Table of Contents

Ultimate TTT Algorithmic Competition

Welcome to the documentation for the competition!

This "book" explains the **Ultimate Tic Tac Toe** game, and the process of writing an algorithm to compete at it.

You can start with the game rules, or - if you already know them - with how to write a player.

The platform to compete in, sample players, implementations... are all Open Source and available in our GitHub organization: https://github.com/socialgorithm

Good Luck!

Preparing your Laptop

These are the minimum requirements to participate in the coding competition:

1. Have a **code editor**. The choice here is basically infinite, so I'll list the most popular and you can choose any.
   - Visual Studio Code *(recommended)*
   - Sublime Text
   - Atom

   > Subjective, I know, and you might be screaming "emacs/vim are SO much better". We know them, and use them (vim personally), but this is targeting people *without* a code editor in their machine already.

2. Install NodeJS >7
3. Install the competition client:

   ```
   npm install -g @socialgorithm/uabc
   ```

   On some computers it might complain that it needs extra permissions (Mac & Linux, depending on your conf) so try:

   ```
   sudo npm install -g @socialgorithm/uabc
   ```

And now continue reading the docs :)

Ultimate Tic Tac Toe

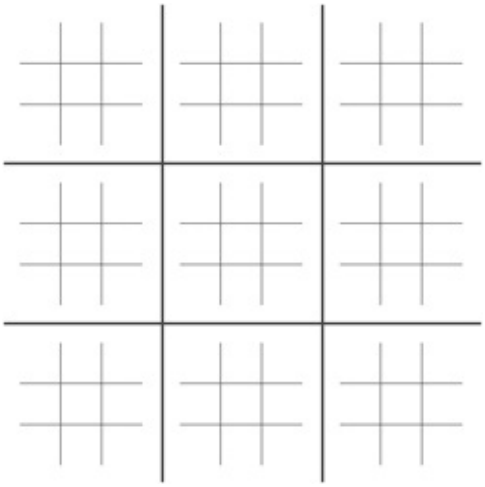

Figure: Ultimate Tic Tac Toe board, with an animation for some rounds

This is the board for an **Ultimate Tic Tac Toe** game, it's essentially a collection of nine small *regular* Tic Tac Toe games.

On each turn, the valid small board is **highlighted in yellow**. Each player's move is displayed in *blue/red*.

You'll be playing on the big board, but in order to *"win"* each cell, you have to win the **game within that cell**.

The first player can play in any cell of any of the small boards. The coordinates of the **move in the small board** determine which *board* the next player has to use to play.

Rules

1. You only play on the small boards.
2. The first player can play on any cell of any small board.
3. You must play on the small board that corresponds to the cell the other player played at.
4. If you are *sent* to a board that has been won already, you must play on any other board. *(Some people play with variations of this rule, but we believe this makes for the most interesting algorithms)*
5. To win the game you must have 3 in a row in the big board in any direction.

Example

If this is the first move of the game:

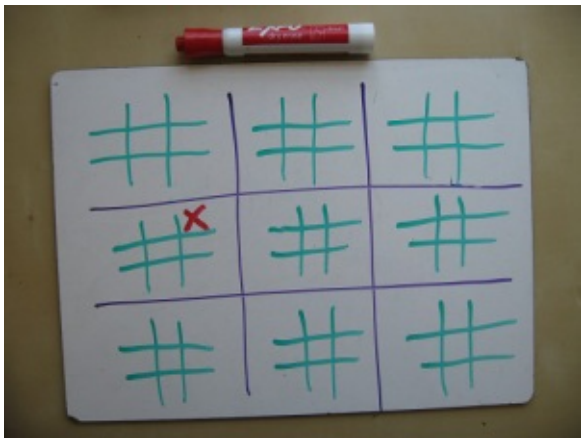

Figure: Example of a first move

Then the other player has to play on the top right board:

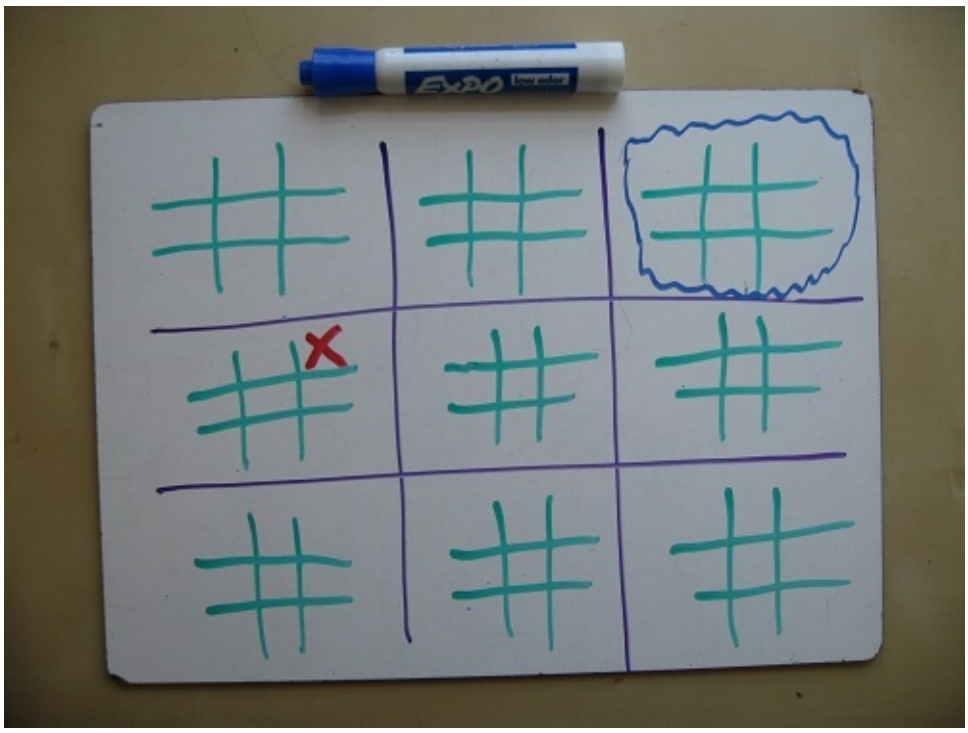

Figure: Board where the next player has to play in

Playing

The game may seem simple at first, but the fact that you get to decide where your opponent plays next means that you'll have to look ahead several turns to decide what the best move is.

You should start by playing at least once on paper with a friend, to get an idea of how the game goes. Sometimes giving up one of the smaller games is necessary to win the big one!

You are now ready to start writing your own player!

Writing a Player

We're going to write an algorithm that can play UTTT, exciting right? :)

At the most basic level, a player needs to be able to calculate a **valid move**, and receive **opponent moves**. Every time it receives a move from its opponent it will run some calculations, and return what it considers to be the optimal move.

A simple start would be to have a program that does just that:

```
// pseudo-code example of the most basic methods
addOpponentMove(board, move) {
    // store his move
}

addMove(board, move) {
    // store our move
}

getMove() {
    // calculate a new move
    return {
        board: ...,
        move: ...
    };
}
```

The problem with his is that we need to store and know the **game state** - have a board, calculate whether a player has won, check if a move is valid...

Initial implementations

On top of writing the game engine, we have created initial implementations of players that are ready to go. They are choosing their moves at random at the moment though, so you'll have to give them some AI ;)

1. Choose the language you want to start in (JavaScript or Python)
2. Go to Github and fork either of the following repositories
   - JavaScript Starter Project
   - Python Starter Project
3. Clone the repository locally: `git clone {repository url}`
4. Start working on your player, and check out the game engine docs for your language:
   - JS Game Engine Documentation
   - Python Game Engine Documentation

> These implementations use the game engines linked above - please visit those repositories for API documentation on what attributes and methods the engines expose!

Head over to Testing locally to see how you can start playing games with your algorithm!

Game Engines & Documentation

Both of those projects use implementations of the Ultimate TTT game logic written by us. If you want to start a player from scratch you can use those directly, or port them to another language.

- JavaScript

- Python

Both projects contain the documentation on how to use them and what API they expose.

JavaScript Starter Project

After you've cloned the repository, run the following command to install the dependencies:

```
$ npm install
```

You can now test the player using:

```
$ uabc -p -g 10 -f "node player.js"
```

The `player.js` file does the stdin/stdout work, and you shouldn't edit it. It requires the file `src/defensive/logic.js`, which is the one you could start with.

Inside the `src` folder you'll find three implementations:

- `defensive/logic.js` - is a good start, as it already provides some minimal logic to compete.
- `firstAvailable/logic.js` - selects the first available valid move, whatever that is.
- `random/logic.js` - selects any valid move at random.

It's interesting to look at each of these to see how they implement the logic and learn about it before starting to make changes.

Python Starter Project

After you've cloned the repository (Initial Implementations section above), you need to ensure you have Python3 installed, run `python3 -v` and ensure it prints 3.6.

> If you don't have Python3 installed, see https://www.python.org/downloads/ for instructions, and download the latest version (3.6).

Once that's done, you need to install the `ultimate_ttt` package on which this player depends:

```
$ pip3 install ultimate_ttt
```

We have provided the `ultimate_ttt_player/random_player.py` file to get you started.

The only method you need to edit to improve this player is `get_my_move` (line 25) - and any other method that this would call. You can see that the random player just picks a valid board at random, then a valid position within that board. You can obviously do better... ;)

After you have made some changes, make sure to test your player works properly:

```
pip3 install . --upgrade
python3 setup.py test
```

If you think you have a good approach, you can test your player against a random one using uabc:

```
uabc -p -f -g 100 "python3 samples/random_player_wrapper.py"
```

We have prepared some ideas on how to write the AI for your player that may help you out!

Otherwise, let's move on and learn how to test your player locally.

There are endless ways in which you can write the logic for your player, which is part of the fun! But here we're going to explore two ideas.

The first is easier to implement, while the second is harder but should choose the best move every time.

Idea 1: Heuristics

How do we choose a move without examining every single combination of moves? We try to see how good/bad a move is by analysing some of its characteristics.

For instance, if the opponent can win after you play a certain move, that makes it a pretty poor choice :)

What we can do is assign an initial score of 0 to each available move in the board we have to play in. Assuming it's empty our initial "score matrix" will look like this:

```
[
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0],
]
```

Now we define a bunch of **conditions** we'll be looking for, and a score modifier for each condition.

Examples:

1. If the move wins me the small board -> +1
2. If the move loses me the small board -> -1
3. If the board where I'll send the opponent can be won by them -> -1
4. If the board where I'll send the opponent is winnable by me -> -1 (because they might block the move)
5. If by playing this move I block 2 in a row from the opponent -> +1

You should also adjust the modifiers, and keep testing the performance of your player against a random implementation, or against the same implementation with different scores, to optimize for the best combination.

If you apply this algorithm correctly, after running the conditions you have defined you'll be left with something like this:

```
[
  [3, -1, 2],
  [1, 2, 0],
  [-4, -2, 1],
]
```

And we can see that the move in `[0, 0]` is the best with a score of `3`.

Idea 2: Monte Carlo Tree Search

MCTS also uses heuristics, but is a more thorough algorithm typically used in similar situations.

The wikipedia page linked above does a really good job of explaining the basics, and enough is already available online about this algorithm so we won't go into more detail.

Keep in mind that for the competition there is a timeout per move, so make sure that you tune the algorithm appropriately.

You should now be ready to learn how to test your player locally!

Testing Locally

> If you've started from one of the sample implementations then you can skip to Installing the Client!

In order to test locally/compete, we have written a small client utility that will execute your algorithm and handle everything.

This means that your player needs to be able to read/write to stdin/stdout *(Don't worry, pretty much every programming language does this easily)*.

Our client will execute your player, and send commands like `move` , or `opponent 1,0;1,1` .

Client Commands

The client will execute your algorithm as a child process, and pipe commands to its stdin/stdout.

For you, this means that you'll do something like the following (JavaScript pseudo-code):

```
process.on('stdin', doSomething); // read commands

console.log('command'); // write commands
```

This means that to get debug statements on the console you'll need to use **stderr**. Every language usually has a function to log to it. In JavaScript you can just use `console.error('text')` for example.

Listening

The uabc client will send you one of the following:

Command	Expects response	Description
`init`	No	The server is telling you that a new game is starting, clear your state and await further commands
`waiting`	No	The server is telling you that the other player is not ready yet
`move`	Yes	Move request, since you can answer directly to an opponent move, this is usually only necessary once at the beginning of the game
`opponent row,col;row,col`	Yes	Sent after an opponent move, it contains their move data in the form `main_board.row, main_board.col; sub_board.row, sub_board.col` . After receiving this you can directly answer with your move

Responding

The only possible response at any given time is a move. If you answer out of place you will lose the game.

Response format

`row,col;row,col` - Where the first coordinates point to a sub_board/game of the main board, and the second are the cell co-ordinates of the sub board (`main_board.row, main_board.col; sub_board.row, sub_board.col`)

Example

The following is a sample game from the point of view of player 1 (the input/output identifiers were added for clarity)

```
[input] init
[input] waiting
[input] opponent 0,0;2,2
[output] 2,2;2,0

[input] opponent 2,0;2,0
[output] 2,0;2,1

[input] opponent 2,1;2,1
[output] 2,1;0,1

...
```

Installing the Client

> You'll need to have Nodejs and NPM installed in your computer. Make sure you have Nodejs version > 6 running (you can check with `node -v`), otherwise upgrade now. This will fix an issue when using the `--log` option on the client.

Install the executable:

```
$ npm install -g @socialgorithm/uabc
```

Verify the installation by running:

```
$ uabc --version
```

Playing Locally

To start a local practice round, with logging to file and console, run:

```
$ uabc -p --log --verbose -f "node path/to/player.js"
```

- `-p` puts the client in practice mode
- `--log` enables logging to a file
- `--verbose` enables logging to the console
- `-f` defines the path to your executable player (for scripting languages like JavaScript or Python you may have to add `node` and `python` to the path, as seen in the example)

For an explanation of all the options run `uabc -h`.

Pro Tip!

Logging to a **file** is very useful because you can then *upload your log file* to see the games and analyze the moves one by one.

Seriously, this is going to be *very* useful, **try it**!

Are you beating the random player consistently? How about competing against the rest of players? :) (although this will have to wait until everyone is ready)

Competing

So, you've written the smartest AI algorithm to play Ultimate Tic Tac Toe? Let's play against everyone else's!

Each client will connect to our server, and they will play 100-1000 games against each other in a tournament style competition, where algorithms will keep advancing towards the final round.

Please make sure that you have tested your client locally and it works. Ideally it should beat the random implementation every single time consistently.

Connecting to the Server

Similarly to local testing, you'll be using the `uabc` client to connect. Simply run the following command:

```
$ uabc --host {server host} --token {your token} -f "node path/to/player.js"
```

- The `host` will be given to you on the competition day.
- The `token` can be anything unique that identifies your team. The server won't allow duplicate tokens.

Once connected just leave the terminal window open, the client will wait for server commands, and play when it's told to play.

If your player crashes simply run the command again, if you were in the middle of the game you will have lost it, but you can continue playing after that. The server will handle all these cases (hopefully).