# QUORIDOR


**ARTIFICIAL INTELLIGENCE FOR VIDEOGAMES**
**A.A. 2021/2022**

**UNIVERSITÀ DEGLI STUDI DI MILANO**

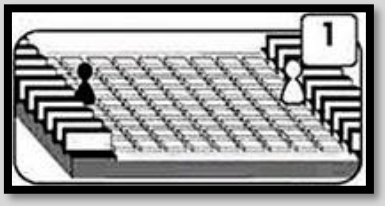
# Summary

**1. Introduction**.....3
1.1.Game rules.....3
1.2.Project concept.....6
**2. Game design**.....7
2.1.Game board.....7
2.2.Moving pawns.....8
2.3.Placing fences.....8
**3. Artificial intelligence**.....9
3.1.The algorithm.....9
3.2.AI design.....10
3.3.AI implementation.....11
3.4.Alternatives.....16
**4. Conclusion**.....17

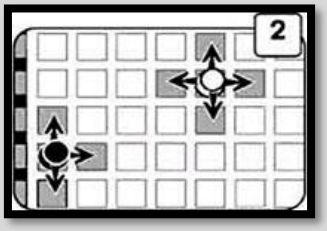
# 1. Introduction

## 1.1. Game Rules

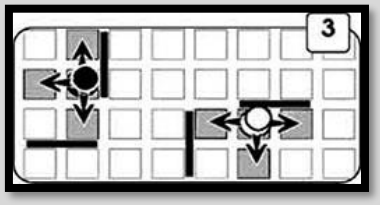
Each player in turn, chooses to move his pawn or to put up one of his fences. When he has run out of fences, the player must move his pawn.
At the beginning the board is empty. Choose and place your pawn in the center of the first line of your side of the board, your opponent takes another pawn and places it in the center of the first line of his side of the board (the one facing yours). Then take 10 fences each.




**Pawn moves**
The pawns are moved one square at a time, horizontally or vertically, forwards or backwards, never diagonally.

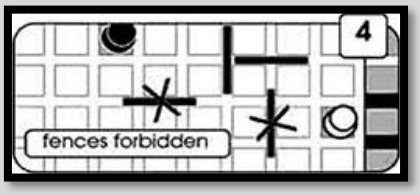



The pawns must bypass the fences. If, while you move, you face your opponent's pawn you can jump over.

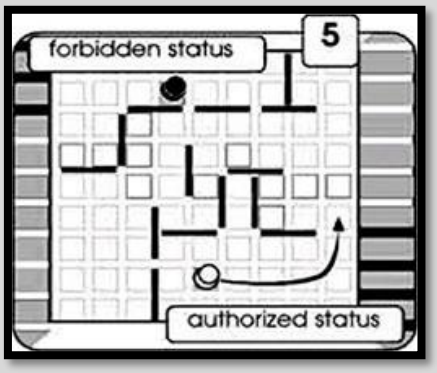


### Positioning of the fences

The fences must be placed between 2 sets of 2 squares.

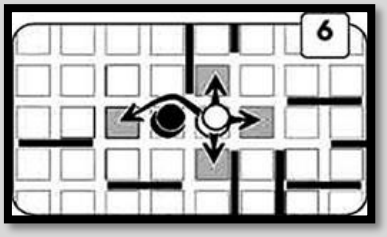



By placing fences, you force your opponent to move around it and increase the number of moves they need to make. But be careful, you are not allowed to lock up to lock up your opponents pawn, it must always be able to reach it's goal by at least one square.

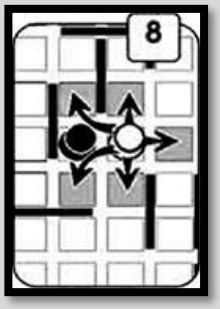


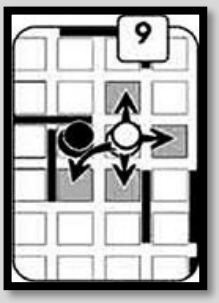

### Face To Face

When two pawns face each other on neighboring squares which are not separated by a fence, the player whose turn it is can jump the opponent's pawn (and place himself behind him), thus advancing an extra square.

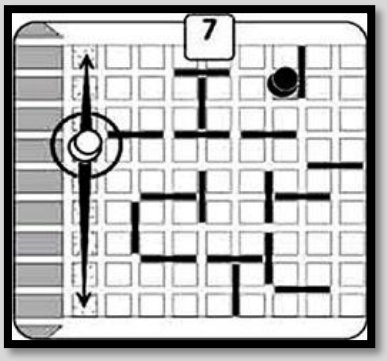


If there is a fence behind the said pawn, the player can place his pawn to the left or the right of the other pawn.

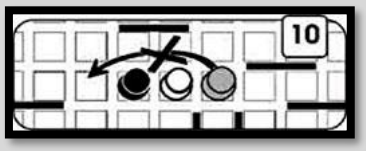



**End of the game**
The first player who reaches one of the 9 squares opposite his base line is the winner.

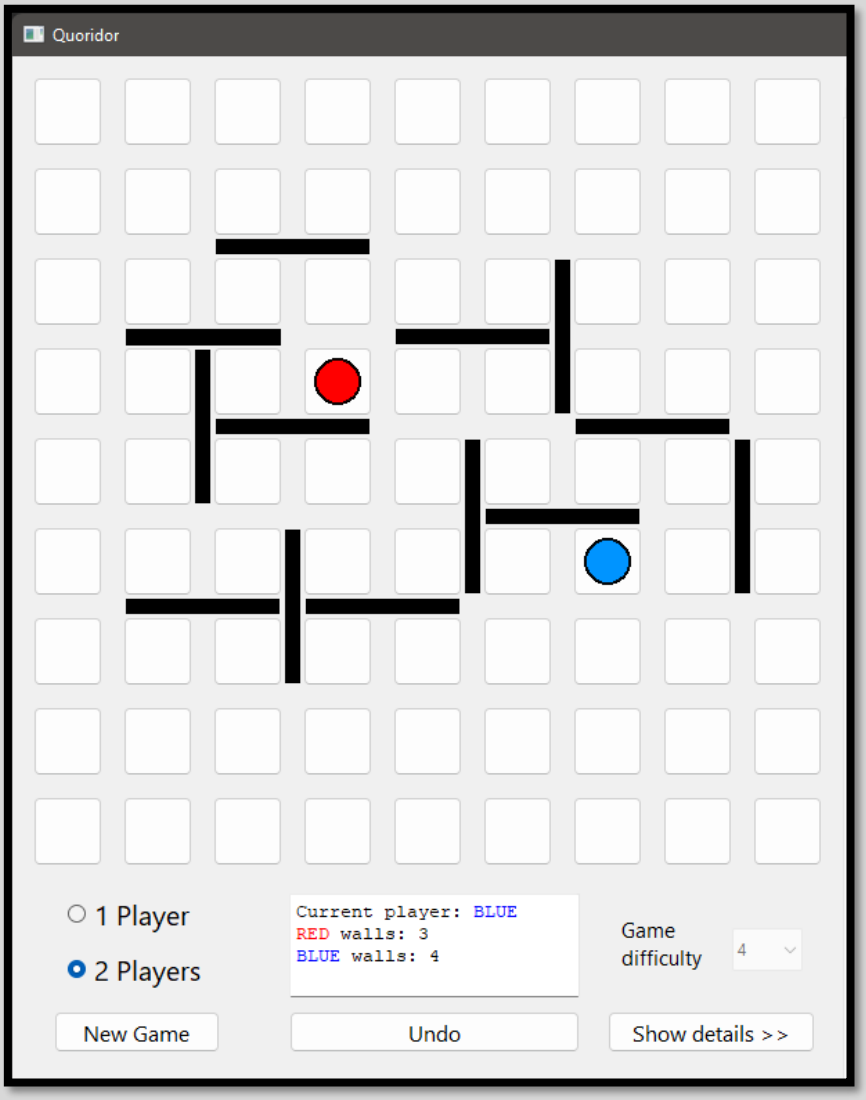



**Rules For 4 Players**
When the game starts, the 4 pawns are placed in the center of each of the sides of the board and each player is given 5 fences.
The rules are identical to those for two players, but it is forbidden to jump more than one pawn.



## 1.2. Project concept

The goal of this project is to create an agent for this game and being able to play a realistic match againt the computer. The agent not only have to follow the game rules, but also being able to create a challenge for the player by placing fences and moving his piece in a reasonably way in order to achieve victory. The design of the AI will be covered later. Before that, it's essential first to build the game interface so that's possible to apply the game rules.




The game interface is very simple: the game board is made of 81 buttons and each one is programmed to move the player into a valid position and fences can be placed between these buttons. The player will be able to play against the computer or with another human friend.

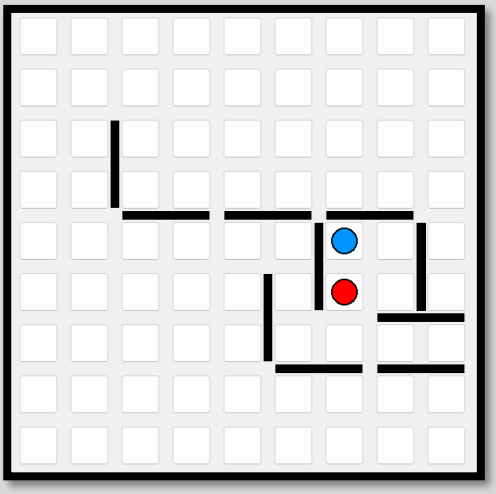
The project is created with the open source version of QT Creator 6.3 and the code is written in C++.

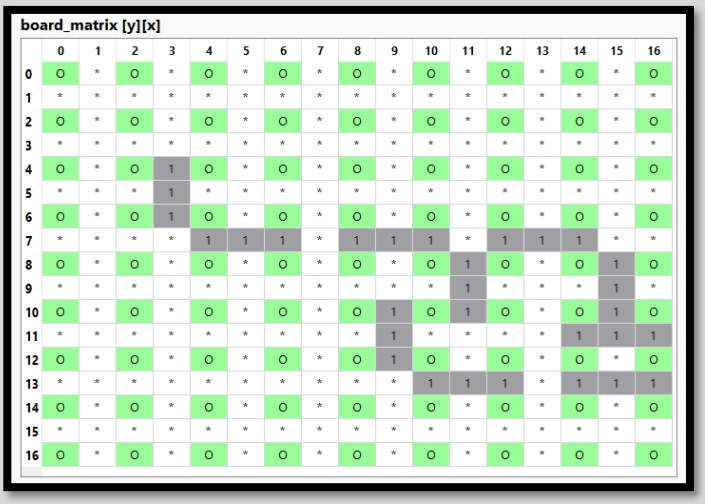
# 2. Game deisgn

## 2.1. Game board

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ● | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ● | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

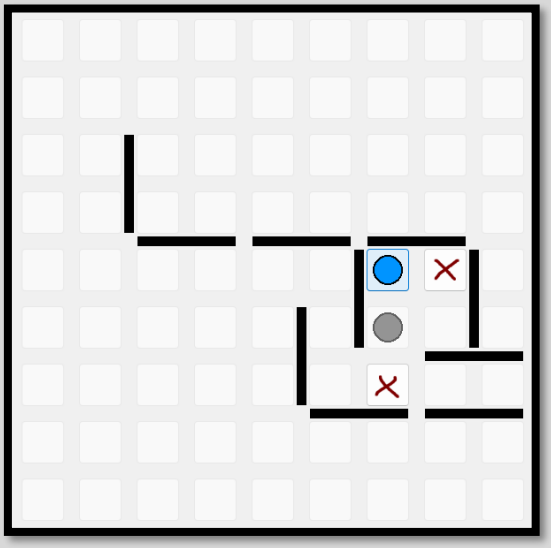
The game board is basically a bidimensional 17x17 matrix with each cell initialized at “zero” (‘0’). If there is a wall at (y, x), that position’s value will become ‘1’. The players position will not be contained inside this matrix so we don’t have to search the player’s coordinates for every position, and the buttons won’t trigger any change to this board. The matrix will only store informations about the walls’ positions. Each wall will take three vertical or horizontal consecutive cells.





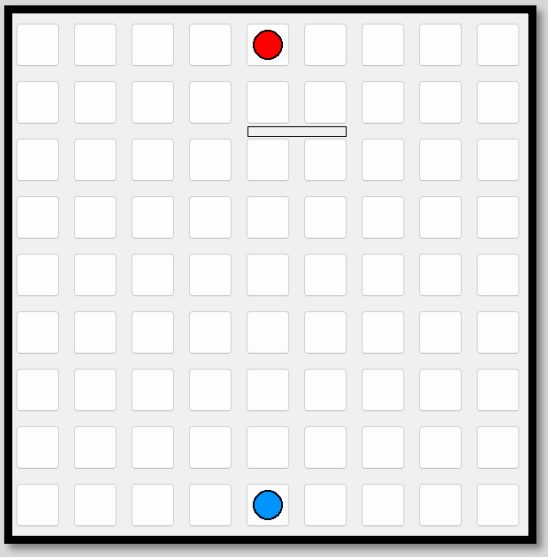
## 2.2. Moving pawns

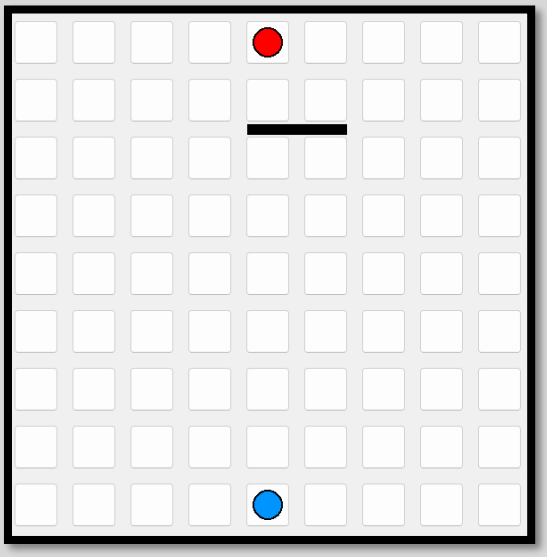
The player's position is referred to the game board but not memorized over there. When pressing the button at the player's position, the buttons near the player will be activated and the others deactivated until the valid one is pressed or it doesn't make any move. After selecting the next place, the player's coordinates will be updated and the turn will be passed to the next player (would've been better to have mapped these button instead of programming each one). When discovering the next possible moves, there is a function that will look into the game board and the opponent's position in order to apply the game's rules.



## 2.3. Placing fences

The game program follows the user's mouse pointer and highlights a possible position to place wall and with the left click, it will display a solid wall only if it's a valid position. This means that the current player need to have enough walls (from 1 to 10) and the wall placement must leave at least one possible path for both player to reach their goal. Before placing the wall the program will perform a search on the game board to find any possible path to the goal for both players.

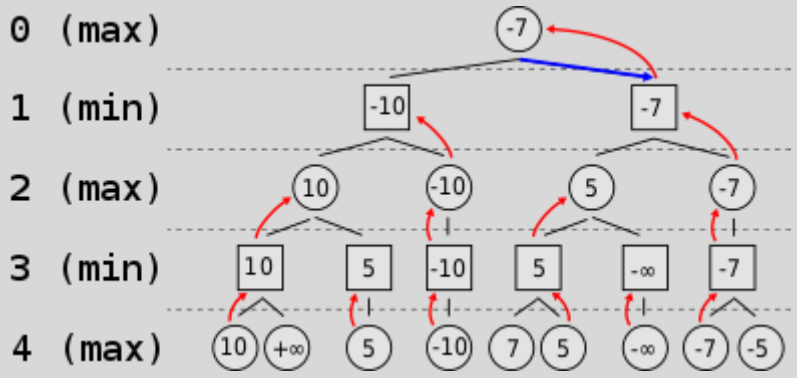




# 3. Artificial intelligence

## 3.1. The algorithm

The algorithm used to determine the next move is the popular minimax algorithm. It is the simplest algorithm used to calculate the next best move for turn-based table games like tic-tac-toe or chess and works quite well when there are a very limited number of moves.
The idea is to predict the next moves for both players and see which one gets the best result, assuming both players are playing optimally. Then the computer will have to make all possible combinations of moves (a lot of moves!) And at the end make an evaluation for each one and choose the move that is most beneficial for itself and least for the opponent.




For example, here we have a two player game where only two moves are available. The algorithm will perform all 16 possible moves for the next 4 rounds and assign a "score" for each move. A score is generated by making an assessment of the state of the game and is used to understand the situation of the game at that moment (for example which player is closest to victory). Assuming we need the highest score to win the game, the opponent must have the lowest.

The pseudocode for the depth-limited minimax algorithm is given below.

```
0
1   function  minimax(node, depth, maximizingPlayer) is
2       if depth = 0 or node is a terminal node then
3           return the heuristic value of node
4       if maximizingPlayer then
5           value := −∞
6           for each child of node do
7               value := max(value, minimax(child, depth − 1, FALSE))
8           return value
9       else (minimizing player)
10          value := +∞
11          for each child of node do
12              value := min( value, minimax(child, depth − 1, TRUE))
13          return value
14
```

Now let’s see how we can apply this algorithm in Quoridor.

## 3.2. AI design

In this game, the goal is to reach the opposite side of the field, but on closer inspection we realize that all we do is make our path shorter and that of the opponent longer. So we can say that at each round, all we do is basically solving pathfinding problems.

But there are other things to consider too, such as the number of walls left and the choice between making a move or placing a wall. There are also (8x8) horizontal + (8x8) vertical = 128 possible positions of the wall on a new board and 4 directions in which the pawn can move.

The minimax algorithm implies that for each possible move we consider all other possible moves for the following rounds. So we have something like $(128 + 4)^k$ possible final outputs and for each of them we do an evaluation to see which next move leads to the best result. Simulating all those moves can be a bit stressful for the computer, and that's one of the downsides of this algorithm.

So, in order to apply the minimax algorithm for this game, we have drastically reduced the number of moves to only two possible moves: one step towards the goal, or placing a wall that increases the opponent's distance from the goal.

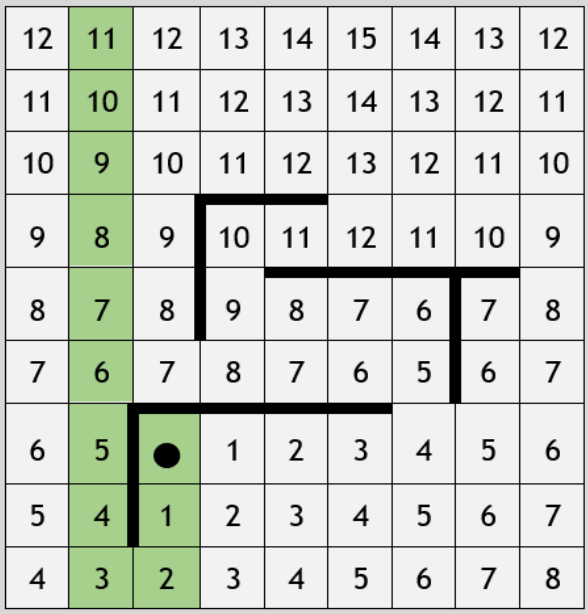
For this we need a program that calculates the shortest distance to victory.

Since we are using a matrix that contains the walls in the right positions, calculating the shortest path is quite simple: just apply Dijkstra's algorithm which solves mazes from the starting point to one of the possible objectives. In this case you don't need a precise goal position but just go "forward" towards the goal. At the end of the execution we will be able to calculate the distance to the goal and the position of the final position.

The algorithm in a nutshell:

1. Determine the starting point of the grid (current player position).
2. Record the cost of reaching that cell: 0.
3. Find that cell’s navigable neighbors.
4. For each neighbor, record the cost of reaching the neighbor: 1.
5. For each neighbor, repeat steps 3-5, taking care not to revisit already-visited cells.

At the end we have somenthing like this:



With this useful program we will be able to figure out which next move is the best or which wall can extend the path for the opponent.

The next step is to generate the output of the minimax tree: following the algorithm we start from two successive moves (pawn or wall) and for each one we make the opponent make the next moves and so on. In the end we have at most $2^k$ possible moves. Among these we discard the useless ones. For example, if there are no more placeble walls, we only consider moving the pawn.

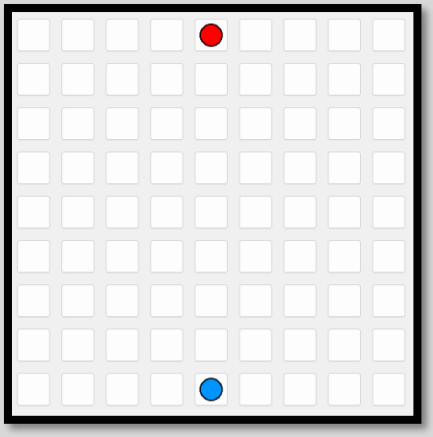
## 3.3. AI implementation

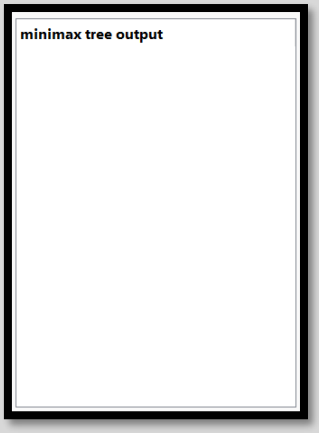
To implement this algorithm in the program it is necessary to record the state of the game at each move and each state must be independent from the others. A game state consists of the board matrix, which contains all the positions of the walls and the positions of both players, which are not stored in the matrix. With this information we can do all the simulations we want.

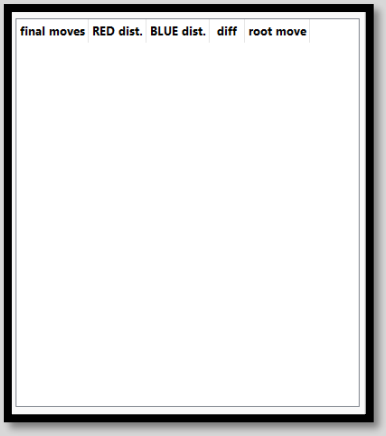
The minimax algorithm applied here is little different and has the following steps:

1. Generate a snapshot of the current game state (walls and player positions).
2. Determine the next move that takes the fewest steps towards the goal and generates a new game state.
3. Determine the best position for the wall to slow down the opponent and generate a new game state.
4. For each move, recursively call the function passing the game state.
5. When it has reached the final depth, it will calculate the distance of both players from the goal and calculate their difference (red distance - blue distance) and determine the lowest difference. This means that either the RED player is near the goal or the BLUE player is very far away. From the result we go back and find the next move to perform.

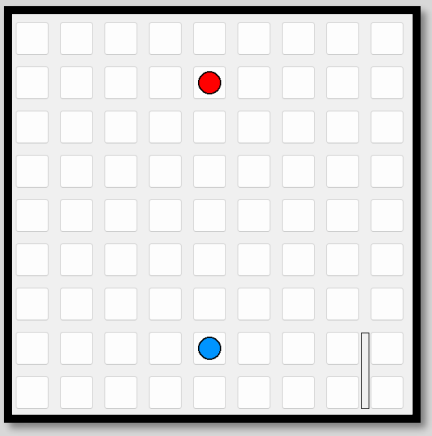
Here's an example of how it works:






Game starts. Both players are distant 8 blocks away from their respective goal, and BLUE player always starts first. For every BLUE player input, the computer will try to predict the next 4 moves ahead. That means we can expect at most $2^4 = 16$ possible combinations of moves and for each of one of them, if reachable, we'll calculate both players distance to the goal and we'll pick the one which the RED player has the lowest one and with the BLUE players highest. So we just have to take the lowest difference between RED and BLUE player's distance.

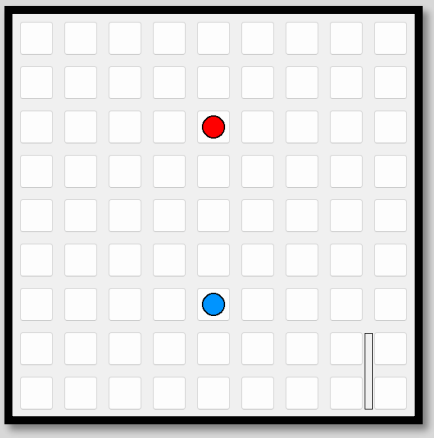


**minimax tree output**

- RED_turn
  - RED_move: m 2 8
    - BLUE_move: m 12 8
      - RED_move: m 4 8
        - BLUE_move: m 10 8
        - BLUE_wall: h 13 8
      - No wall
        - No move
        - No wall
    - BLUE_wall: v 14 13
      - RED_move: m 4 8
        - BLUE_move: m 12 8
        - BLUE_wall: v 14 11
      - RED_wall: h 13 8
        - BLUE_move: m 14 6
        - BLUE_wall: v 14 11
  - RED_wall: h 13 8
    - BLUE_move: m 14 6
      - RED_move: m 2 8
        - BLUE_move: m 12 6
        - No wall
      - No wall
        - No move
        - No wall
    - BLUE_wall: v 14 13
      - RED_move: m 2 8
        - BLUE_move: m 14 6
        - BLUE_wall: v 14 11
      - RED_wall: v 14 7
        - BLUE_move: m 14 10
        - BLUE_wall: v 14 5

| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 10 8 | 5 | 4 | 1 | m 2 8 |
| 2 | h 13 8 | 6 | 5 | 1 | m 2 8 |
| 3 | m 12 8 | 5 | 5 | 0 | m 2 8 |
| 4 | v 14 11 | 5 | 6 | -1 | m 2 8 |
| 5 | m 14 6 | 7 | 7 | 0 | m 2 8 |
| 6 | v 14 11 | 8 | 8 | 0 | m 2 8 |
| 7 | m 12 6 | 7 | 6 | 1 | h 13 8 |
| 8 | m 14 6 | 7 | 7 | 0 | h 13 8 |
| 9 | v 14 11 | 8 | 8 | 0 | h 13 8 |
| 10 | m 14 10 | 9 | 8 | 1 | h 13 8 |
| 11 | v 14 5 | 9 | 9 | 0 | h 13 8 |

The BLUE player moved forward. RED player have to decide between moving to the position [2,8], or place a horizontal wall at [13,8]. For each one of these two moves, the computer makes his predictions. We decided to discard wall positions that increase RED players distance, so we have 11 moves out of 16. We can see that if we follow this order: RED-> move to [2,8], BLUE-> place vertical wall at [14,13], RED-> move to [4,8], BLUE-> place vertical wall at [14,11] (walls that increase both players distance by equal value are discarded), the RED player will be 5 blocks from goal while BLUE player is 6, and their difference is -1 which is the lowest. So the final decision would be to move forward instead placing a wall.



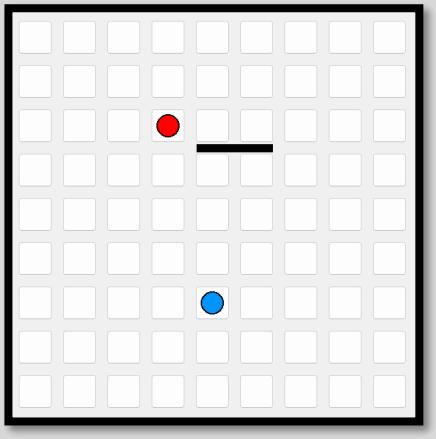
**minimax tree output**

- RED_turn
  - RED_move: m 4 8
    - BLUE_move: m 10 8
      - RED_move: m 6 8
        - BLUE_move: m 8 8
        - BLUE_wall: h 13 8
      - No wall
        - No move
        - No wall
    - BLUE_wall: h 13 8
      - RED_move: m 6 8
        - BLUE_move: m 10 8
        - BLUE_wall: h 13 4
      - RED_wall: h 11 8
        - BLUE_move: m 12 6
        - BLUE_wall: h 13 4
  - RED_wall: h 11 8
    - BLUE_move: m 12 6
      - RED_move: m 4 8
        - BLUE_move: m 10 6
        - No wall
      - No wall
        - No move
        - No wall
    - BLUE_wall: h 13 8
      - RED_move: m 4 8
        - BLUE_move: m 12 6
        - BLUE_wall: h 13 4
      - RED_wall: v 12 7
        - BLUE_move: m 12 10
        - BLUE_wall: h 13 12

| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 8 8 | 4 | 3 | 1 | m 4 8 |
| 2 | h 13 8 | 5 | 4 | 1 | m 4 8 |
| 3 | m 10 8 | 5 | 4 | 1 | m 4 8 |
| 4 | h 13 4 | 6 | 5 | 1 | m 4 8 |
| 5 | m 12 6 | 6 | 6 | 0 | m 4 8 |
| 6 | h 13 4 | 8 | 7 | 1 | m 4 8 |
| 7 | m 10 6 | 6 | 5 | 1 | h 11 8 |
| 8 | m 12 6 | 6 | 6 | 0 | h 11 8 |
| 9 | h 13 4 | 8 | 7 | 1 | h 11 8 |
| 10 | m 12 10 | 8 | 7 | 1 | h 11 8 |
| 11 | h 13 12 | 8 | 8 | 0 | h 11 8 |

BLUE player makes another step forward. Same process like the previous one with similar results, the lowest difference now is 0, but there are in total 3 of them and they belong to different moves. The

algorithm will always pick the first one. Probably a deeper tree (making more predictions) would determine which one is better.

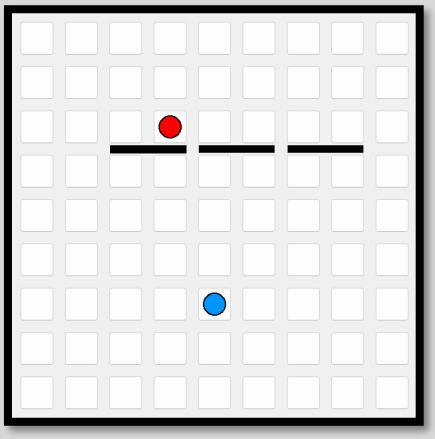


**minimax tree output**

- RED_turn
  - RED_move: m 4 6
    - BLUE_move: m 10 8
      - RED_move: m 6 6
        - BLUE_move: m 8 8
        - No wall
      - RED_wall: h 9 6
        - BLUE_move: m 10 6
        - BLUE_wall: h 13 8
    - No wall
      - No move
        - No move
        - No wall
      - No wall
        - No move
        - No wall
  - RED_wall: h 11 6
    - BLUE_move: m 12 6
      - RED_move: m 4 6
        - BLUE_move: m 12 4
        - BLUE_wall: h 13 6
      - No wall
        - No move
        - No wall
    - BLUE_wall: h 13 8
      - RED_move: m 4 6
        - BLUE_move: m 12 6
        - BLUE_wall: h 13 4
      - RED_wall: v 14 13
        - BLUE_move: m 12 6
        - BLUE_wall: h 13 4

| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 8 8 | 5 | 4 | 1 | m 4 6 |
| 2 | m 10 6 | 7 | 6 | 1 | m 4 6 |
| 3 | h 13 8 | 7 | 7 | 0 | m 4 6 |
| 4 | m 12 4 | 6 | 6 | 0 | h 11 6 |
| 5 | h 13 6 | 7 | 7 | 0 | h 11 6 |
| 6 | m 12 6 | 7 | 7 | 0 | h 11 6 |
| 7 | h 13 4 | 8 | 8 | 0 | h 11 6 |
| 8 | m 12 6 | 8 | 7 | 1 | h 11 6 |
| 9 | h 13 4 | 8 | 8 | 0 | h 11 6 |

Now the BLUE player has placed a wall right in front of the RED player, which is trying to bypass it.

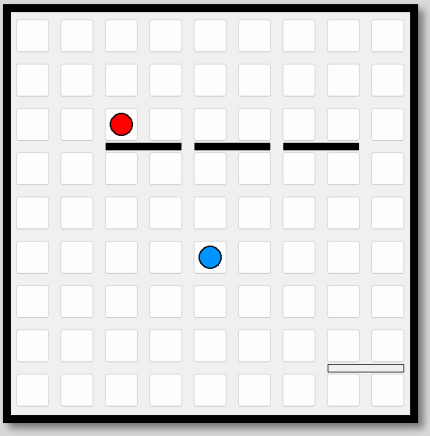


**minimax tree output**

- RED_turn
  - RED_move: m 4 4
    - BLUE_move: m 10 8
      - RED_move: m 4 2
        - BLUE_move: m 8 8
        - No wall
      - RED_wall: h 5 12
        - BLUE_move: m 8 8
        - BLUE_wall: h 13 8
    - No wall
      - No move
        - No move
        - No wall
      - No wall
        - No move
        - No wall
  - RED_wall: h 5 12
    - BLUE_move: m 10 8
      - RED_move: m 4 4
        - BLUE_move: m 8 8
        - BLUE_wall: h 13 8
      - No wall
        - No move
        - No wall
    - BLUE_wall: h 13 8
      - RED_move: m 4 4
        - BLUE_move: m 10 8
        - BLUE_wall: h 13 4
      - RED_wall: h 5 0
        - BLUE_move: m 10 8
        - BLUE_wall: h 13 4

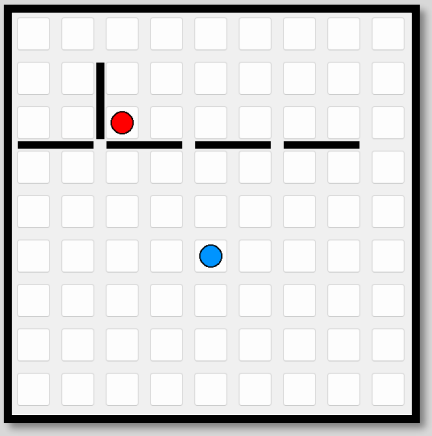
| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 8 8 | 6 | 6 | 0 | m 4 4 |
| 2 | m 8 8 | 7 | 7 | 0 | m 4 4 |
| 3 | h 13 8 | 7 | 8 | -1 | m 4 4 |
| 4 | m 8 8 | 7 | 7 | 0 | h 5 12 |
| 5 | h 13 8 | 7 | 8 | -1 | h 5 12 |
| 6 | m 10 8 | 7 | 8 | -1 | h 5 12 |
| 7 | h 13 4 | 7 | 9 | -2 | h 5 12 |
| 8 | m 10 8 | 11 | 9 | 2 | h 5 12 |
| 9 | h 13 4 | 11 | 10 | 1 | h 5 12 |

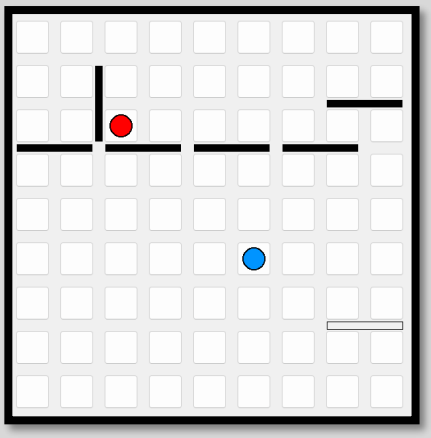
The BLUE player puts another wall at [5,4]. The following sequence of moves: RED-> wall[5,12], BLUE-> wall[13,8], RED-> move[4,4], BLUE-> wall[13,4] lead to the choice of placing a wall at [5,12]. This move will increase BLUE player distance from 7 to 8.

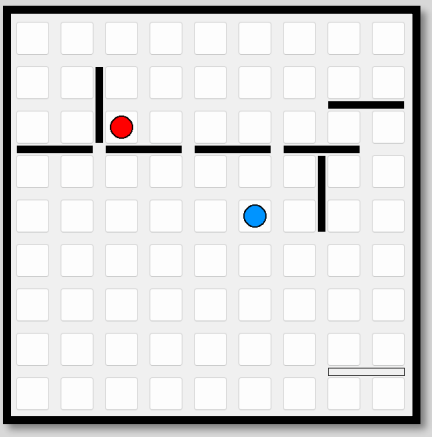



| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 6 8 | 6 | 5 | 1 | m 4 4 |
| 2 | h 13 8 | 6 | 6 | 0 | m 4 4 |
| 3 | m 8 8 | 6 | 6 | 0 | m 4 4 |
| 4 | h 13 4 | 6 | 7 | -1 | m 4 4 |
| 5 | m 8 8 | 12 | 8 | 4 | m 4 4 |
| 6 | h 13 4 | 12 | 9 | 3 | m 4 4 |
| 7 | m 6 8 | 10 | 7 | 3 | h 5 0 |
| 8 | m 8 8 | 10 | 8 | 2 | h 5 0 |
| 9 | h 13 4 | 10 | 9 | 1 | h 5 0 |
| 10 | m 8 8 | 11 | 10 | 1 | h 5 0 |
| 11 | h 13 4 | 11 | 11 | 0 | h 5 0 |

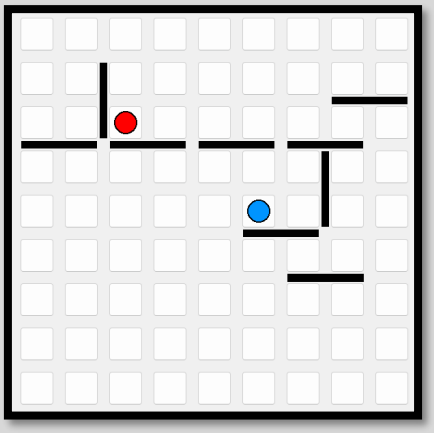
BLUE player will make a move forward again while RED player will also try to get closer to the goal.







BLUE player has placed a vertical wall near RED player, which has also placed a wall at [5,0]. Then BLUE player starts rushing to the end, while RED player will try to make the path as long as possible.



**minimax tree output**

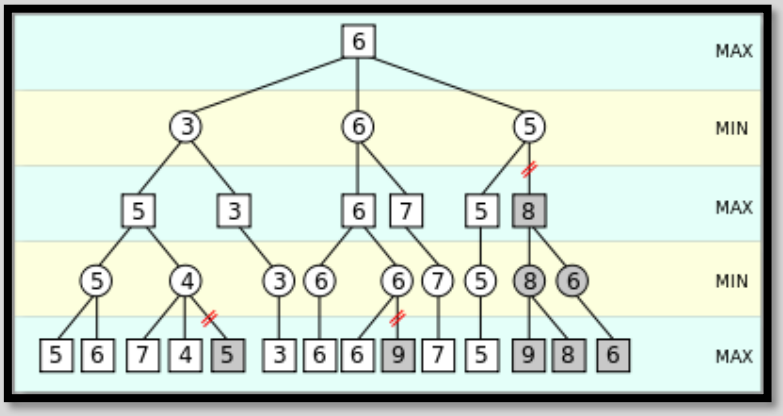
- RED_turn
  - RED_move: m 4 6
    - BLUE_move: m 10 10
      - RED_move: m 4 8
        - BLUE_move: m 10 12
        - No wall
      - RED_wall: v 10 11
        - BLUE_move: m 8 10
        - BLUE_wall: h 13 8
    - No wall
      - No move
        - No move
        - No wall
      - No wall
        - No move
        - No wall
  - RED_wall: h 9 10
    - BLUE_move: m 8 8
      - RED_move: m 4 6
        - BLUE_move: m 10 8
        - BLUE_wall: h 9 6
      - No wall
        - No move
        - No wall
    - BLUE_wall: h 9 6
      - RED_move: m 4 6
        - BLUE_move: m 8 8
        - BLUE_wall: h 9 2
      - RED_wall: v 10 11
        - BLUE_move: m 8 8
        - BLUE_wall: h 9 2

| | final moves | RED dist. | BLUE dist. | diff | root move |
|---|---|---|---|---|---|
| 1 | m 10 12 | 10 | 9 | 1 | m 4 6 |
| 2 | m 8 10 | 11 | 11 | 0 | m 4 6 |
| 3 | h 13 8 | 11 | 12 | -1 | m 4 6 |
| 4 | m 10 8 | 11 | 11 | 0 | h 9 10 |
| 5 | h 9 6 | 11 | 16 | -5 | h 9 10 |
| 6 | m 8 8 | 11 | 16 | -5 | h 9 10 |
| 7 | h 9 2 | 11 | 21 | -10 | h 9 10 |
| 8 | m 8 8 | 12 | 20 | -8 | h 9 10 |
| 9 | h 9 2 | 12 | 25 | -13 | h 9 10 |

If we are not careful, RED player will always find a way to increase the distance to the end and game will result in a loss for BLUE player.

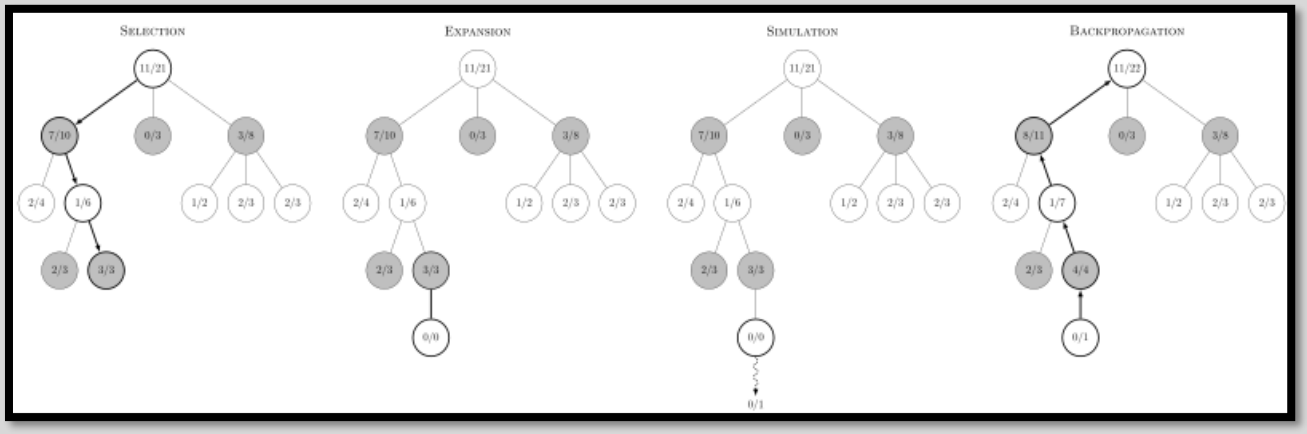
## 3.4. Alternatives

### Minimax with alpha-beta pruning




This is an updated version of the minimax algorithm. When searching for all possible combinations of moves, consider only those that have a best (or worst for the opponent) solution and discard the worst (or best for the opponent). In this way, search times can be significantly reduced, making it possible for example to perform a more in-depth search.

### Monte Carlo tree search (MCTS)




This is another type of tree search algorithm. The idea is to simulate different matches (*playouts*) played from an initial state to the end of the game and then go back to the beginning to identify the sequence of moves with a high probability to achieve a certain goal (victory or defeat). Compared to the minimax algorithm, this algorithm does not need to evaluate the quality of a move, because it only has to check if a final state is reached and therefore it can be easily applied to many types of board games, however it requires a lot of memory and computation to determine a single move.

# 4. Conclusion

The game has been tested several times and the algorithm seems to work efficiently enough to give a minimum of challenge even for slightly more experienced players. This is generally true only when using the shortest path strategy as the main strategy, because the AI only focuses on this one. Other strategies are not applied.
Since there aren't random elements in the game, the algorithm will became repetitive, so after a few plays the game becomes boring and predictable and the AI can be easily fooled. The project focuses only on the operational part of the artificial intelligence, it is not intended to be a complete game of Quoridor.

## Sources:

Project repository:
https://github.com/crimvael/Quoridor.git
QT framework:
https://www.qt.io/
Quoridor Rules:
https://www.ultraboardgames.com/quoridor/game-rules.php
Minimax algorithm:
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
https://en.wikipedia.org/wiki/Minimax
Minimax alpha-beta pruning algorithm:
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
Monte Carlo Tree search algorithm:
https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
BFS algorithm:
https://www.techiedelight.com/breadth-first-search/
Dijkstra's algorithm:
https://daemianmack.org/posts/2019/12/mazes-for-programmers-dijkstras-algorithm.html