****

**BAHRIA UNIVERSITY**

**(Karachi Campus)**

*Department of Software Engineering*

**ARTIFICIAL INTELLIGENCE**

**PROJECT REPORT**

**Submitted To:**

**Sir Zia Ul Islam / Sir Rehan Baig**

**BSE 7B**

**Group Member:**

**M Sohail Zakir (51742)**

**Ammara Riaz (51730)**

Contents

[**1.** **Dots And Boxes** 2](#_Toc62041890)

[**2.** **Strategy** 3](#_Toc62041891)

[**3.** **Programmer's Reference** 4](#_Toc62041892)

[**3.1.** **Making a move** 4](#_Toc62041893)

[**3.2.** **Understanding the board** 4](#_Toc62041894)

[**3.3.** **Making a valid move** 4](#_Toc62041895)

[**3.4.** **Helper functions** 5](#_Toc62041896)

[5. Reference: 7](#_Toc62041897)

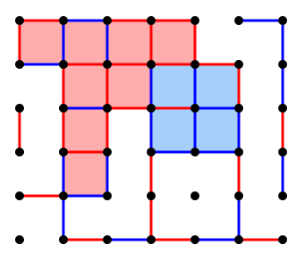
[https://inventwithpython.com/chapter10.html 7](#_Toc62041898)

[https://codereview.stackexchange.com/questions/108738/python-tic-tac-toe-game 7](#_Toc62041899)

[https://dev.to/nestedsoftware/tic-tac-toe-with-the-minimax-algorithm-5988 7](#_Toc62041900)

1. **Dots And Boxes**

Dots And Boxes is played on a rectangular grid of dots. Players alternate drawing a horizontal or a vertical line between adjacent dots - if they complete a box then they score 1 point (per box) and get another turn. The player with the most points once all of the boxes have been completed wins.



1. **Strategy**

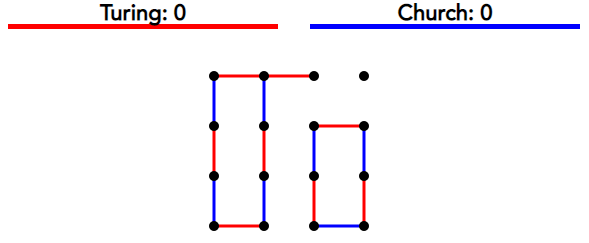
The template code, found in the Online Code Editor, works out what all the valid moves are and picks one at random. This can be improved by finding valid moves which complete a box and picking one of those at random, if any exist. Can you use the doesCompleteSquare() function to implement this?

In some cases, one line can complete two different boxes. By adapting doesCompleteSquare(), or writing a new function, you could prioritise such lines over lines which complete only one box.

Next, you could think about how to guard against your opponent using the same strategy. It might help to start by writing a function which calculates whether, if you play a certain move, your opponent will be able to complete a box on their next turn.

One way to do this is to calculate what the new grid and squares will look like after your move, and then using findLegalMoves() and doesCompleteSquare(), but there is a faster method. Maybe trying out a practice game on pen and paper will be helpful – what type of moves let your opponent complete a square in the next turn?

Advanced strategies involve analysis of “chains” – a collection of adjacent boxes where the exterior lines are filled in, and the interior lines can all be filled in by one player in a sequence of moves, as they get another turn each time they complete a square. For instance, the game below has a chain of size 3 in the leftmost columns, which makes a better sequence of moves than joining a line to make two squares in the rightmost columns. (In fact, in this case the player can complete both chains to score five points, but advanced analysis may look at which chains it is possible to create/complete in a couple of turns in the future.)



1. **Programmer's Reference**
   1. **Making a move**

The calculate\_move() function is the equivalent of your main function. This is where you need to make your changes and from where you will control the game. The function:

1. Receives information about the game in a Python dictionary called 'gamestate'.
2. Returns a game move.
   1. **Understanding the board**

There are two representations of the board in gamestate, one representing the lines that have been played ("grid") and another representing each box ("squares").

* **Coordinate**: a list of two numbers, representing a dot on the grid. [0, 0] is the top left dot. The coordinates represent the x and the y axis, using 0-based indexing, so for instance [2, 3] is the dot on the third column and the fourth row.
* **Grid**: A list of lists, where each list represents a line. It contains two coordinates and an integer, -1, 0 or 1. This integer is -1 if neither player has drawn the line, 0 if you have drawn this line already and 1 if your opponent has drawn this line. For instance, the list [[1, 0], [1, 1], 1] means that your opponent has drawn a vertical line between the dots in the top two rows of the second column.
* **Squares**: A list of lists, where each list represents a box. It contains four lines and an integer, -1, 0 or 1. Each line consists of two coordinates and a Boolean, which is False if neither player has drawn the line and True if either player has drawn the line. The integer at the end of the list is -1 if the box has not been completed (i.e. not all lines have been drawn), 0 if you have completed the box and 1 if your opponent has completed the box.
  1. **Making a valid move**

Every time it is your turn, the game engine will run your calculate\_move() function, which needs to return a valid move. A valid move is a list of two coordinates, representing adjacent dots which do not currently have a line between them.

For example, [[0, 1], [1, 1]] is a valid move (unless it has already been played), because there is a horizontal line between [0, 1] and [1, 1].

* 1. **Helper functions**
* **findLegalMoves(grid)** – Given a list representing the lines on the current board, return a list of all possible valid moves – that is, a list of representations of each line which has not yet been drawn.
* **doesCompleteSquare(move, squares)** – Given a valid move and a list representing the squares on the current board, return a Boolean which is True if the move would complete at least one box (i.e. if the new line forms a square with other lines already on the board) and False otherwise.

1. **Output**

To make the game clearer, I'll run through a sample game. Let's say that

two players, A and B, are playing Dots-And-Boxes on a 3x3 board:

+ + +

+ + +

+ + +

Here's one run through the game:

+--+ +

+ + +

+ + +

Player A chooses to join the first two dots.

+--+ +

|

+ + +

+ + +

Player B chooses to join the mid-top and middle. The game can proceed

like this:

+--+ +

|

+ + + A moves a bit more down.

|

+ + +

+--+ +

|

+ + + B makes it snake around a bit.

|

+ +--+

+--+ +

|

+ + + A connects the lower left and mid left points.

| |

+ +--+

+--+ +

|

+ + + A connects the lower left and mid left points.

| |

+ +--+

+--+ +

|

+ + + B forgets that the game is not Hangman.

| |

+--+--+

+--+ +

|

+--+ + A completes a square. Because of this, it's A's turn

| A| again.

+--+--+

+--+ +

| A|

+--+ + A completes the upper left square. Another free turn.

| A|

+--+--+

+--+--+

| A|

+--+ + A plays the upper right square, finally giving a turn

| A| to B. But it's too late.

+--+--+

+--+--+

| A|

+--+ + B places a futile line.

| A| |

+--+--+

+--+--+

| A|

+--+--+ And A promptly trounces.

| A| A|

+--+--+

+--+--+

| A| A|

+--+--+ A dominates the game.

| A| A|

+--+--+

# Reference:

# <https://inventwithpython.com/chapter10.html>

# <https://codereview.stackexchange.com/questions/108738/python-tic-tac-toe-game>

# <https://dev.to/nestedsoftware/tic-tac-toe-with-the-minimax-algorithm-5988>