# Pi Square

## Getting Started

Before you can start tackling the Pi Square challenge, you need to have your Raspberry Pi up and running with the X-windows graphical user interface loaded (type startx at the command prompt). If you need help getting to this point, take a look at the getting started resources on the VLE.

If you have never programmed before, you might like to work through some of the programming resources on the VLE before going any further. However, feel free to download the template and start looking at the code. You may find that python is quite easy to read and understand.

You will need to download the template source code from the VLE. This is a zip file (template.zip) that needs copying to your Raspberry Pi and uncompressing. If you are able to connect your RPi to the internet then you can download it directly to there. Otherwise, you will need to copy it onto a USB stick and from there copy it onto your Pi.

If you're happy with python and have the template files, you're ready to tackle the challenge so read on…

## The Game

Squares (also called "dots and boxes") is a classic pencil and paper game for two players. The game is played one a grid of dots. A turn consists of drawing a horizontal or vertical line (we'll call this an "edge") between two adjacent dots. If the line completes a square, then the player "captures" the square and is allowed to have another turn. Otherwise play switches to the other player. The game ends when all edges have been drawn. The winner is the player who has captured the most squares.

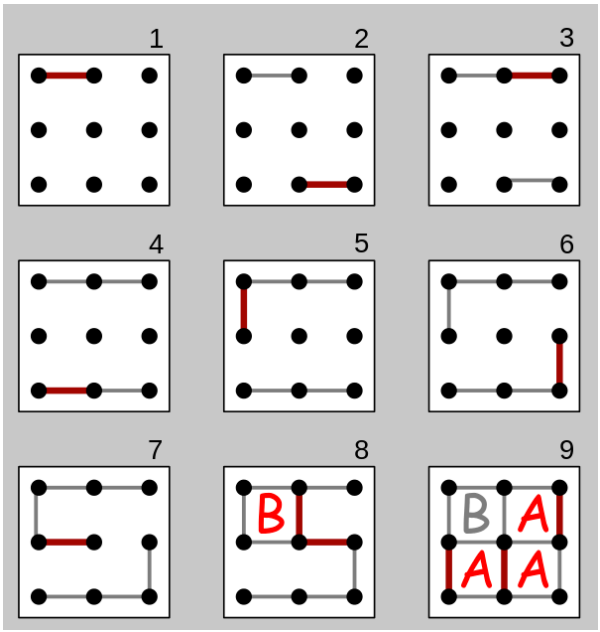


**Figure 1: An example game of squares**

The diagram above shows an example game on a small board consisting of 9 dots and 4 squares. The second player (B) plays the mirror image of the first player's move, hoping to divide the board into two pieces and tie the game. The first player (A) makes a *sacrifice* at move 7; B accepts the sacrifice, getting one box. However, B must now add another line, and connects the centre dot to the centre-right dot, causing the remaining boxes to be joined together in a *chain* as shown at the end of move 8. With A's next move, A gets them all, winning the game 3 squares to 1.

## PI SQUARE

To make things more interesting (and to avoid anyone using code straight off the web), the Pi Square challenge takes place on a non-standard board. This is in the shape of (unsurprisingly) the Greek letter Pi (see Figure 2). For each square you capture, you win a pie. The player with the most pies wins the game. Sadly, the pies are virtual so this is no free lunch. A match is best of three, so you need to win two games to win a match.

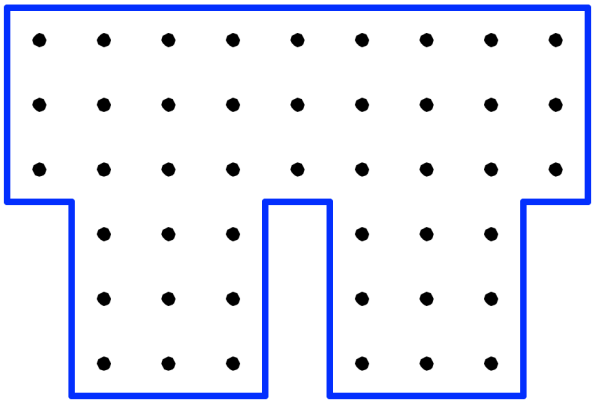

**Figure 2: The Pi Square board before any moves have been played**

The Pi Square board contains 45 dots (we'll call these "vertices"), 72 edges and 28 squares. Hence, a game can be drawn if both players capture 14 pies.

## TEMPLATE

In the file template.zip on the VLE we have provided everything you need to get started.

The `GameServer` directory contains a simple version of the game server. The game server keeps track of the state of the game and communicates with two players. It also has a GUI which displays the current state of the game and the scores. The game server we use on the day of the challenge will also monitor how long your player is taking to make a move.

The `MyPlayer` directory contains your player. This is the only directory that you will need on the day of the challenge. The only file that you should edit is the `player_strategy.py` file in this directory.

The `DummyPlayer` directory contains another player so that you can test your solution against something.

To run a match between MyPlayer and DummyPlayer on your RPi, do as follows. I am assuming that you have uncompressed template.zip in your home directory. Open three command windows (the LXTerminal icon on your RPi). In the first one, start the game server running by typing:

```
cd GameServer
python pisquare_game_server.py
```

This starts the game server which will now wait for two players to connect. Next, in the second command window start your player:

```
cd MyPlayer
python player_clientside.py
```

In the third command window start the dummy player:

```
cd DummyPlayer
python player_clientside.py
```

As soon as the second player connects, the game server will display the GUI (see Figure 3) and start the match. Play continues automatically until someone has won (after either 2 or 3 games).

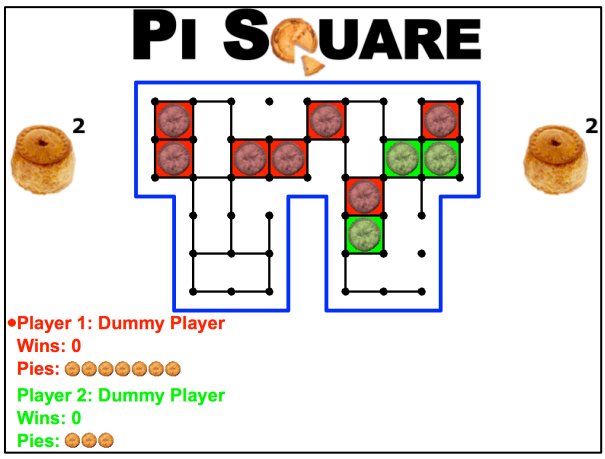


**Figure 3: A Pi Square game in progress displayed in the GUI.**

In the actual competition, only your player will run on your Raspberry Pi. The game server will be accessed across a network and the other player will be another student's entry.

Both players in the template are using the same strategy which is to randomly select from the remaining unplayed edges. This is not a good strategy as it will miss the chance to take squares and may set up the opponent to be able to take squares.

If you don't manage to improve this simple strategy, we encourage you to still enter the competition! It will be good fun and, if you get lucky, you may even win some matches. But hopefully this is just the starting point for your improved solution.

## SUGGESTED APPROACH

Begin by playing squares against a friend or computer (there are many free versions online). Get a feel for the strategic aspects of the game. Then you can begin to implement more advanced strategies.

Inside the file `player_strategy.py` you will see that there are two functions that you may wish to modify. Also, you should change the player name to your own name so that this is displayed on the board.

`chooseMove` is the most important function. The game server sends the current state of the game (a binary string encoding which edges have been played and who has captured which square). The object `board` contains everything you need to know about the board and the current state of the game. It can provide a number of different pieces of information. The template version only accesses a single method: `board.getUnplayedEdges()` which returns a list of edges that have not yet been played. It then randomly selects one of these edges and returns it.

You may also modify the function `newGame` if you want to change your strategy between games. At the moment, this doesn't do anything.

Once you've read and understood those functions, the first step is to improve the strategy. An obvious improvement would be to check whether any unplayed edges would give you a square. The easiest way to do this would be to loop through the unplayed edges, set the state of each one in turn to played:

```
board.setEdgeState(edge,
const.PLAYED)
```

And then check whether this captures any squares:

```
captured=
board.findCaptured(edge)
```

This returns either 0, 1 or 2 squares that have become captured. You should set the edge back to unplayed before testing any others:

```
board.setEdgeState(edge,
const.UNPLAYED)
```

Next, you could think about avoiding edges that will give your opponent a square. Or, when there are multiple edges that will give you a square, which is the best one to take? Beyond this your strategy could become much more sophisticated. Try looking for long chains of squares and either take the longest or avoid giving your opponent long chains. You will discover that advanced players use something called a "double-cross strategy". You could try to implement this.

Or you could use search-based AI techniques. The idea is to search ahead, trying all possible moves and assuming your opponent will select the best move for them. This is called minimax search. This gets complicated and difficult to do inside 5 seconds on a Raspberry Pi! You might want to abandon the search if time is running out.

## RULES

Your program must run on a standalone Raspberry Pi. We will provide you with a keyboard, monitor and mouse on the day of the competition.

Your program is expected to make a move within 5 seconds of being prompted to do so. Any entry that is taking consistently more time than this may be disqualified. The game server will keep track of average move times during gameplay.

The competition will be a knock-out format. The draw will be done randomly and each match consists of best of three games (i.e. you need to win two games to beat your opponent). In the event of a draw, the winner will be the player with the shorter average move time. If you make an invalid move (select an occupied edge), you lose the current game. The grand final will be played as a best of 5 games match. Members of staff will act as referees on the day and their decisions in all cases of doubt are final.

## THE GORY DETAILS

You only need to read this section if you want to know more about the board object and how it is stored. This may be useful for finding chains.

Vertices, edges and squares are referred to by index numbers as shown in Figure 4. Vertices are numbered 0 to 44, left to right, top to bottom. Edges are numbered 0 to 71, also left to right, top to bottom. Squares are numbered by the vertex in their top left hand corner (so, for example, there is no square 8). The board object provides a number of structures relating vertices, edges and squares:

`board.Edges[i]` – Returns the pair of vertices at either end of edge i, e.g. `board.Edges[8]` returns (0,9).

`board.Squares[i]` – Returns the four edges surrounding square i, e.g. `board.Squares[19]` returns (35,42,43,48).

`board.Edge2Squares[i]` – Returns the 1 or 2 squares adjacent to edge i. e.g. `board.Edge2Squares[56]` returns (30,31).

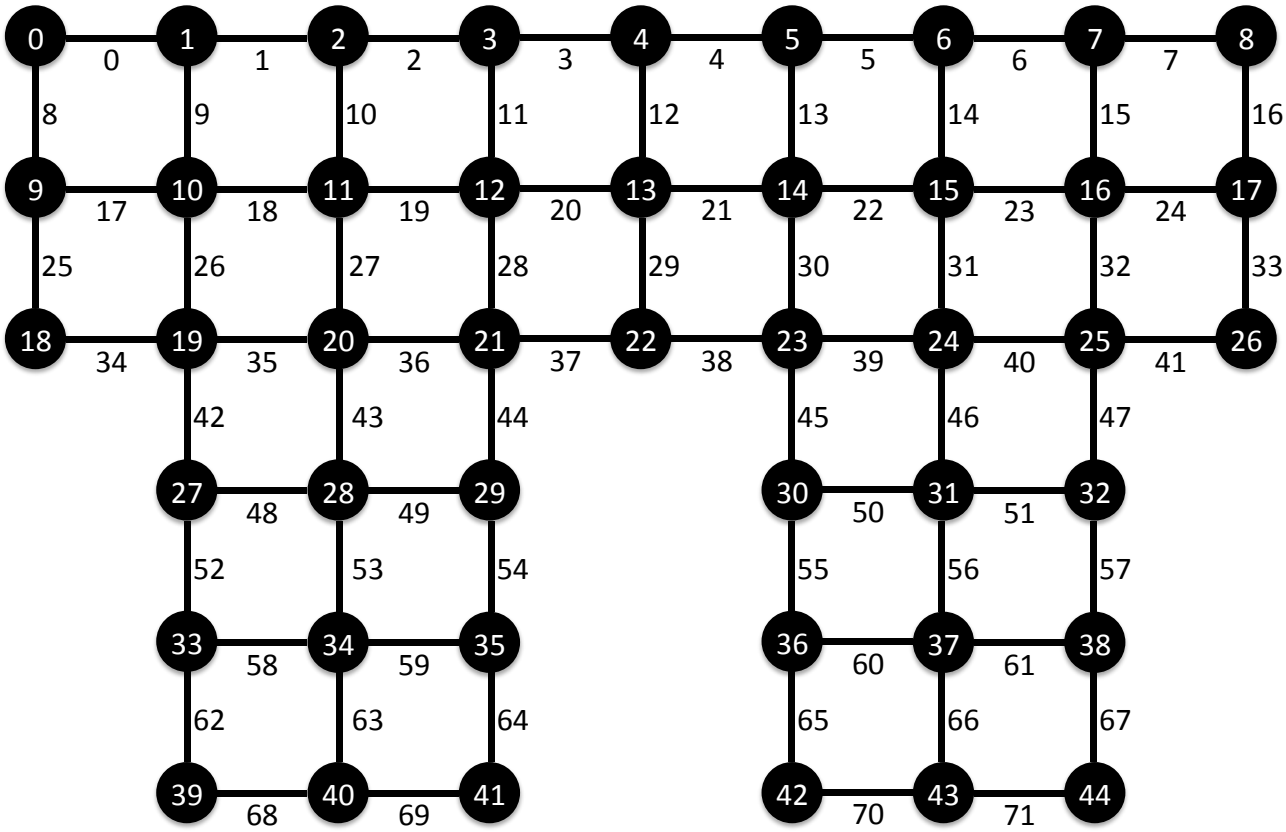


**Figure 4: The vertex and edge numbering system for the Pi Square board.**