

Jogo da Velha em Quint

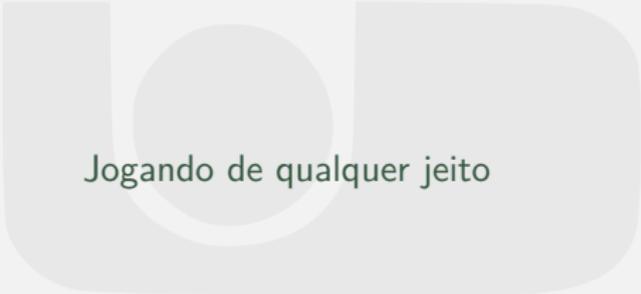
Aula para disciplina de Métodos Formais

Gabriela Moreira

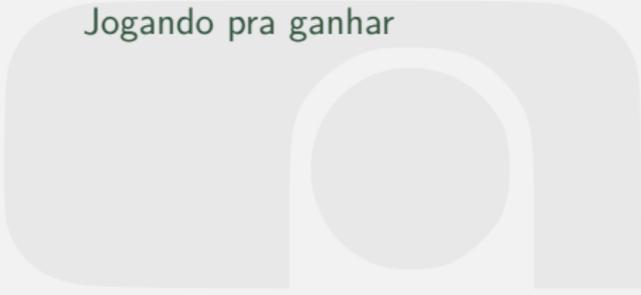
Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

09 de setembro de 2024

Conteúdo

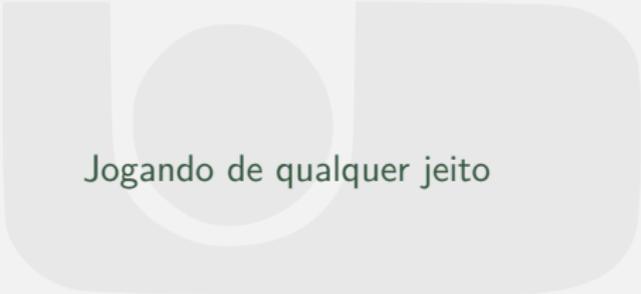


Jogando de qualquer jeito

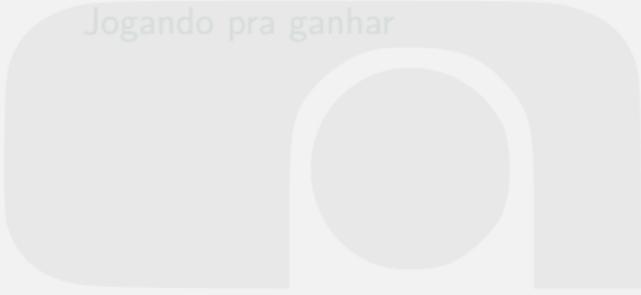


Jogando pra ganhar

Outline



Jogando de qualquer jeito



Jogando pra ganhar

Jogo da Velha

Todos conhecem jogo da velha?

Jogo da Velha

Todos conhecem jogo da velha?

PS: a partir de hoje, não vou mais traduzir as especificações para português

- Acho legal usar português no início para que fique claro o que são **keywords** (sempre em inglês) e o que podemos escolher o nome (nesses casos, em português)
- A partir daqui, vou usar os exemplos originais, em inglês.

Tipos e variáveis

Definimos os seguintes tipos

```
1 type Player = X | O
2 type Square = Occupied(Player) | Empty
```

E as seguintes variáveis

```
1 /// A 3x3 tic-tac-toe board
2 var board: int -> int -> Square
3
4 /// Who goes next
5 var nextTurn: Player
```

Definições sobre coordenadas

```
1 pure val boardCoordinates = tuples(1.to(3), 1.to(3))
2
3 def square(coordinate: (int, int)): Square =
4     board.get(coordinate._1).get(coordinate._2)
5
6 def hasPlayer(coordinate, player) =
7     match square(coordinate) {
8         | Empty          => false
9         | Occupied(p) => player == p
10    }
11
12 def isEmpty(coordinate) =
13     match square(coordinate) {
14         | Empty => true
15         | _    => false
16    }
```

Definições sobre o tabuleiro

```
1 val boardEmpty = boardCoordinates.forall(isEmpty)
2
3 val boardFull = not(boardCoordinates.exists(isEmpty))
```

Definindo “ganhar” - coordenadas

- Como o tabuleiro é sempre 3x3, é mais fácil listar todas as combinações de coordenadas que levam a uma vitória do que implementar os cálculos.
- Usamos `Set` - não precisamos de ordem nem de repetição, logo não devemos usar `List`.

```
1 pure val winningPatterns = Set(  
2   // Horizontal wins  
3   Set((1,1), (1,2), (1,3)),  
4   Set((2,1), (2,2), (2,3)),  
5   Set((3,1), (3,2), (3,3)),  
6   // Vertical wins  
7   Set((1,1), (2,1), (3,1)),  
8   Set((1,2), (2,2), (3,2)),  
9   Set((1,3), (2,3), (3,3)),  
10  // Diagonal wins  
11  Set((1,1), (2,2), (3,3)),  
12  Set((3,1), (2,2), (1,3))  
13 )
```

Definindo “ganhar” - operador won

Usamos as definições para `winningPatterns` e `hasPlayer` para determinar se um jogador venceu.

```
1 def won(player) = winningPatterns.exists(pattern =>
2   pattern.forall(coordinate => hasPlayer(coordinate,
3     player)))
```

Definindo “ganhar” - operador won

Usamos as definições para `winningPatterns` e `hasPlayer` para determinar se um jogador venceu.

```
1 def won(player) = winningPatterns.exists(pattern =>
2   pattern.forall(coordinate => hasPlayer(coordinate,
3     player)))
```

Com essa definição e `boardFull`, podemos determinar se um jogo já acabou.

```
1 val gameOver = won(X) or won(O) or boardFull
```

- Reparem que o operador `or` pode ser usado na forma infixa (no meio dos argumentos)

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
1 action Move(player, coordinate) = all {  
2   isEmpty(coordinate),  
3   board' = board.setBy(  
4     coordinate._1,  
5     row => row.set(coordinate._2, Occupied(player))  
6   ),  
7 }
```

- Qual é a pré-condição pra essa ação?

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
1 action Move(player, coordinate) = all {  
2   isEmpty(coordinate),  
3   board' = board.setBy(  
4     coordinate._1,  
5     row => row.set(coordinate._2, Occupied(player))  
6   ),  
7 }
```

- Qual é a pré-condição pra essa ação?
 - A pré-condição para essa ação é que a coordenada esteja vazia

Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
1 action Move(player, coordinate) = all {  
2   isEmpty(coordinate),  
3   board' = board.setBy(  
4     coordinate._1,  
5     row => row.set(coordinate._2, Occupied(player))  
6   ),  
7 }
```

- Qual é a pré-condição pra essa ação?
 - A pré-condição para essa ação é que a coordenada esteja vazia
- `setBy` é bem útil pra atualizar mapas aninhados (como nesse caso, `int -> int -> Square`)

Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 action MoveToEmpty(player) = all {  
2   not(gameOver),  
3   nondet coordinate = boardCoordinates.filter(isEmpty)  
4     .oneOf()  
5   Move(player, coordinate)  
6 }
```

- Qual é a pré-condição pra essa ação?

Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 action MoveToEmpty(player) = all {  
2   not(gameOver),  
3   nondet coordinate = boardCoordinates.filter(isEmpty)  
4     .oneOf()  
5   Move(player, coordinate)  
6 }
```

- Qual é a pré-condição pra essa ação?
 - A pré-condição para essa ação é que o jogo ainda não tenha acabado

Ações - MoveO e MoveX

- Por enquanto, as ações `MoveO` e `MoveX` são bem parecidas porque ambos jogam “de qualquer jeito”. Não vamos parametrizar elas porque depois vamos mudar somente o comportamento de `X`.

```
1 action MoveO = all {  
2   nextTurn == O,  
3   MoveToEmpty(O),  
4   nextTurn' = X,  
5 }
```

```
1 action MoveX = all {  
2   nextTurn == X,  
3   MoveToEmpty(X),  
4   nextTurn' = O,  
5 }
```

- Qual é a pré-condição pra essas ações?

Ações - MoveO e MoveX

- Por enquanto, as ações `MoveO` e `MoveX` são bem parecidas porque ambos jogam “de qualquer jeito”. Não vamos parametrizar elas porque depois vamos mudar somente o comportamento de `X`.

```
1 action MoveO = all {  
2   nextTurn == O,  
3   MoveToEmpty(O),  
4   nextTurn' = X,  
5 }
```

```
1 action MoveX = all {  
2   nextTurn == X,  
3   MoveToEmpty(X),  
4   nextTurn' = O,  
5 }
```

- Qual é a pré-condição pra essas ações?
 - Para ambas, a pré-condição é que seja o turno do jogador a fazer a jogada
 - Implicitamente, também temos a pré-condição de `MoveToEmpty` empregada nessa ação

Estado inicial

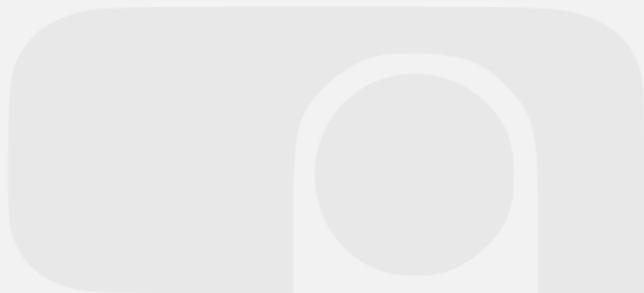
```
1 action init = all {  
2   // X always goes first  
3   nextTurn' = X,  
4   // Every space in the board starts blank  
5   board' = 1.to(3).mapBy(_ => 1.to(3).mapBy(_ => Empty  
6   )),  
6 }
```

Transições

```
1 action step = any {
2   MoveX,
3   MoveO,
4   // If the game is over, we don't need to do anything
5   all { gameOver, board' = board, nextTurn' = nextTurn
6     },
6 }
```

Rodando jogos aleatórios com o simulador

```
1 quint run tictactoe.qnt --max-samples=1
```



Rodando jogos aleatórios com o simulador

```
1 quint run tictactoe.qnt --max-samples=1
```

```
1 ...
2
3 [State 20]
4 {
5   board:
6     Map(
7       1 -> Map(1 -> Occupied(0), 2 -> Occupied(X), 3
8         -> Occupied(X)),
9       2 -> Map(1 -> Occupied(X), 2 -> Occupied(0), 3
10        -> Occupied(0)),
11      3 -> Map(1 -> Occupied(X), 2 -> Occupied(0), 3
12        -> Occupied(X))
13    ),
14   nextTurn: 0
15 }
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
1 val stalemate = boardFull and not(won(X)) and not(won(O))
2
3 val NotStalemate = not(stalemate)
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
1 val stalemate = boardFull and not(won(X)) and not(won(O))
2
3 val NotStalemate = not(stalemate)
```

Essa invariante é fácil de quebrar, podemos usar o simulador ao invés do *model checker* tranquilamente:

```
1 quint run tictactoe.qnt --invariant=NotStalemate
```

Usando uma invariante para procurar jogos que “dão velha”

“Dar velha”, ou *stalemate*, quer dizer que o tabuleiro está cheio e ninguém ganhou. É um empate.

```
1 val stalemate = boardFull and not(won(X)) and not(won(O))
2
3 val NotStalemate = not(stalemate)
```

Essa invariante é fácil de quebrar, podemos usar o simulador ao invés do *model checker* tranquilamente:

```
1 quint run tictactoe.qnt --invariant=NotStalemate
```

Mas podemos usar o *model checker* também! Ele vai demorar mais, porque faz BFS e vai levar um tempo para chegar em jogos com 9 jogadas feitas, que são necessárias para um tabuleiro completo.

```
1 quint verify tictactoe.qnt --invariant=NotStalemate
```

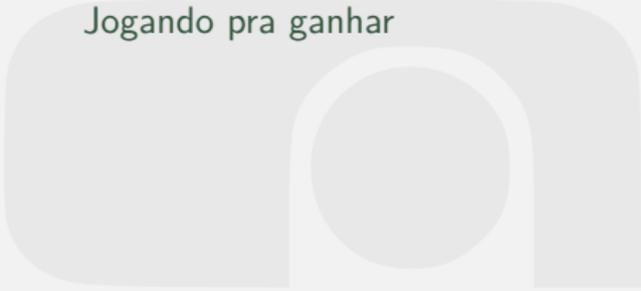
Contraexemplo

```
1 [State 9]
2 {
3   board:
4     Map(
5       1 -> Map(1 -> Occupied(0), 2 -> Occupied(X), 3
6         -> Occupied(0)),
7       2 -> Map(1 -> Occupied(0), 2 -> Occupied(X), 3
8         -> Occupied(X)),
9       3 -> Map(1 -> Occupied(X), 2 -> Occupied(0), 3
10        -> Occupied(X))
11     ),
12   nextTurn: 0
13 }
```

Outline



Jogando de qualquer jeito



Jogando pra ganhar

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que “sempre dá velha”
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
 - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - “sempre dá velha”

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que “sempre dá velha”
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
 - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - “sempre dá velha”

Estratégia:

- A primeira jogada é sempre nos cantos
- As outras jogadas fazem a primeira jogada possível nessa lista de prioridade:
 - Ganhar
 - Bloquear
 - Jogar no centro
 - Preparar uma vitória (preenchendo 2 de 3 quadrados numa fila/coluna/diagonal)
 - Jogada qualquer

Jogando pra ganhar

- Jogo da velha é um jogo bem simples e fácil
- Ainda quando crianças, enjoamos do jogo, porque percebemos que “sempre dá velha”
- Hipótese: Se um jogador seguir uma certa estratégia, ele nunca perde.
 - Consequência: Se os dois jogadores seguirem essa estratégia, nenhum dos dois perde - “sempre dá velha”

Estratégia:

- A primeira jogada é sempre nos cantos
- As outras jogadas fazem a primeira jogada possível nessa lista de prioridade:
 - Ganhar
 - Bloquear
 - Jogar no centro
 - Preparar uma vitória (preenchendo 2 de 3 quadrados numa fila/coluna/diagonal)
 - Jogada qualquer

Vamos implementar essa estratégia para o jogador X, enquanto o jogador O continua jogando “de qualquer jeito”.

Começando com os cantos

```
1 pure val corners = Set(  
2   (1,1),  
3   (3,1),  
4   (1,3),  
5   (3,3)  
6 )  
7  
8 action StartInCorner =  
9   nondet corner = oneOf(corners)  
10  Move(X, corner)
```

Condições para as jogadas

Precisamos definir as condições que determinam se cada uma das jogadas na lista de prioridade pode ser feita.

- Ganhar
- Bloquear
- Jogar no centro
- Preparar uma vitória

```
val canWin = winningPatterns.exists(canWinWithPattern)
val canBlock = winningPatterns.exists(canBlockWithPattern)
val canTakeCenter = isEmpty((2,2))
val canSetupWin = winningPatterns.exists(canSetupWinWithPattern)
```

(`canWinWithPattern`, `canBlockWithPattern` e `canSetupWinWithPattern` a seguir)

Condições para as jogadas - definições auxiliares

Dado um *winning pattern*, podemos **ganhar** com aquele *pattern* sse duas das coordenadas tiverem **X** e a outra estiver vazia. Lembrando que a ordem não importa.

```
1 def canWinWithPattern(pattern) = and {  
2   pattern.filter(coordinate => coordinate.hasPlayer(X)  
3     ).size() == 2,  
4   pattern.filter(coordinate => coordinate.isEmpty()).  
5     size() == 1,  
6 }  
7 }
```

Condições para as jogadas - definições auxiliares

Dado um *winning pattern*, podemos **ganhar** com aquele *pattern* sse duas das coordenadas tiverem **X** e a outra estiver vazia. Lembrando que a ordem não importa.

```
1 def canWinWithPattern(pattern) = and {  
2   pattern.filter(coordinate => coordinate.hasPlayer(X)  
3     ).size() == 2,  
4   pattern.filter(coordinate => coordinate.isEmpty()).  
5     size() == 1,  
6 }
```

Dado um *winning pattern*, podemos **bloquear** com aquele *pattern* sse duas das coordenadas tiverem **O** e a outra estiver vazia.

```
1 def canBlockWithPattern(pattern) = and {  
2   pattern.filter(coordinate => coordinate.hasPlayer(O)  
3     ).size() == 2,  
4   pattern.filter(coordinate => coordinate.isEmpty()).  
5     size() == 1,  
6 }
```

Condições para as jogadas - definições auxiliares II

Dado um *winning pattern*, podemos **preparar uma vitória** com aquele *pattern* sse uma das coordenadas tiver **X** e as outras duas estiverem vazias.

```
1 def canSetupWinWithPattern(pattern) = and {  
2   pattern.filter(coordinate => coordinate.hasPlayer(X)  
3     ).size() == 1,  
4   pattern.filter(coordinate => coordinate.isEmpty()).  
5     size() == 2,  
6 }  
7 }
```

Ações - Win

```
1 action Win = all {  
2   canWin,  
3   nondet pattern = winningPatterns.filter(  
4     canWinWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```

- Qual é a pré-condição pra essa ação?

Ações - Win

```
1 action Win = all {  
2   canWin,  
3   nondet pattern = winningPatterns.filter(  
4     canWinWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```

- Qual é a pré-condição pra essa ação?
 - `canWin`, lembrando que `canWin` é definido por:

```
1 val canWin = winningPatterns.exists(  
2   canWinWithPattern)
```

Ações - Win

```
1 action Win = all {  
2   canWin,  
3   nondet pattern = winningPatterns.filter(  
4     canWinWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```

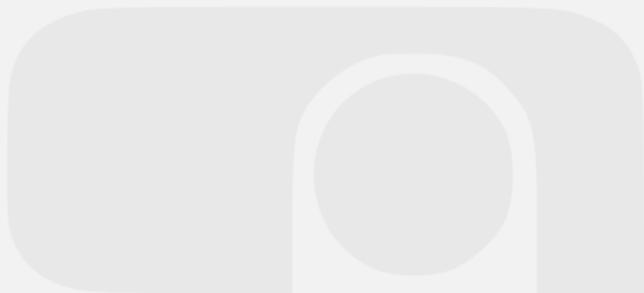
- Qual é a pré-condição pra essa ação?
 - `canWin`, lembrando que `canWin` é definido por:

```
1 val canWin = winningPatterns.exists(  
2   canWinWithPattern)
```

- Isso é importante para garantir que nunca estamos chamando `oneOf` em um set vazio.

Ações - Block

```
1 action Block = all {  
2   canBlock,  
3   nondet pattern = winningPatterns.filter(  
4     canBlockWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```



Ações - Block

```
1 action Block = all {  
2   canBlock,  
3   nondet pattern = winningPatterns.filter(  
4     canBlockWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```

Observem o uso de `oneOf` para selecionar a coordenada aqui. Nesses casos (tanto `Win` quanto `Block`), essa seleção é **determinística**, porque sabemos que sempre haverá uma única coordenada vazia nesses *patterns*. Contudo, o Quint não sabe disso.

Ações - Block

```
1 action Block = all {  
2   canBlock,  
3   nondet pattern = winningPatterns.filter(  
4     canBlockWithPattern).oneOf()  
5   nondet coordinate = pattern.filter(isEmpty).oneOf()  
6   Move(X, coordinate),  
7 }
```

Observem o uso de `oneOf` para selecionar a coordenada aqui. Nesses casos (tanto `Win` quanto `Block`), essa seleção é **determinística**, porque sabemos que sempre haverá uma única coordenada vazia nesses *patterns*. Contudo, o Quint não sabe disso.

- Não existe algo como “pegar o primeiro elemento do set” - porque sets não são ordenados!

Ações - TakeCenter e SetupWin

```
1 action TakeCenter = Move(X, (2, 2))
2
3 action SetupWin = all {
4     nondet pattern = winningPatterns.filter(
5         canSetupWinWithPattern).oneOf()
6     nondet coordinate = pattern.filter(isEmpty).oneOf()
7     Move(X, coordinate),
8 }
```

Ações - alterando MoveX

Temos todas as ações para a estratégia definidas, agora basta definir um novo **MoveX** que chama essas ações conforme a prioridade estabelecida.

Ações - alterando MoveX

Temos todas as ações para a estratégia definidas, agora basta definir um novo **MoveX** que chama essas ações conforme a prioridade estabelecida.

```
1 action MoveX = all {  
2     nextTurn == X,  
3     if (boardEmpty) StartInCorner else  
4     if (canWin) Win else  
5     if (canBlock) Block else  
6     if (canTakeCenter) TakeCenter else  
7     if (canSetupWin) SetupWin else  
8     MoveToEmpty(X),  
9     nextTurn' = 0,  
10 }
```

Invariantes

Com isso, temos nosso modelo. Agora, vamos definir algumas invariantes para o uso dessa estratégia.

```
1  /// X has not won. This does not hold, as X wins
   most of the times.
2  val XHasNotWon = not(won(X))
3
4  /// O has not won. This should hold, as O can only
   achieve a draw.
5  val OHasNotWon = not(won(O))
```

Fórmulas temporais

```
1 /// This is not always true, as if 0 picks the right
   moves, the game will
2 /// result in a stalemate.
3 temporal XMustEventuallyWin = eventually(won(X))
```

- Infelizmente, a implementação de propriedades temporais no Apache ainda é bem rudimentar.
- Podemos traduzir Quint pra TLA+ e usar o TLC para checar essa propriedade
 - Esse processo ainda tem alguns problemas, então vamos deixar quieto por enquanto
- O simulador não suporta fórmulas temporais
 - Poderia suportar com aquela implementação que fizemos em C++/Haskell na disciplina
- Vamos ver essa mesma especificação em TLA+, e aí podemos explorar melhor as propriedades temporais

Tarefa de casa

Tarefa para a próxima aula: ler o blogpost

<https://elliotswart.github.io/pragmaticformalmodeling/>

- Serve como uma revisão de alguns conteúdos da matéria até agora
- Explica a modelagem do jogo da velha em TLA+, que veremos na próxima aula
- Também conta como referência pra essa aula :)

Jogo da Velha em Quint

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

09 de setembro de 2024