Алгоритмы Minimax и Alpha Beta Pruning на примере крестиков-ноликов

В основе всех самых современных крутых шахматных движков лежат два алгоритма: Minimax и Alpha Beta Pruning. Эти алгоритмы придуманы еще в середине 20-го века, но как ни странно, ничего лучшего с тех пор не придумали. По сути все, что делают шахматные движки - это бесконечно гоняют Minimax и Alpha Beta Pruning (разумеется, снабженные массой других алгоритмов, которые делают работу этих двух быстрее и эффективнее). Так делают даже движки, основанные на нейросетях!

Minimax и Alpha Beta Pruning недостаточно, чтобы движки обыгрывали людей одной левой, но с их помощью можно сделать многое.

Чтобы понять как эти алгоритмы работают, я предлагаю их реализовать на очень простой игре: крестики-нолики на поле 3х3.

Эта игра не так проста как кажется: в ней есть 5477 уникальных позиций. Но для компьютера это ничто - поэтому с помощью одного только алгоритма Minimax можно добиться от компьютера идеальной игры в крестики-нолики.

(а вот если мы увеличим размер игрового поля хотя бы до 4х4, лафа заканчивается: счет уникальных позиций уже пойдет, скорее всего, на миллионы)

Начинаем кодить.

[code]board\_side\_length = 3

board\_length = board\_side\_length \* board\_side\_length

[/code]

Я выделяю размер стороны игрового поля в отдельную переменную, потому что так проще будет расширять игровое поле когда мы этого захотим. Изменили одну переменную - и мы уже играем не на поле 3х3, а на поле 4х4, 5х5 или сколько нам заблагорассудится.

Создаем игровое поле.

[code]board = [0] \* board\_length[/code]

Каждая клеточка в этом поле расшифровывается так:

0 - пустое поле

1 - крестик

-1 - нолик

[code]whose\_turn = -1[/code]

Эту переменную можно называть по-разному: piece, symbol, почему-то часто встречается player (хотя на мой взгляд, это слишком неконкретное обозначение). Но мне кажется, что мой вариант исчерпывающе показывает, что эта переменная делает

Начинаем игру.

[code] empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

print("Draw!")

break

[/code]

Проверяем все свободные клетки - это наши потенциальные ходы. Если потенциальных ходов нет, игра прекращается и объявляется ничья.

[code] best\_move = -1

if whose\_turn == -1:

while True:

row = int(input("Row: ")) - 1

column = int(input("Column: ")) - 1

best\_move = row \* board\_side\_length + column

if board [ best\_move ] == 0:

break

[/code]

Если очередь ходить ноликов (то есть, нас), то мы вводим ход вручную. Вообще-то погромисты должны уметь считать и от 0, но я сделал так, что счет идет от 1. Например, чтобы поставить нолик в центральную клетку, нужно ввести "2" и "2".

[code] else:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(0, board, False, -whose\_turn)

if score > best\_score:

best\_score = score

best\_move = square\_index

board[square\_index] = 0

[/code]

А вот если ходят крестики (то есть, наш противник - компьютер), то ход будет выбирать алгоритм Minimax.

Смысл простой: мы перебираем все возможные наши (то есть, компьютера) ходы и для каждого из них спрашиваем Minimax: а как ты оценишь этот ход? И тот ход, который Minimax оценит наиболее высоко, компьютер и предпочтет.

Но мы еще не написали сам код Minimax. Давайте исправим эту оплошность.

[code] if player\_has\_won(-whose\_turn, board, board\_side\_length):

if maximizing\_player:

return -1000 + depth

else:

return 1000 - depth

[/code]

Что тут происходит? depth, maximizing\_player, тысяча какая-то… Еще и функцию player\_has\_won вызываем, которой нет в нашем коде.

Эта функция является главной во всей программе крестиков-ноликов. Она проверяет, выиграл ли игрок или нет. Я бы рекомендовал ее просто скопипастить и использовать. Можете, на досуге и изучить: быть может, вы сможете сделать более быстрый вариант или с более красивым кодом.

Но преимущество моего варианта над теми, что я видел в интернете, в том, что он универсален для любого размера поля. Функция будет работать хоть для поля 3 на 3, хоть для поля 1000 на 1000 (другой вопрос, хватит ли компьютерных мощностей, чтобы все это посчитать). Она принимает три параметра: игровое поле, размер одной стороны игрового поля, и для кого мы проверяем выигрыш: для крестиков или для ноликов.

[code]def player\_has\_won(piece, board, board\_side\_length):

for row in range(board\_side\_length):

won = True

for col in range(board\_side\_length):

index = row \* board\_side\_length + col

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

for col in range(board\_side\_length):

won = True

for row in range(board\_side\_length):

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

col = 0

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col += 1

row += 1

if won:

return True

col = board\_side\_length - 1

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col -= 1

row += 1

if won:

return True

return False

[/code]

Итак, эту сугубо техническую (но очень важную) деталь мы закрыли, и теперь мы умеем в любой момент проверять, а не выиграл ли кто часом в нашу игру.

Возвращаемся в Minimax.

Заметили, что для Maximizing Player мы почему-то возвращаем отрицательное число, а для Minimizing Player - положительное. Нетрудно догадаться почему: мы ведь оцениваем результат ПРЕДЫДУЩЕГО хода, когда значение переменной maximizing\_player было противоположным.

Стоп! А что это вообще такое: Maximizin Player, Minimizing Player?!

Алгоритм Minimax делит двух противоборствующих игроков на два лагеря: на того, кто стремится набрать как можно больше очков, и на того, кто стремится набрать как можно меньше очков. Это абсолютная необходимость для того, чтобы алгоритм работал.

Maximizing player - это тот, кто стремится набрать как можно больше очков. Minimizing player - тот, кто стремится набрать как можно меньшеочков. Они постоянно передают друг другу ход.

Абсолютно не обязательно привязывать Maximizing Player к крестикам, а Minimizing Player к ноликам - можно делать и наоборот. Maximizing player и Minimizing player можно оставить как абстрактные сущности, а более высокая абстракция дает больше свободы выбора и больше возможностей для расширения кода.

Но я для простоты явсе же синхронизировал так, что Maximizing Player у нас будут крестики, а Minimizing Player - нолики. Но вполне можно было бы сделать и наоборот.

Теперь перейдем к depth.

depth - очень важная часть Minimax. Она выполняет две функции, и к сожалению, в данной статье мы сможем использовать только одну из них (хотя вторая, на самом деле, гораздо важнее).

depth показывает, сколько ходов вглубь мы рассмотрели. Не вширь ("куда мне крестик поставить: направо или налево?"), а вглубь ("если я поставлю крестик в центр, а он в ответ поставит нолик, в левый верхний угол, а я после этого поставлю крестик в противоположный угол...").

Для чего нам это нужно? Чтобы компьютер не выглядел идиотом.

Представим, что компьютер нашел способ выиграть в три хода, а также способ выиграть в четыре хода. Какой бы вариант он не выбрал, он в любом случае нас обыграет, но если мы увидим, что он не воспользовался кратчайшим способом выиграть, у нас возникнет плохое впечатление от него (дескать, плохо играет товарищ).

Поэтому мы вычитаем из количества набранных очков глубину. Чем выше глубина, тем меньше очков получит игрок за победу или ничью.

Продолжаем писать код Minimax.

[code] empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

if maximizing\_player:

return 0 + depth

else:

return 0 - depth

[/code]

Если ходов нет, значит ничья, и оценка позиции: 0 минус глубина.

[code] best\_move = -1

if maximizing\_player:

best\_score = -2

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(board, not maximizing\_player, whose\_turn)

if score > best\_score:

best\_score = score

board[square\_index] = 0

[/code]

Да это же практически копия того кода, что был в основном теле игры! Мы уже, на самом деле, практически написали код алгоритма Minimax. Поэтому я сейчас приведу код Minimax полностью, а потом словами опишу, как этот алгоритм работает.

[code]def minimax(depth, board, maximizing\_player, whose\_turn):

if player\_has\_won(-whose\_turn, board, board\_side\_length):

return 1000 - depth

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

return 0 - depth

if maximizing\_player:

best\_score = -2

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1, board, not maximizing\_player, -whose\_turn)

if score > best\_score:

best\_score = score

board[square\_index] = 0

return best\_score

else:

best\_score = 2

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1,board, not maximizing\_player, -whose\_turn)

if score < best\_score:

best\_score = score

board[square\_index] = 0

return best\_score

[/code]

Minimax - это симулятор счета вариантов. Как человек считает варианты?

«Если я поставлю крестик в левом верхнем углу, а оппонент поставит нолик в центре, то я потом могу поставить крестик в середине верхнего ряда, и тогда оппоненту ничего не останется кроме как поставить нолик справа сверху, и тогда я обязан буду поставить крестик в левом нижнем углу…».

Minimax делает то же самое. За одним-единственным исключением.

Человек всегда считает только основные ходы. Он интуитивно отбрасывает ходы, которые не считает важными или вероятными и сосредотачивается на основных. А Minimax так не делает. Он работает как брутфорс - смотрит все возможные ходы. А потом смотрит все возможные ответы на каждый из ходов. А потом смотрит все возможные ответы на каждый из ответов. Minimax покрывает всю поляну.

Но он не делает это бесконечно. Он прекращает свою работу и возвращает оценку позиции в двух случаях:

1) Либо когда игра закончилась

2) Либо когда глубина достигла определенного значения (в этой статье такой вариант рассматриваться не будет)

Но мы продолжаем писать код игры. Возвращаемся в тело игры, внутрь того бесконечного цикла While:

[code] print\_board(board, board\_side\_length)

[/code]

Нам надо же знать, как сейчас выглядит игровое поле. Скопипастите код:

[code]def print\_board(board, board\_side\_length):

for i in range(0, board\_side\_length):

print('-------------')

out = '| '

for j in range(0, board\_side\_length):

index = i \* board\_side\_length + j

if board[index] == 1:

token = 'x'

if board[index] == -1:

token = 'o'

if board[index] == 0:

token = ' '

out += token + ' | '

print(out)

print('-------------')

[/code]

…и едем дальше

[code] if player\_has\_won(whose\_turn, board, board\_side\_length):

print(f"{["Naughts", "None", "Crosses"][whose\_turn + 1]} has won!")

Break

[/code]

Поздравляю! Код написан! Можете его запустить, и убедиться, что компьютер играет в крестики-нолики за крестики идеально.

[code]def player\_has\_won(piece, board, board\_side\_length):

for row in range(board\_side\_length):

won = True

for col in range(board\_side\_length):

index = row \* board\_side\_length + col

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

for col in range(board\_side\_length):

won = True

for row in range(board\_side\_length):

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

col = 0

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col += 1

row += 1

if won:

return True

col = board\_side\_length - 1

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col -= 1

row += 1

if won:

return True

return False

def print\_board(board, board\_side\_length):

for i in range(0, board\_side\_length):

print('-------------')

out = '| '

for j in range(0, board\_side\_length):

index = i \* board\_side\_length + j

if board[index] == 1:

token = 'x'

if board[index] == -1:

token = 'o'

if board[index] == 0:

token = ' '

out += token + ' | '

print(out)

print('-------------')

def minimax(depth, board, maximizing\_player, whose\_turn):

if player\_has\_won(-whose\_turn, board, board\_side\_length):

if maximizing\_player:

return -(1000 - depth)

else:

return 1000 - depth

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

if maximizing\_player:

return 0 + depth

else:

return 0 - depth

if maximizing\_player:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1, board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score > best\_score:

best\_score = score

return best\_score

else:

best\_score = 2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1,board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score < best\_score:

best\_score = score

return best\_score

board\_side\_length = 3

board\_length = board\_side\_length \* board\_side\_length

board = [0] \* board\_length

whose\_turn = -1

while True:

whose\_turn = -whose\_turn

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

print("Draw!")

break

best\_move = -1

if whose\_turn == -1:

while True:

row = int(input("Row: ")) - 1

column = int(input("Column: ")) - 1

best\_move = row \* board\_side\_length + column

if board [ best\_move ] == 0:

break

else:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(0, board, False, -whose\_turn)

board[square\_index] = 0

if score > best\_score:

best\_score = score

best\_move = square\_index

board[best\_move] = whose\_turn

print\_board(board, board\_side\_length)

if player\_has\_won(whose\_turn, board, board\_side\_length):

print(f"{["Naughts", "None", "Crosses"][whose\_turn + 1]} has won!")

break

[/code]

Ну что, мы освоили суть компьютерных движков? Теперь, вооружившись знаниями о Minimax, идем крушить шахматы, шашки или го?

Увы. Не то, что в шахматы - а даже если увеличить поле крестиков-ноликов до 4х4, то компьютер зависнет. Количество вариантов настолько быстро растет, что Minimax будет их обрабатывать безумно долго.

Давайте посчитаем, сколько ходов требуется обработать компьютеру на первом ходу, чтобы посчитать все варианты в наших крестиках-ноликах.

Там, где у нас блок:

[code]board\_side\_length = 3

board\_length = board\_side\_length \* board\_side\_length

[/code]

добавим еще одну переменную

[code]number\_of\_moves\_counted = 0

[/code]

А в начало функции Minimax добавим это:

[code]def minimax(depth, board, maximizing\_player, whose\_turn):

global number\_of\_moves\_counted

number\_of\_moves\_counted += 1

[/code]

а в теле игры сделаем так:

[code] best\_move = square\_index

print(f"Number of moves counted: {number\_of\_moves\_counted}")

[/code]

549945 ходов пришлось обработать компьютеру! Для игры в крестики-нолики! Понимаете, какое расточительство?

К счастью, умные люди придумали алгоритм Alpha Beta Pruning. Это такое дополнение к Minimax, которое позволяет на математическом уровне отсекать рассмотрение тех вариантов, которые даже теоретически не могут улучшить нашу позицию. Один только этот алгоритм может отсечь, скажем, 95% всех ходов как неперспективные, и решить ту же самую задачу радикально быстрее.

Сразу к делу: мы меняем код Minimax и превращаем его в следующее:

[code]def minimax(depth, alpha, beta, board, maximizing\_player, whose\_turn):

global number\_of\_moves\_counted

number\_of\_moves\_counted += 1

if player\_has\_won(-whose\_turn, board, board\_side\_length):

if maximizing\_player:

return -(1000 - depth)

else:

return 1000 - depth

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

if maximizing\_player:

return 0 + depth

else:

return 0 - depth

if maximizing\_player:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1, alpha, beta, board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score > best\_score:

best\_score = score

if score > alpha:

alpha = score

if beta <= alpha:

break

return best\_score

else:

best\_score = 2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1,alpha, beta, board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score < best\_score:

best\_score = score

if score < beta:

beta = score

if beta <= alpha:

break

return best\_score

[/code]

а в теле игры вызываем нашу функцию так:

[code] score = minimax(0, -30000, 30000, board, False, -whose\_turn)

[/code]

Альфа, Бета... Что еще за чертовщина? Совсем с ума сошли?

Если честно, - да. Я считаю эти названия крайне неудачными. Они только запутывают.

На самом деле, Альфа - это количество очков, которые гарантированы для Maximizing Player в данной позиции. Бета - гарантированное количество очков, которые имеет Minimizing Player в данной позиции.

Но не будем забивать себе этим голову - запустим теперь нашу программу.

Всего 34202 ходов потребовалось обработать компьютеру, чтобы полностью просчитать позицию! В 16 (шестнадцать) раз меньше! И это - не предел.

Теперь я объясню как работает алгоритм Alpha Beta Pruning. Я объясню на конкретном примере - потому что сам не очень хорошо пока алгоритм понимаю, а “объяснения", которые я нахожу на Ютубе, с моей точки зрения, просто ужасны. Поэтому - расскажу как могу.

Перейдем к делу. Давайте представим, что игра начинается, и крестики “думают” (с помощью алгоритмов Minimax и Alpha Beta Pruning) следующим образом: «Если я поставлю крестик в левый верхний угол, а соперник поставит нолик в середину верхнего ряда, а я потом поставлю крестик в правый верхний угол, а соперник поставит нолик в середину левого столбца, а я поставлю крестик в центре, а соперник поставит нолик в середине правого столбца…».

-------------

| x | o | x |

-------------

| o | x | o |

-------------

| | | |

-------------

«…то если я поставлю крестик в левом нижнем углу, то выиграю партию и получу за это 994 очка (ибо потратил семь ходов на все мероприятие).

Если же я поставлю крестик не в левом нижнем углу, а в середине нижнего ряда, то соперник поставит нолик в левом нижнем углу, а я в ответ поставлю крестик в правом нижнем углу, выиграю партию и получу за это 992 очка (ибо потратил девять ходов на все мероприятие).

А что будет, если я поставлю крестик в середине нижнего ряда, а соперник поставит нолик не в левом нижнем углу, а в правом? Я этот вариант даже смотреть буду. Ни сам ход ни мои возможные ответы на него.

Почему? Потому что мой соперник уже знает ход, где он гарантированно получит не больше 992 очков, а значит он уже никогда не выберет какой-либо другой ход, при котором очков будет больше. И поэтому я в середину нижнего ряда уже по-любому крестик ставить не буду - потому что этот ход принесет мне не больше 992 очков. Но я-то уже выяснил на предыдущей итерации, что если поставлю крестик левый нижний угол, то получу не меньше 994 очков. Конечно я предпочту этот ход!

Идем дальше:: если я поставлю крестик в правый нижний угол…»

Может показаться, что мы отсекли всего-то пару ходов, и этот эффект незначительный. Это обманчивое впечатление. Alpha Beta Pruning позволяет отсекать не просто отдельные ходы - а целые ветки. Он может отсечь целую цепочку из серии: "У меня есть пять ходов, на них есть 4 ответа, на каждый из которых есть 3 ответа…" - он может в эту цепочку вообще не лезть потому что уже заранее знает, что в ней его позиция точно не улучшится.

Мои тесты показали, что компьютер может идеально играть в крестики-нолики 3x3, просчитав меньше тысячи ходов.

Но от чего это зависит? От порядка ходов. Алгоритм Alpha Beta Pruning очень чувствителен к тому, чтобы в начале списка ходов были самые сильные. Чем сильнее ходы обрабатываются в самом начале, тем большее количество ходов будет отсекаться.

Полный код:

[code]def player\_has\_won(piece, board, board\_side\_length):

for row in range(board\_side\_length):

won = True

for col in range(board\_side\_length):

index = row \* board\_side\_length + col

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

for col in range(board\_side\_length):

won = True

for row in range(board\_side\_length):

if not board[row \* board\_side\_length + col] == piece:

won = False

break

if won:

return True

col = 0

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col += 1

row += 1

if won:

return True

col = board\_side\_length - 1

row = 0

won = True

while row < board\_side\_length:

if not board[row \* board\_side\_length + col] == piece:

won = False

break

col -= 1

row += 1

if won:

return True

return False

def print\_board(board, board\_side\_length):

for i in range(0, board\_side\_length):

print('-------------')

out = '| '

for j in range(0, board\_side\_length):

index = i \* board\_side\_length + j

if board[index] == 1:

token = 'x'

if board[index] == -1:

token = 'o'

if board[index] == 0:

token = ' '

out += token + ' | '

print(out)

print('-------------')

def minimax(depth, alpha, beta, board, maximizing\_player, whose\_turn):

global number\_of\_moves\_counted

number\_of\_moves\_counted += 1

if player\_has\_won(-whose\_turn, board, board\_side\_length):

if maximizing\_player:

return -(1000 - depth)

else:

return 1000 - depth

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

if maximizing\_player:

return 0 + depth

else:

return 0 - depth

if maximizing\_player:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1, alpha, beta, board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score > best\_score:

best\_score = score

if score > alpha:

alpha = score

if beta <= alpha:

break

return best\_score

else:

best\_score = 2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(depth + 1,alpha, beta, board, not maximizing\_player, -whose\_turn)

board[square\_index] = 0

if score < best\_score:

best\_score = score

if score < beta:

beta = score

if beta <= alpha:

break

return best\_score

board\_side\_length = 3

board\_length = board\_side\_length \* board\_side\_length

number\_of\_moves\_counted = 0

board = [0] \* board\_length

whose\_turn = -1

while True:

whose\_turn = -whose\_turn

empty\_squares\_indices = []

for square\_index in range(len(board)):

if board[square\_index] == 0:

empty\_squares\_indices.append(square\_index)

if len(empty\_squares\_indices) == 0:

print("Draw!")

break

best\_move = -1

if whose\_turn == -1:

while True:

row = int(input("Row: ")) - 1

column = int(input("Column: ")) - 1

best\_move = row \* board\_side\_length + column

if board [ best\_move ] == 0:

break

else:

best\_score = -2000

for square\_index in empty\_squares\_indices:

board[square\_index] = whose\_turn

score = minimax(0, -30000, 30000, board, False, -whose\_turn)

board[square\_index] = 0

if score > best\_score:

best\_score = score

best\_move = square\_index

print(f"Number of moves counted: {number\_of\_moves\_counted}")

board[best\_move] = whose\_turn

print\_board(board, board\_side\_length)

if player\_has\_won(whose\_turn, board, board\_side\_length):

print(f"{["Naughts", "None", "Crosses"][whose\_turn + 1]} has won!")

break

[/code]