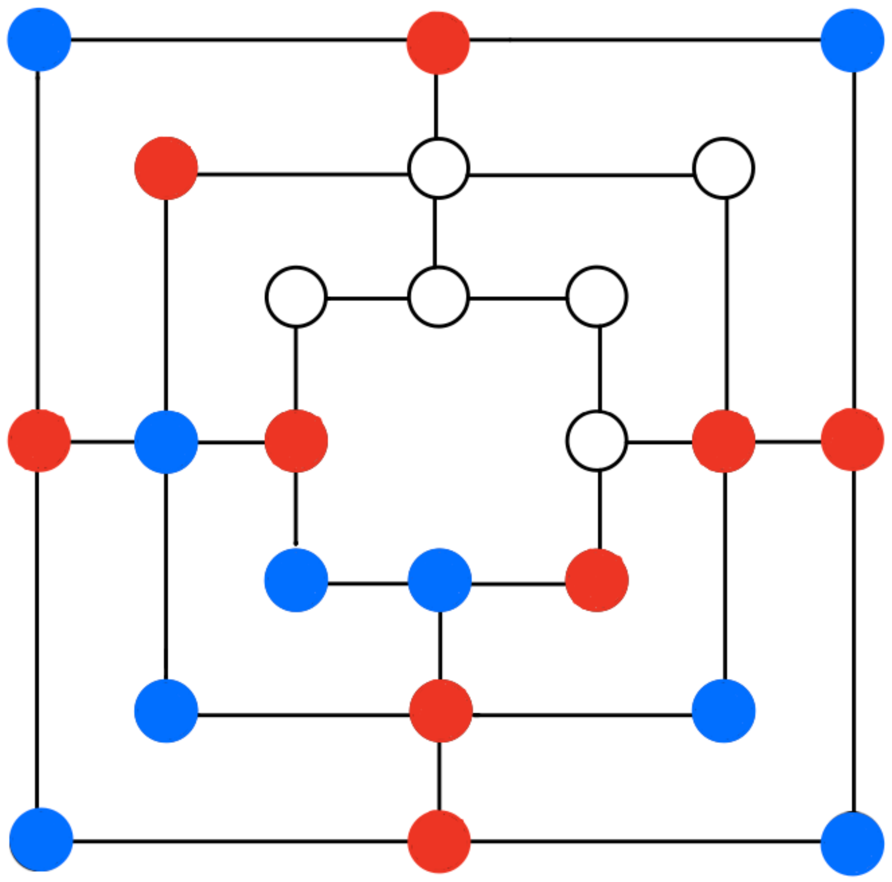


Red turn!

Coordinate input: 10,6

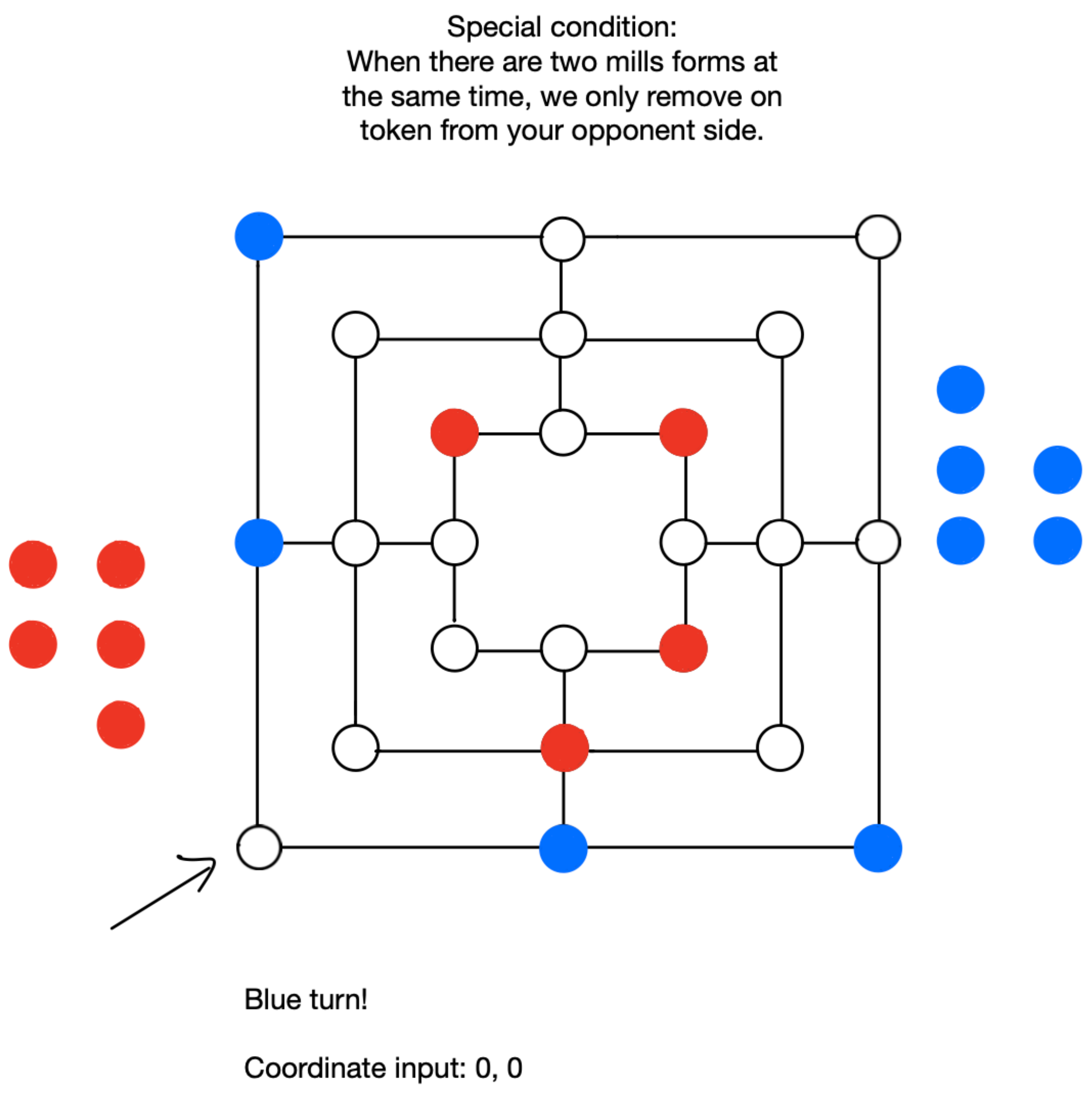
4.4.2 Extreme case 2

Extreme case 2:
All the blue token be blocked, red win the game straight away.



Red Win!

4.4.3 Special Condition

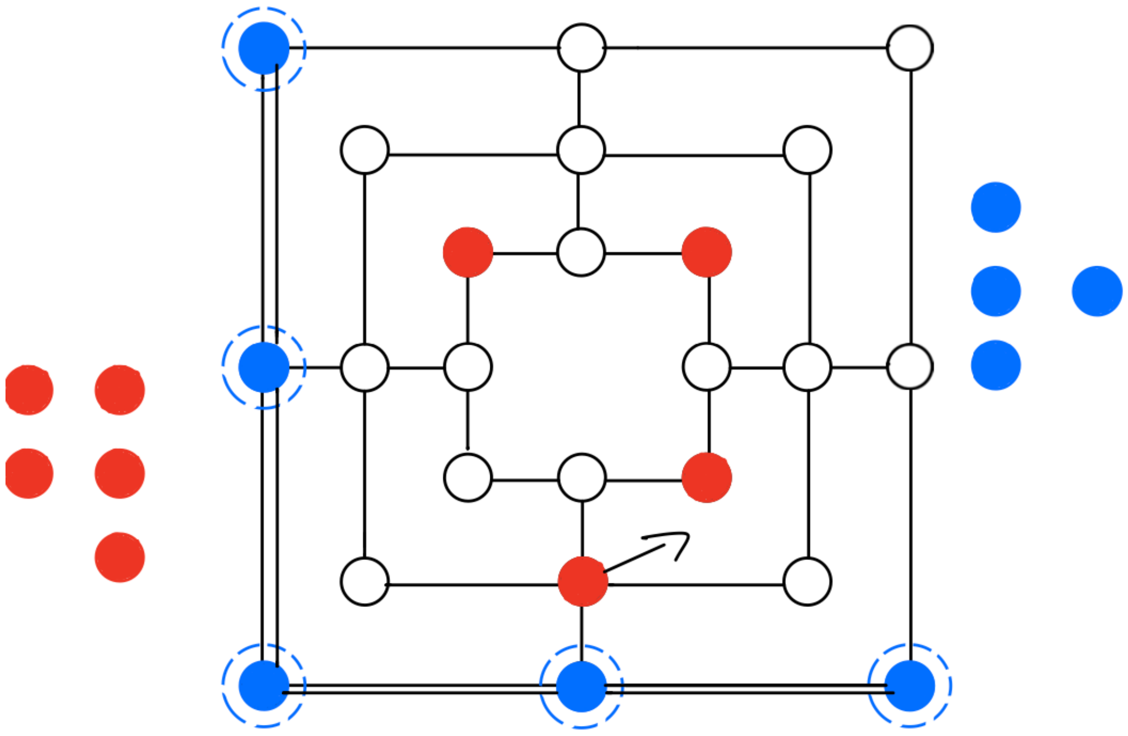


Special condition:
When there are two mills forms at the same time, we only remove on token from your opponent side.

Blue turn!

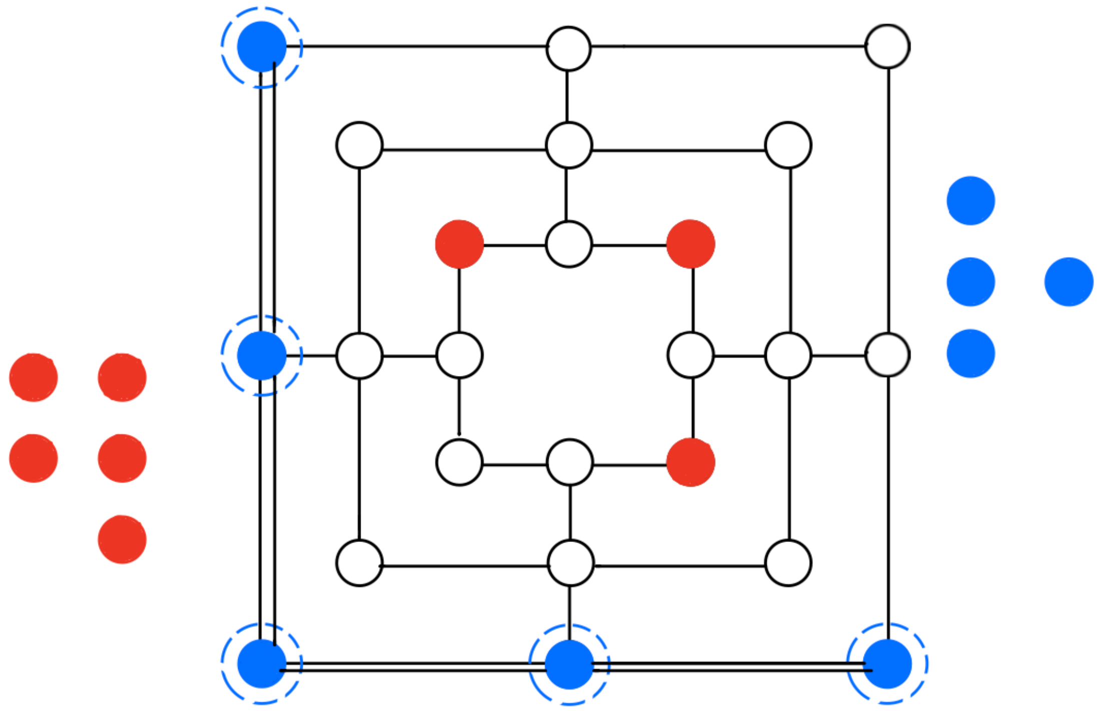
Coordinate input: 0, 0

Special condition:
When there are two mills forms at the same time, we only remove on token from your opponent side.



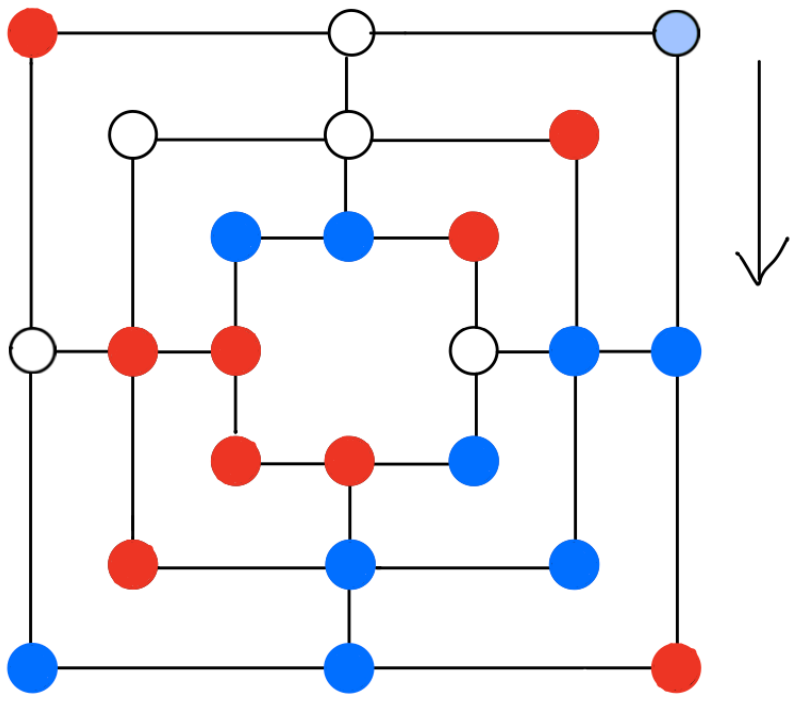
Blue turn!

Choose a token you want to remove: 6, 2



4.5 Stage Two

Stage 2:
This is the moving stage, players can move the token to the nearest vacant point.



Red turn!
Choose one token from you:

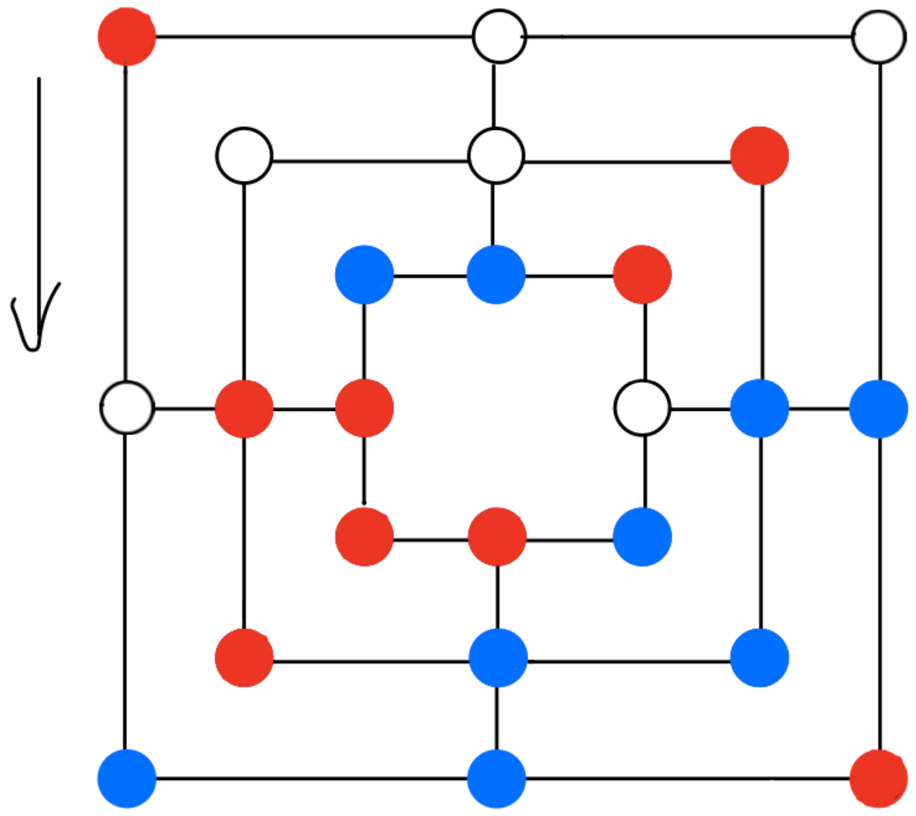
A. 12, 0
B. 0, 12
C. 10, 10
D. 6, 4
E. 2, 2
F. 4, 4
G. 4, 6
H. 8, 8
I. 2, 6

$\xrightarrow{B}$

Red turn!
You want to move to:

A. 0, 6 $\longrightarrow$ B
B. 6, 12

Stage 2:
After one player move the token, the other player will move his token.



Red turn!
Choose one token from you:

A. 12, 0
B. 0, 12
C. 10, 10
D. 6, 4
E. 2, 2
F. 4, 4
G. 4, 6
H. 8, 8
I. 2, 6

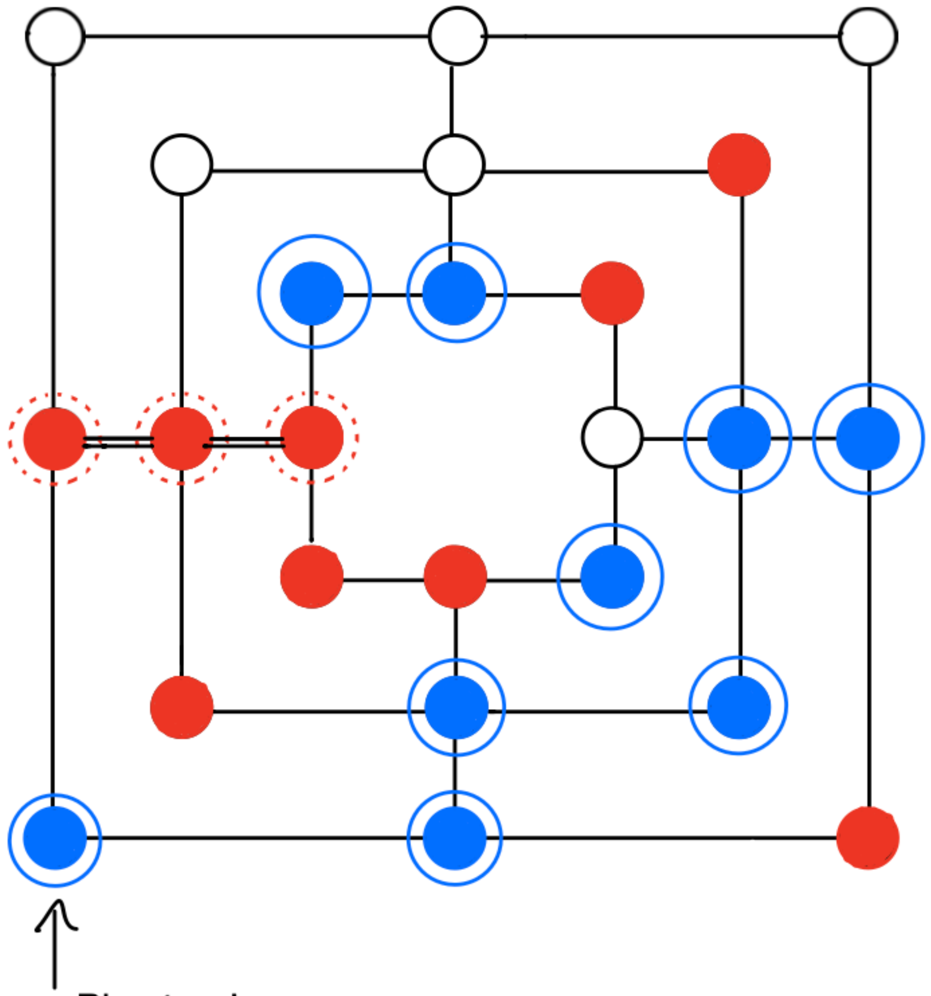
$\xrightarrow{B}$

Red turn!
You want to move to:

A. 0, 6
B. 6, 12

$\longrightarrow A$

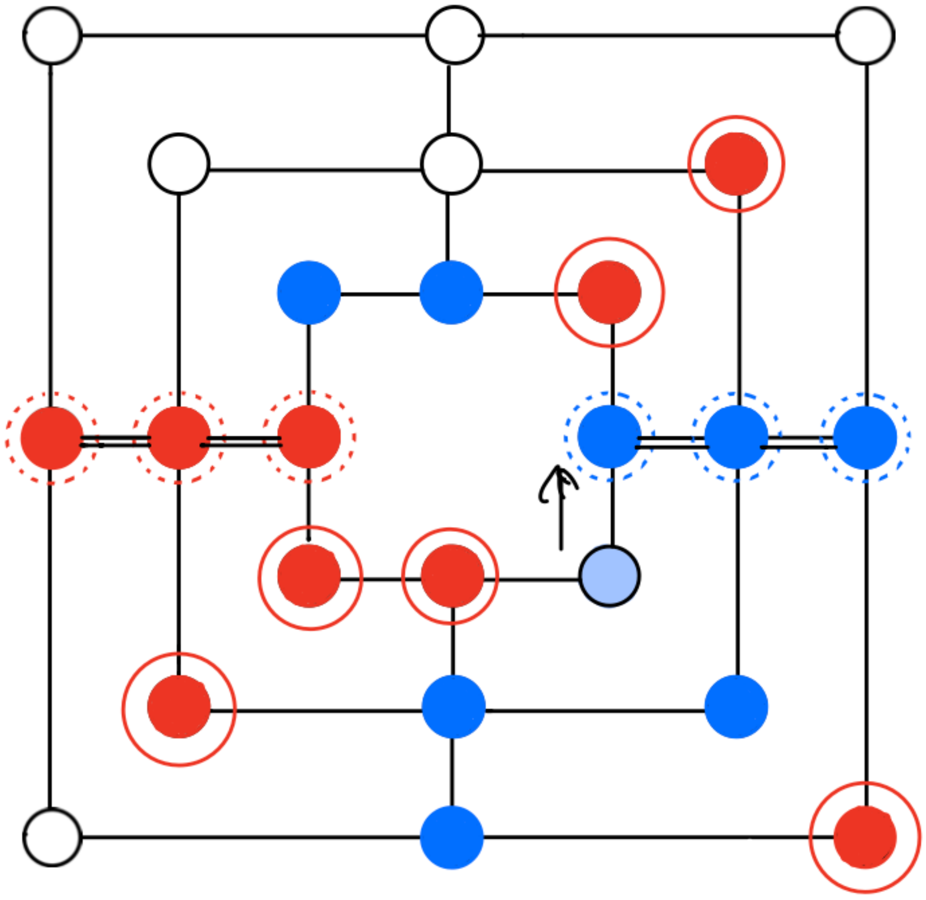
Stage 2:
When forms a mill can take a token from opponent just same as the situation in stage one



Blue turn!
Choose one token you want to take: A

A. 0, 0
B. 12, 6
C. 6, 0
D. 6, 2
E. 10, 2
F. 6, 8
G. 8, 4
H. 4, 8
I. 10, 6

Stage 2
When forms a mill, like stage one
not able to take the token from mill



Blue turn!
Choose one token you want to take:

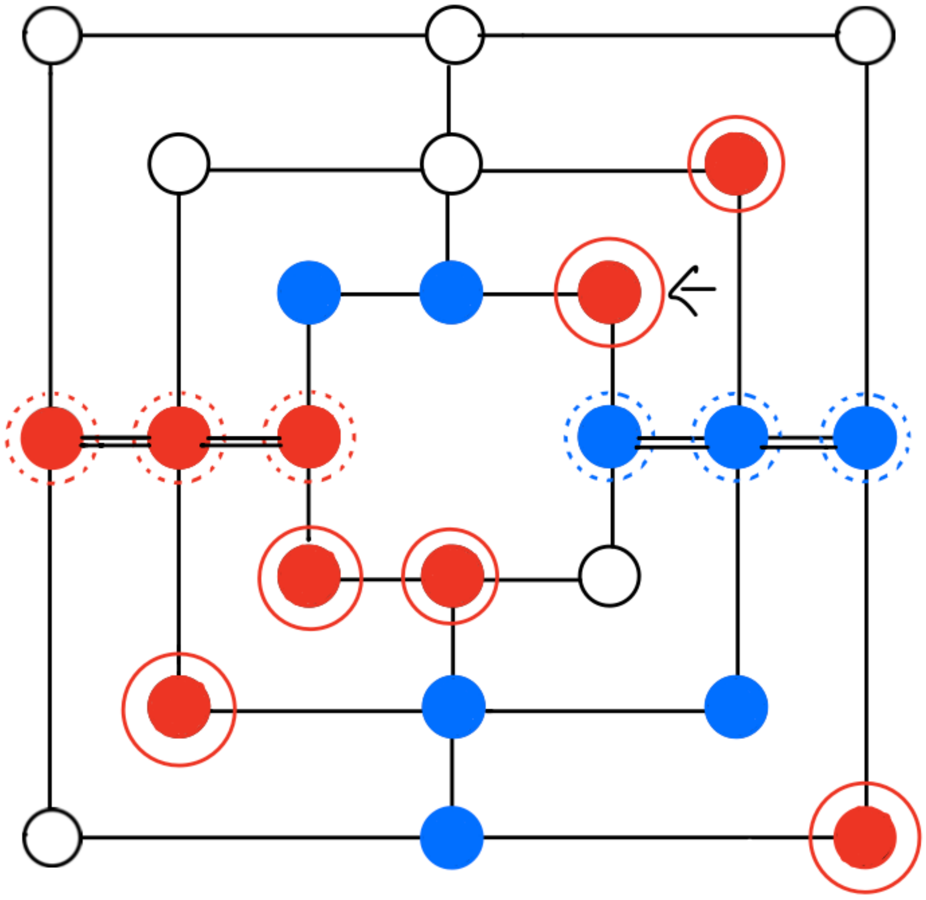
A. 12, 0
B. 10, 10
C. 6, 4
D. 2, 2
E. 4, 4
F. 8, 8

$\xrightarrow{F}$

Blue turn!
You want to move to:

A. 8, 6 $\longrightarrow$ A

Stage 2
After we take the token, we repeat these steps



Blue turn!
Choose one token you want to take: F

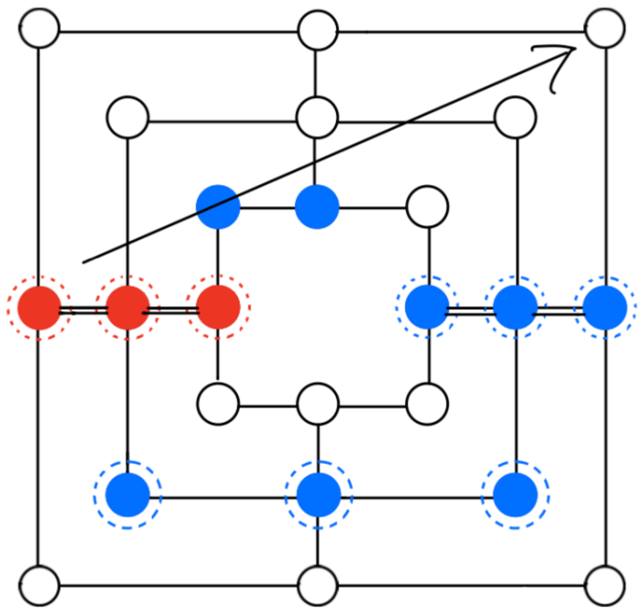
A. 12, 0
B. 10, 10
C. 6, 4
D. 2, 2
E. 4, 4
F. 8, 8

4.6 Stage Three

Stage 3

After several round, it may change to this situations, and we will in the stage 3 (flying stage)

you can move your token to all the Vacant point on the board



Red turn!

Choose one token you want to move:

A. 0, 6
B. 2, 6
C. 6, 6

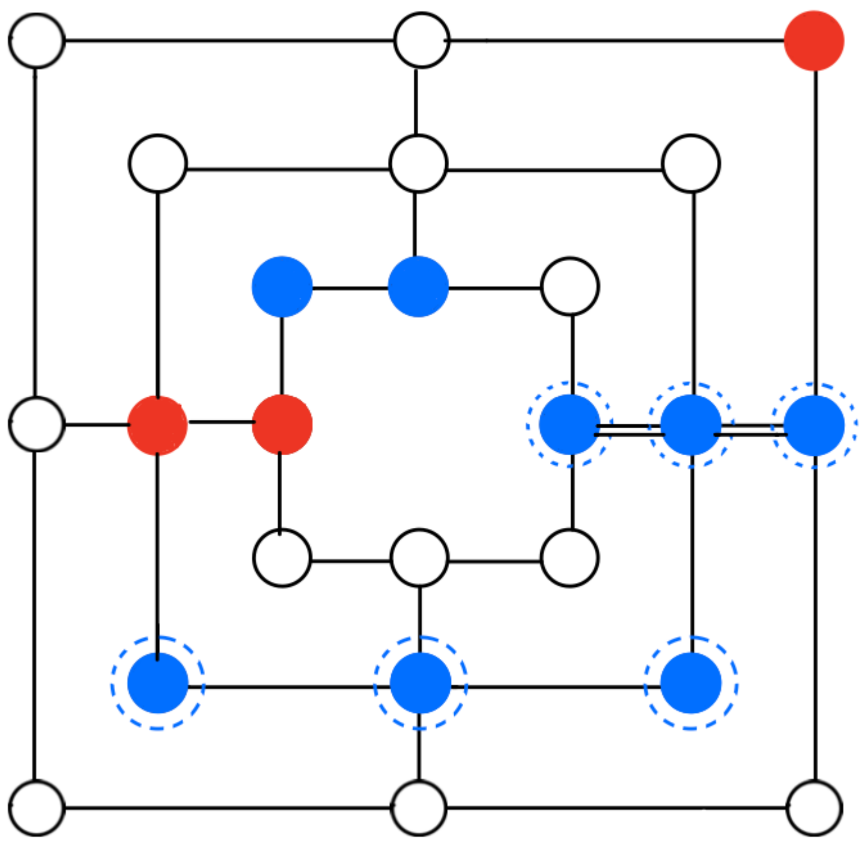
A

Red turn!

Input the place you want to move to:
12, 12

Stage 3

The side only left 3 token, can move any token to any vacant point



Blue turn!
Choose one token you want to move:

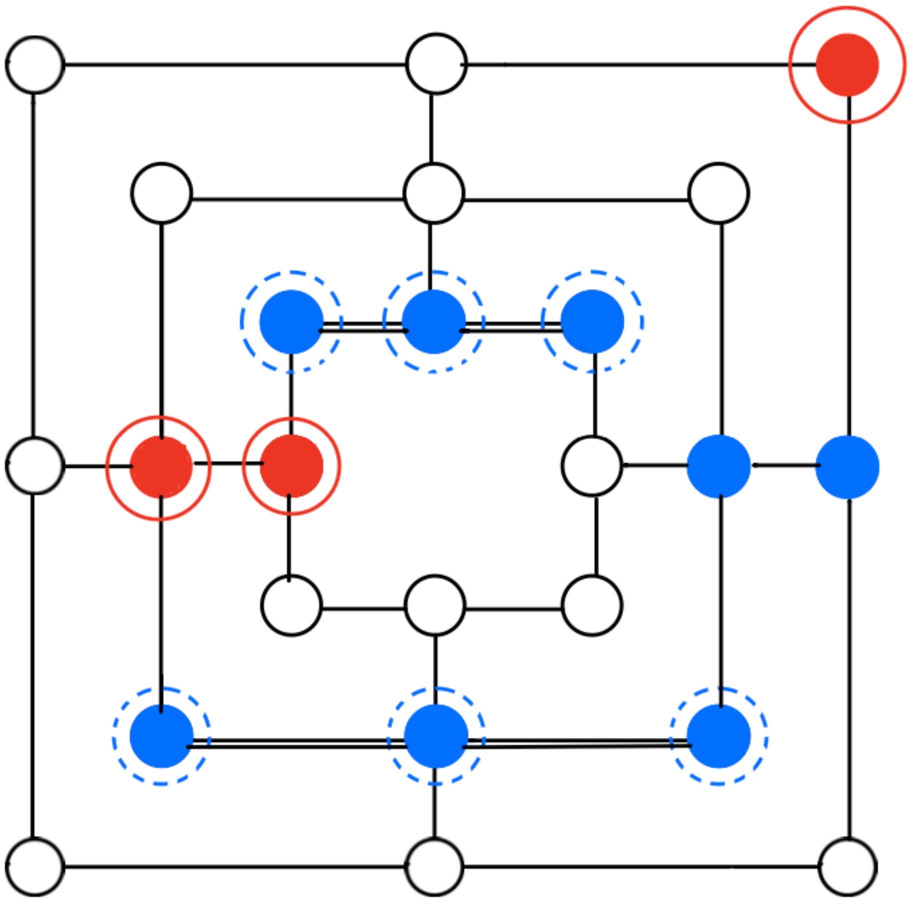
A. 12, 6
B. 6, 2 $\xrightarrow{F}$
C. 2, 2
D. 10, 2
E. 6, 8
F. 8, 6
G. 4, 8
H. 10, 6

Blue turn!
You want to move to:

A. 8, 8
B. 8, 4 $\longrightarrow$ A

4.7 Stage Four

Stage 4
When one side have only 2 token left, the other side win

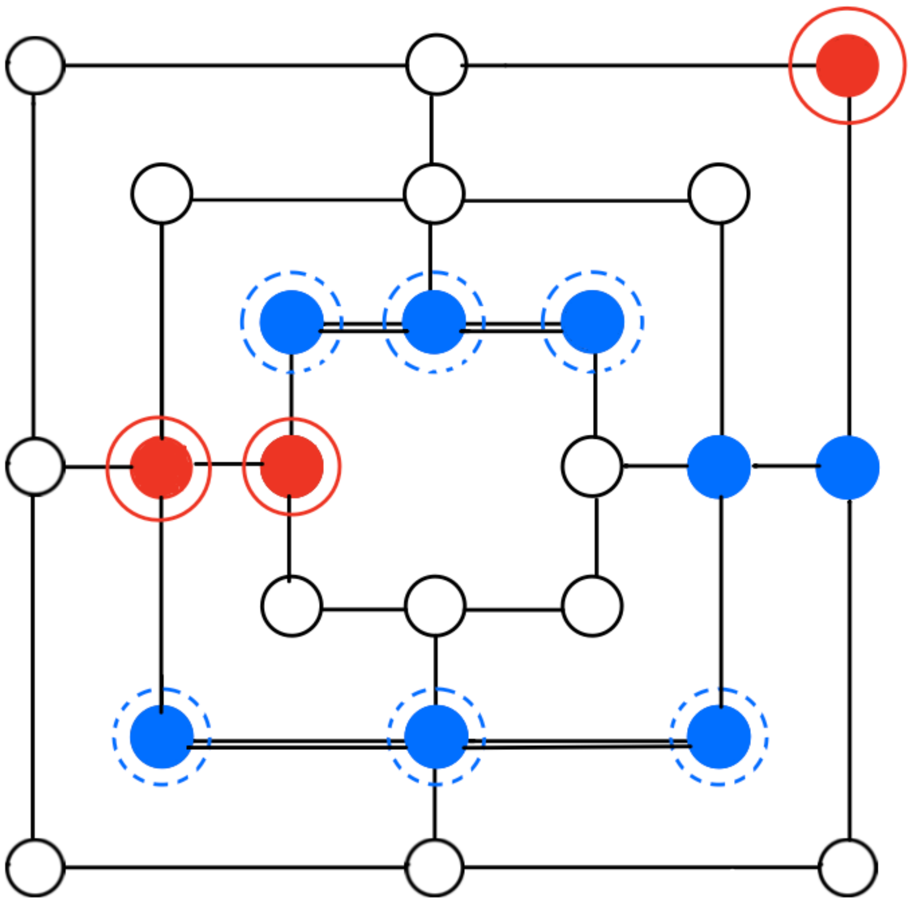


Blue turn!
Choose one token you want to remove: C

A. 2, 6
B. 4, 6
C. 12, 12

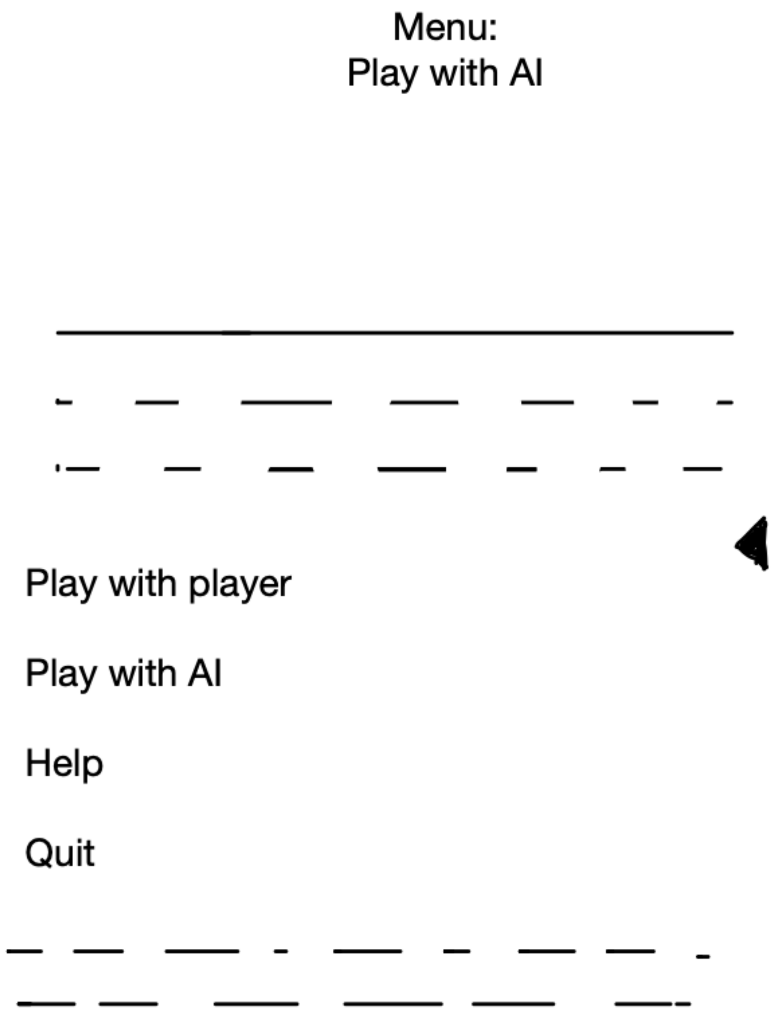
Stage 4

One side win

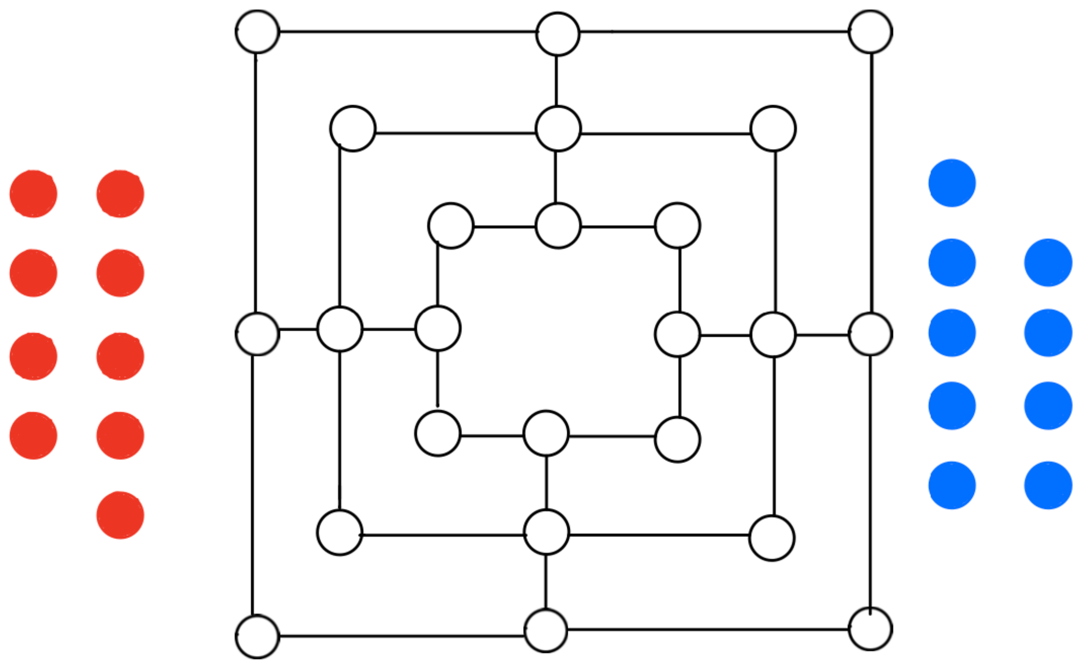


Blue Win!

4.8 AI



Choose side
Player can choose head or tail
to see if they can choose side

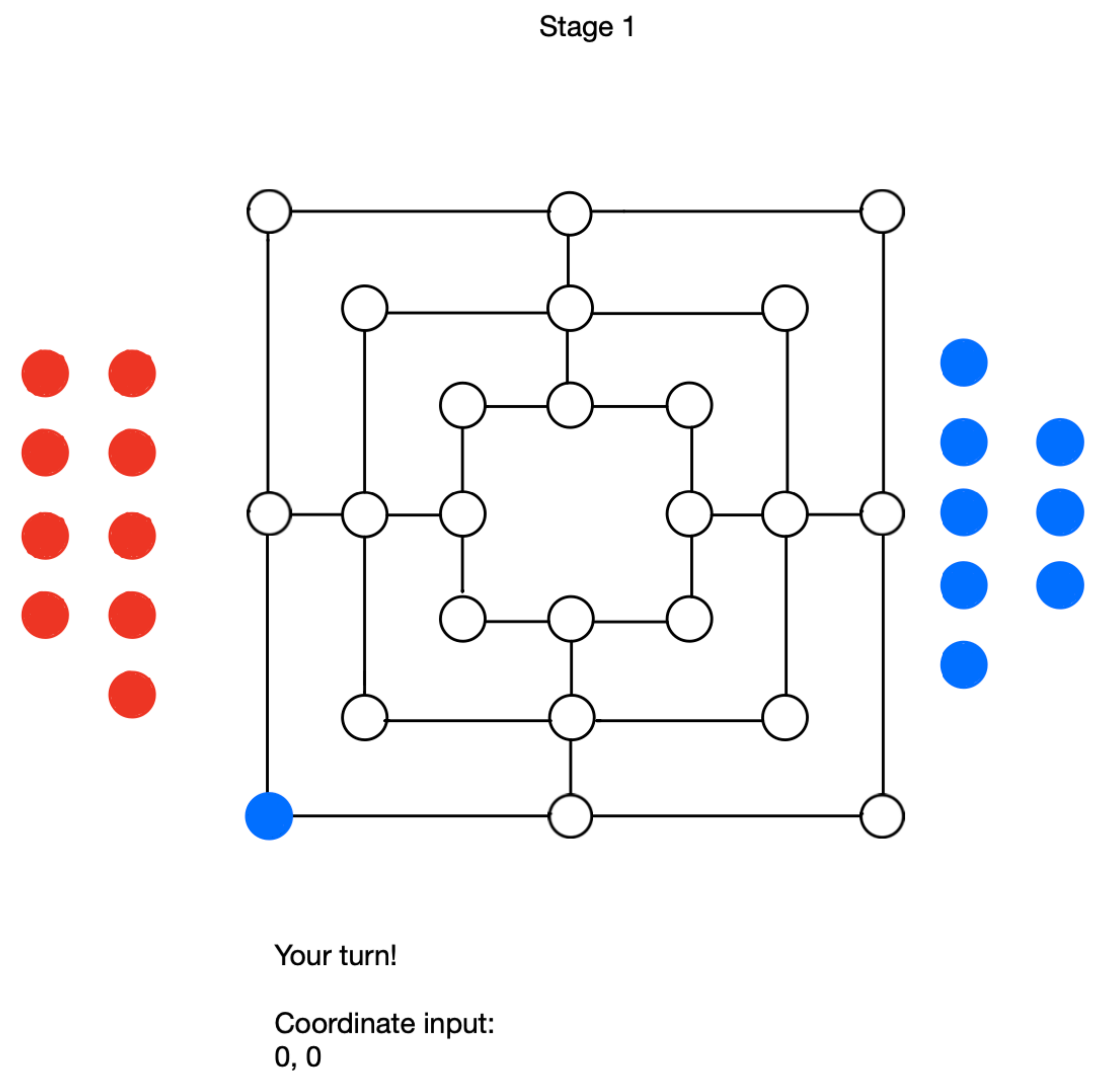


Player choice:
A. Head
B. Tail

or

Head!
Player choice your side:
A. Blue(first)
B. Red(second)

Tail!
You are on the red side

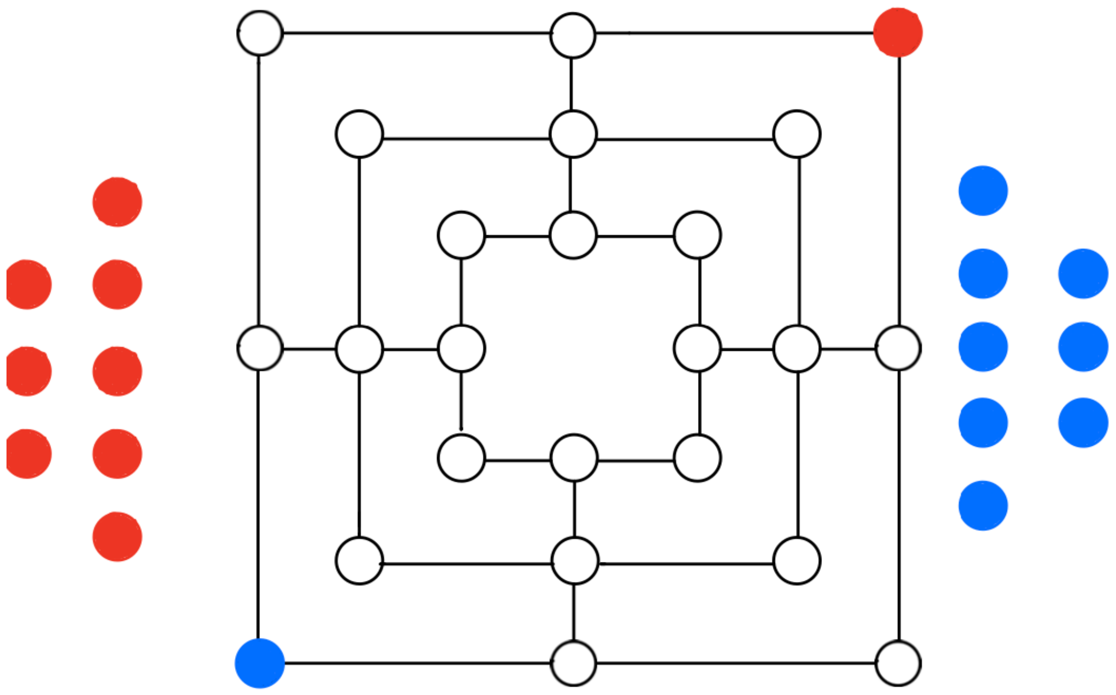


Stage 1

Your turn!

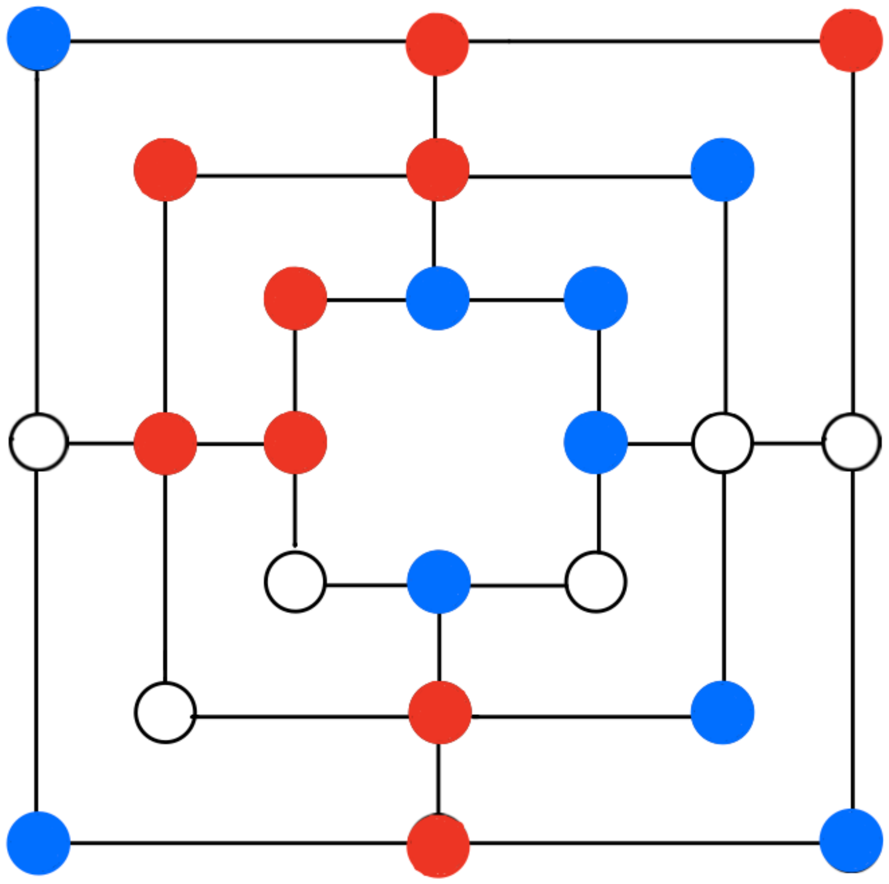
Coordinate input:
0, 0

End of stage 1



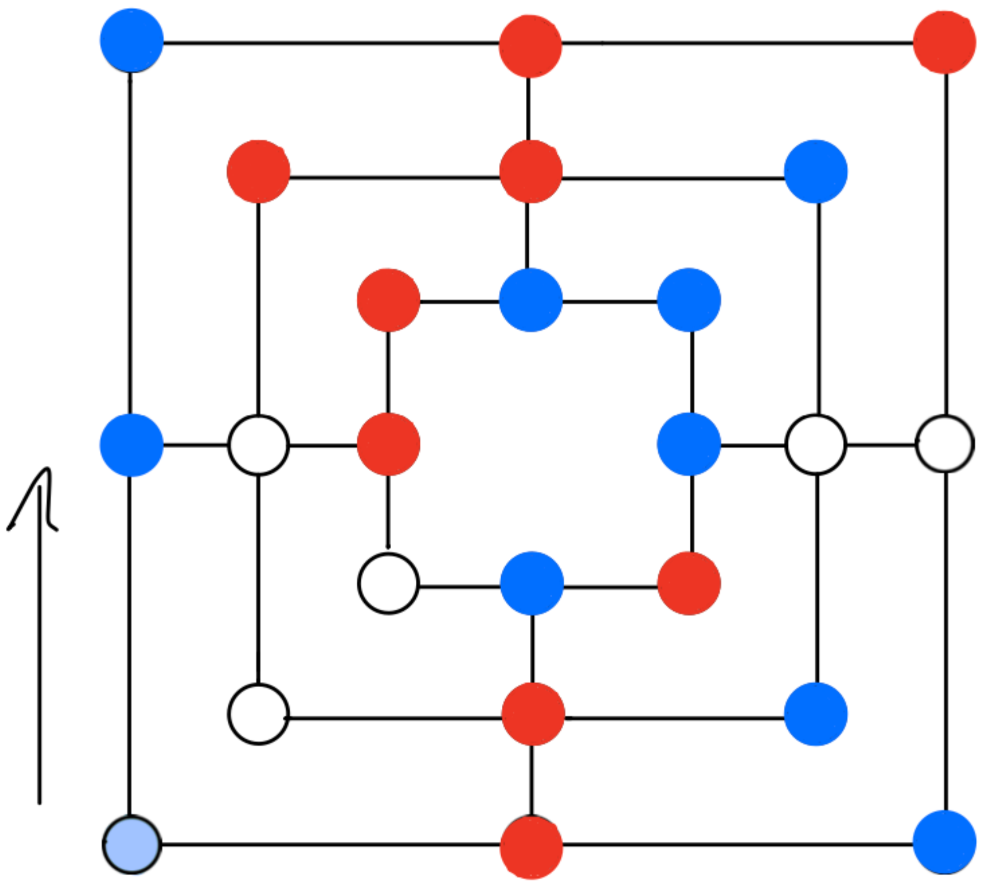
AI place a token at 12, 12

End of stage 1



When player and AI finish placing their token,
we will finish stage 1

Stage 2

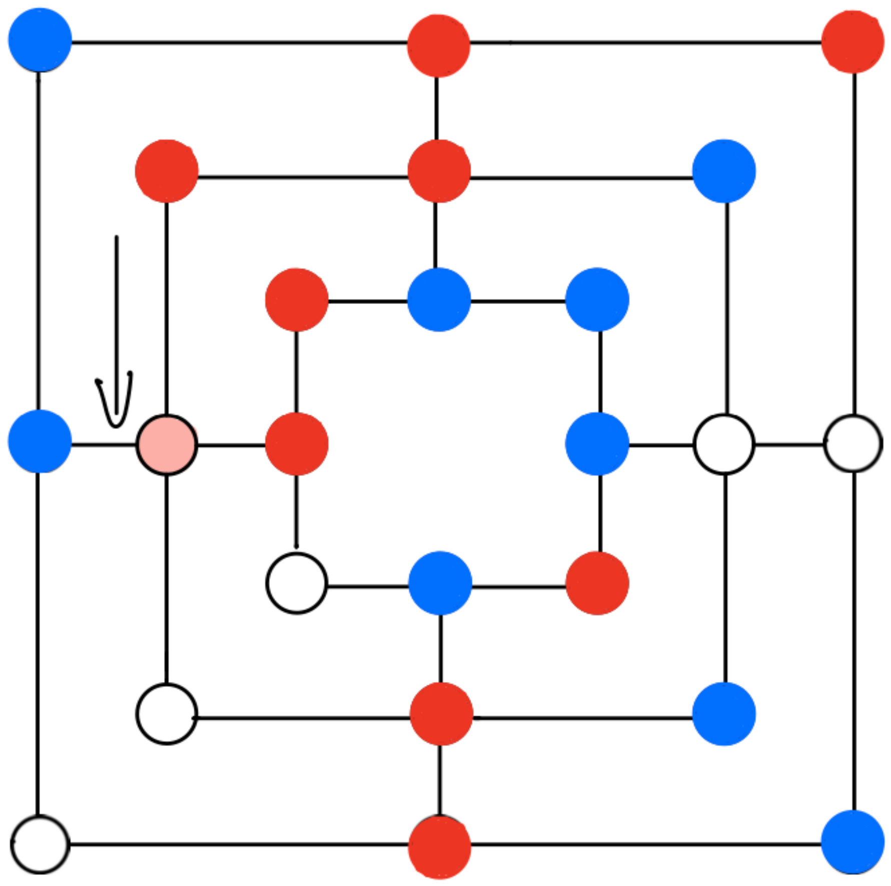


Your turn!

Choose one token from you:
A. 0, 0
B. 0, 12
C. 12, 0
D. 10, 2
E. 10, 10
F. 6, 6
G. 6, 10
H. 8, 6
I. 8, 10

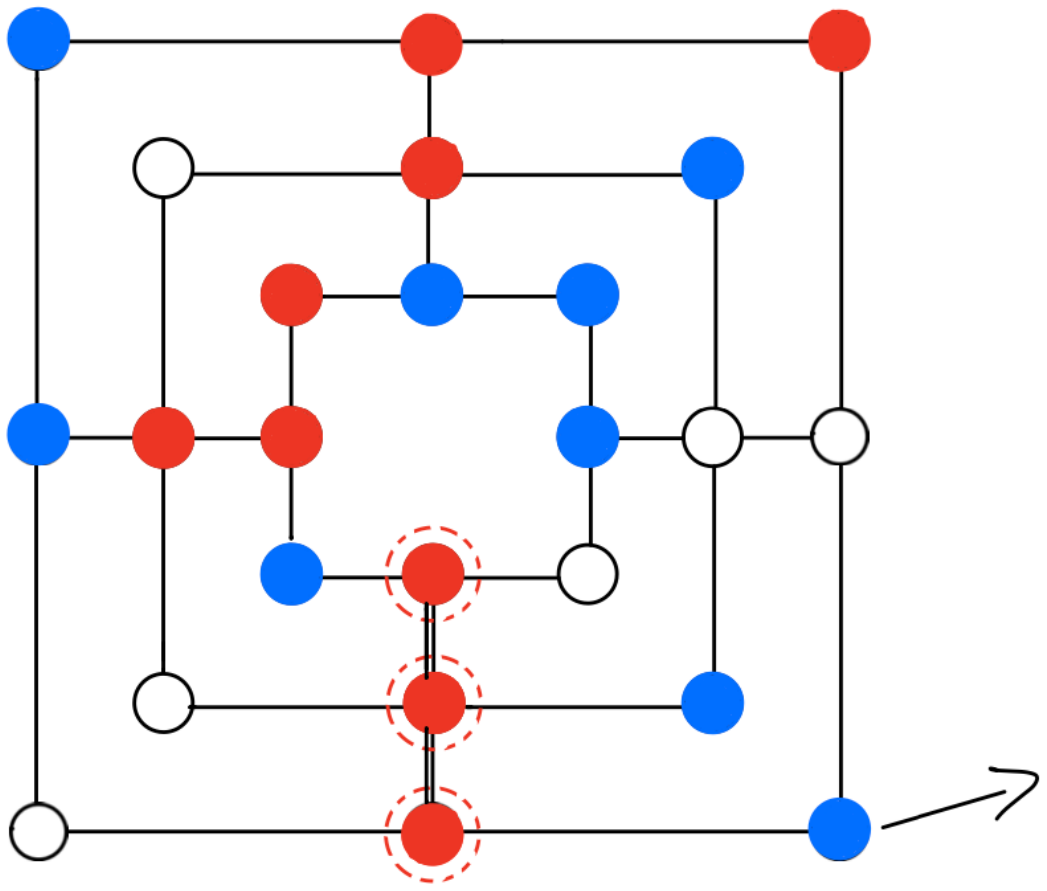
You want move to:
A. 0, 6

Stage 2

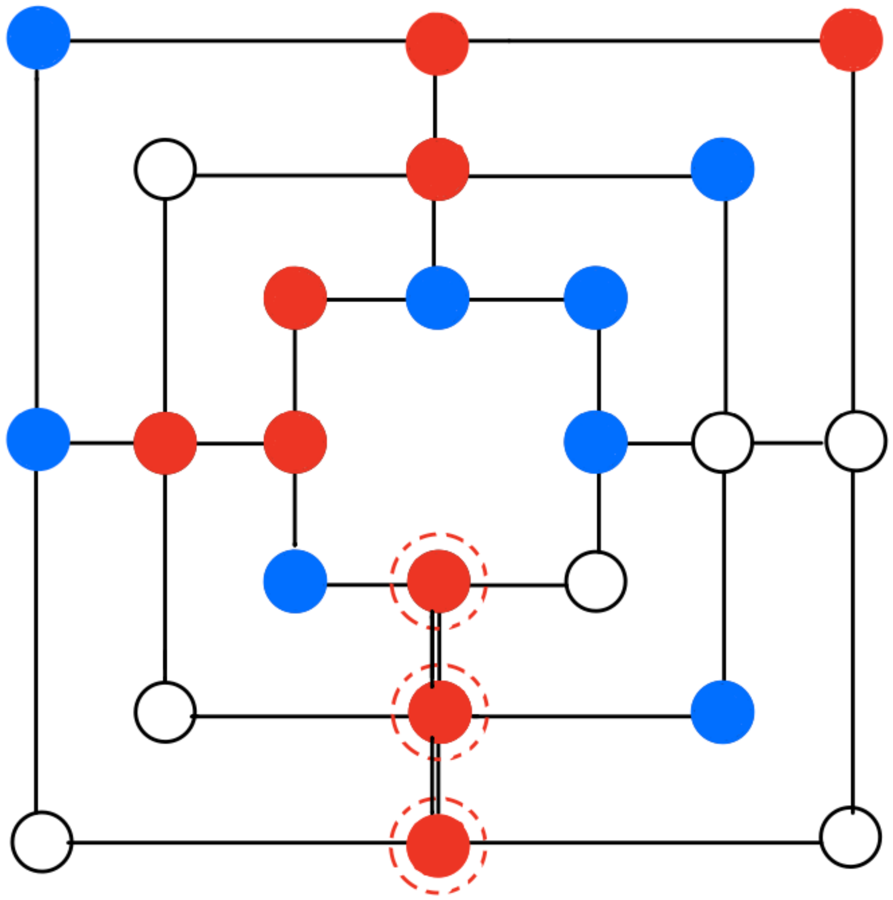


Computer move token on (2, 10) to (2, 6)

After several turns, computer might form a mill



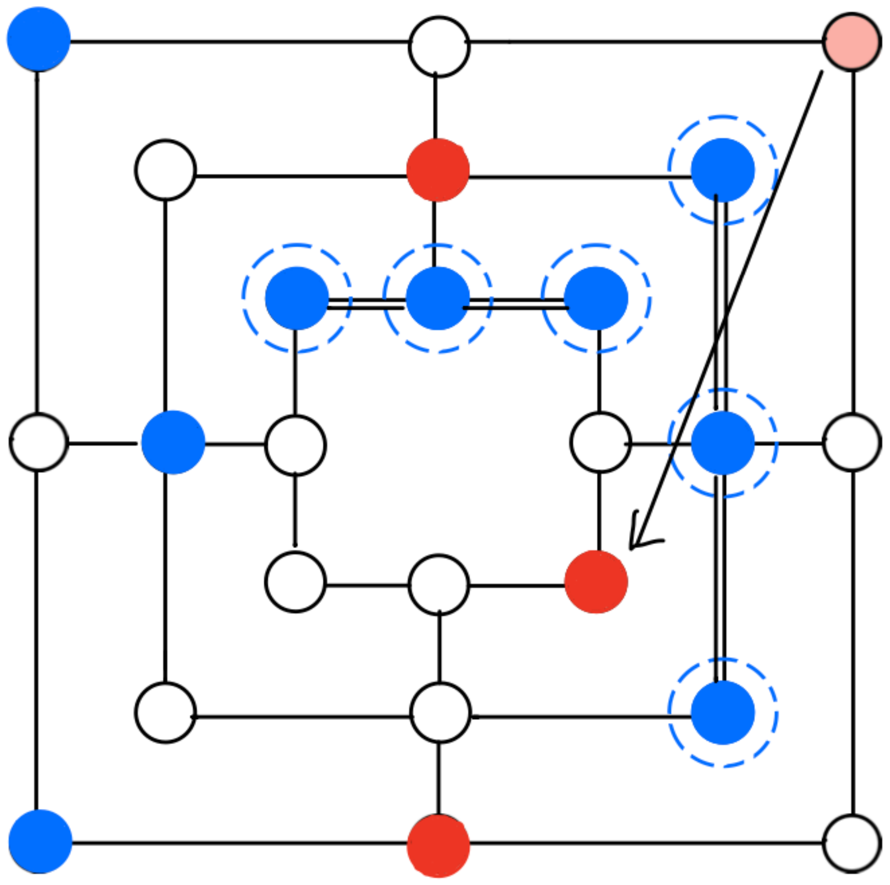
AI forms a mill, computer choose to take your token on (12, 0)



None of your token will be highlighted because computer will know which token it can take.

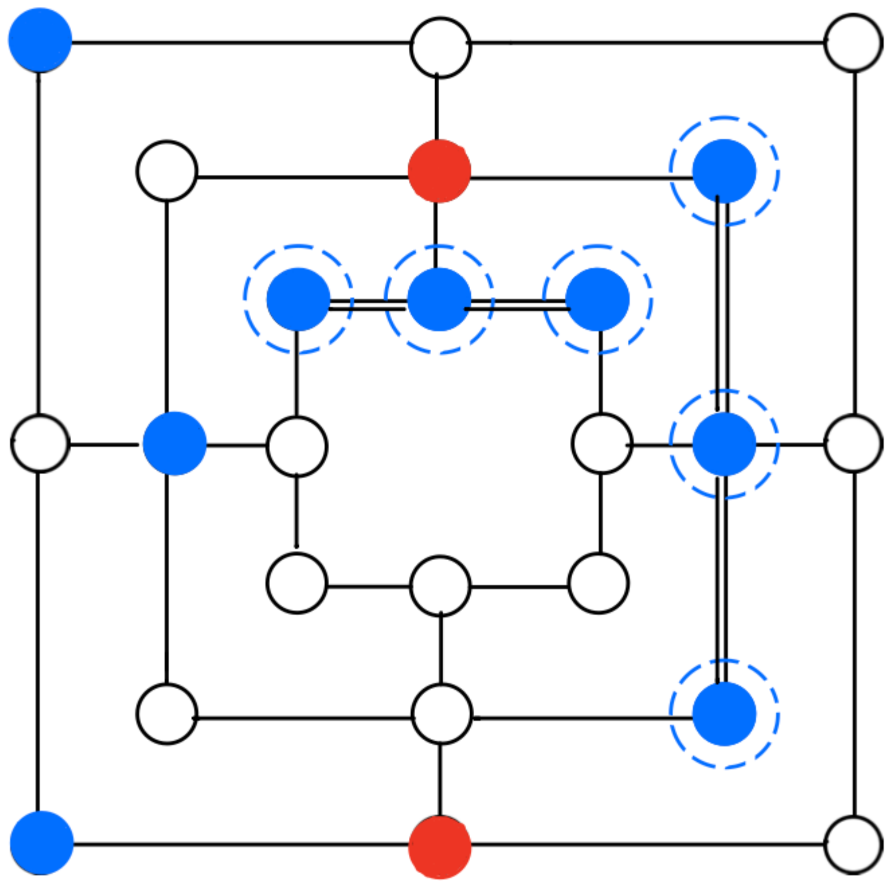
All the other part remains same till the end of the game

Stage 3
Now we look at what will happened when AI in flying stage



AI move the token at (12, 12) to (8, 4)

Stage 4
When you win the game



Congratulations!
You are better than the AI!

5. Reference

[1] Gasser, Ralph (1996). "Solving Nine Men's Morris" (PDF). *Games of No Chance*.