Testing ϵGroundedFairBot in a Transparent Prisoner's Dilemma Tournament

Caspar Oesterheld

February 8, 2018

Contents

1 Introduction

In this short document I describe how I tested ϵGroundedFairBot (see algorithm 1) against the submissions in Alex Mennen's transparent prisoner's dilemma tournament[1] – that is, a tournament of programs playing the one-shot prisoner's dilemma against each other while being able to read each other's source code.

Data: own source code p and opponent source code p'
Result: action $a \in \{C, D\}$
if *sample* $(0, 1) < \epsilon$ **then**
  return C
end
return $sample(apply(p', (p, p')))$

Algorithm 1: The ϵGroundedFairBot. The program makes use of a function *sample* which samples uniformly from the given interval or probability distribution. Furthermore, $apply(p', (p, p'))$ interprets p' for the input (p, p'). It is assumed that ϵ is computable.

2 Technical setup

In this section, I briefly describe the technical details of how I ran the tournament. It is primarily relevant for readers interested in reproducing my results or testing different strategies in the given tournament environment.

[1] Mennen announced the tournament at `https://www.lesserwrong.com/posts/BY8kvyuLzMZJkwTHL/prisoner-s-dilemma-with-visible-source-code-tournament` and the results at `https://www.lesserwrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0`.

The program tournament was conducted in Scheme. To execute Scheme code, I used Racket[2] version 6.11. In the original announcement of the tournament, Mennen stated that he, too, would use Racket. However, at the time when the tournament was conducted, the current version of Racket was 5.3.5[3].

As far as I know, Mennen hasn't published the code he used for conducting the tournament. Therefore, I used the code provided by user "selbram"[4], which has the additional advantage of allowing the tournament to be run multiple times. This is useful considering that many of the submitted programs take actions non-deterministically.

Relative to selbram's code, I made the following changes:

- I added `#lang scheme` at the top to ensure that racket interprets the program as a Scheme program.
- I removed PBot (submitted by user "nshepperd"), because it did not run properly.[5] This bot performed badly in the original tournament.

Inserting the code for ϵGroundedFairBot as the new PBot, I then ran the tournament using:

```
$ racket
Welcome to Racket v6.11.
> (enter! "testrig.scm")
```

There were still some issues hinting at backward compatibility issues. In particular, I got the following error:

```
; curry: undefined;
;  cannot reference undefined identifier
; [,bt for context]
```

Furthermore, the output was not formatted properly. However, the results roughly match user selbram's. Thus, I did not investigate these issues further.

3 A note on the quality of the submissions

There are many reasons to assume that the submissions for the tournament were not the best ones one could achieve:

- As far as I know, the tournament was solely announced on the LessWrong community blog.
- There was a prize of 0.5 Bitcoin, worth about $50 at the time[6].
- As noted in the comments to the original announcement and the results, many people did not know the programming language Scheme/Lisp/Racket. That said, this problem seems hard to avoid, since the most common programming languages (Java, C/C++, Python, etc.) do not have an `eval` function.
- As noted in the comments to the original announcement and the results, the need to quine induces a high barrier for high-quality entries. This could have been avoided by providing programs with their own source code as an additional argument.

While it was difficult and not strongly incentivized (e.g., via prizes) to submit a strong program, it was easy to submit fairly simple programs. Despite all of this, there were a number of intricate and thought out submissions.

But there were also a few weak programs that affect the success of other strategies. For example, there were three programs (H, I and J) that just cooperate without doing any kind of simulation. Another particularly

[2] See `https://racket-lang.org/`.

[3] See `https://download.racket-lang.org/all-versions.html`.

[4] See `https://www.lesserwrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0#8wHjftMdLzP22GocW`.

[5] Apparently, this is not a problem with my execution environment. See `https://www.lesserwrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0#itCdWwJFHoYbnsT6L`.

[6] See `https://en.wikipedia.org/wiki/History_of_bitcoin#Prices_and_value_history`.

unsuccessful bot (U) simply tests whether the opponent's source code contained expressions such as `quine`. The author of another unsuccessful submission wrote[7]:

> I'm not surprised that laziness did not pay off. I wrote a simple bot, then noticed that it cooperated against defectbot and defected against itself. I thought to myself, "This is not a good sign." Then I didn't bother changing it.

4 Implementing a quining ϵGroundedFairBot in Scheme

Since the program tournament was conducted in Scheme, I had to implement ϵGroundedFairBot in Scheme. Because programs weren't given their own source code, I had to implement a quine version of ϵGroundedFairBot, that is, a program that implements ϵGroundedFairBot by reproducing its own source code. My implementation is given in algorithm 2.

```
(define PBot
 '(lambda (opp)
    (define (all-but-last l) (reverse (cdr (reverse l))))
(define (last_element l)
     (cond ((null? (cdr l)) (car l))
        (else (last_element (cdr l)))))
(define (repl lst to)
     (cond ((null? lst) '())
           ((list? (last_element lst)) (append (all-but-last lst) `(,(repl (last_element lst) to))))
           (else (append (all-but-last lst) `((quote ,to)))))
   )
(define (full_repl lst) (repl lst lst))
     (if (eq? (random 200) 0) 'C
   ((eval opp) (full_repl '(lambda (opp) (define (all-but-last l) (reverse (cdr (reverse l))))
(define (last_element l)
     (cond ((null? (cdr l)) (car l))
        (else (last_element (cdr l)))))
(define (repl2 lst to)
     (cond ((null? lst) '())
           ((list? (last_element lst)) (append (all-but-last lst) `(,(repl (last_element lst) to))))
           (else (append (all-but-last lst) `((quote ,to)))))
   )
(define (full_repl lst) (repl lst lst))
(if (eq? (random 200) 0) 'C  ((eval opp) (full_repl y)))))))))
```

Algorithm 2: An implementation of ϵGroundedFairBot in Scheme, where $\epsilon = 0.5\%$

The algorithm follows a standard pattern of quines. It first defines a function `full_repl` that replaces the last item of a nested list with that list itself. To output the source code of the program, `full_repl` is then applied to a copy of the source code, where the input of `full_repl` is replaced with a marker `y` that `full_repl` then replaces with another copy of the source code with that marker in place of the input of `full_repl`.

5 Results and analysis

I first ran ϵGroundedFairBot against the original submissions to the tournament (removing the dysfunctional PBot). The results can be found in table 1.[8] The ϵGroundedFairBot came in 6th out of 21. While this is a

[7] See `https://www.lesserwrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0#pBLaQqkdphqR7Zax5`.

[8] I uploaded the output of running this tournament to `https://foundational-research.org/files/typescript10cleaner.txt`. The output contains detailed information about the outcome of each individual matchup. For convenience, I cleaned up the output.

Rank	Bot	Score
1	K	3744
2	S	3704
3	Q	3597
4	N	3553
5	R	3371
6	P	3263
7	C	3258
8	A	3257
9	F	3138
10	O	3039
11	E	3037
12	B	3001
13	G	2892
14	L	2863
15	T	2722
16	D	2678
17	J	2664
18	H	2620
19	I	2610
20	M	2526
21	U	2522

Table 1: Running ϵGroundedFairBot in the original tournament environment. P refers to ϵGroundedFairBot, where $\epsilon = 0.5\%$. 100 rounds were played.

respectable result, it loses out to more exploitative bots. As mentioned in section 3, three CooperateBots (H, I and J) were submitted in the original tournament. Another few bots (A, C and Q) mereley flipped a coin (in the case of Q a biased coin) to decide whether to cooperate or defect. Others tried to see whether cooperation would make their opponents cooperate but did so in ways that can easily be tricked, e.g. by including the pattern `quine` in the source code. The most successful bots were thus ones that benefitted from defecting against opponents who choose their strategies without (properly) looking at *their* opponents. For example, K achieved first place by defecting with $> 90\%$ probability against everyone but ϵGroundedFairBot. At the same time, it is so complex that it fools some opponents into cooperating. It also includes the pattern `quine`. S is simpler. It cooperates with CooperateBots in an attempt to elicit cooperation from bots (like B) that cooperate if their opponent cooperates against CooperateBots. Otherwise it defects, but only after waiting for a few seconds to pretend that it is thinking.

I conducted a second tournament[9] in which I only included the best programs from the first round.[10] This served two purposes. First, it is a test of the hypothesis that success in the first round is in great part determined

In particular, I used

```
$ cat typescript10 | tr -d '\040\011\012\015' > typescript10clean
```

to remove the abundant "white space" in the output. I then used

```
$ cat typescript10clean | sed 's/\#hash/\'$'\n\#hash/g' > typescript10cleaner
```

to insert new linebreaks that make the output at least slightly more readable. I also removed error messages and irrelevant output.

[9] I could also have generated the results of the second tournament from a subset of the results of the first tournament.

[10] This iterative/evolutionary process is a standard approach to such program tournaments (see, e.g., Axelrod, 2006, ch. 2). Besides punishing programs that mostly succeed by exploiting weak programs, it also prevents strong forms of collusion (see, e.g., Slany and Kienreich, 2007) by the same mechanism. Such an evolutionary procedure has also been proposed in the comments on the results of Mennen's tournament, see `https://www.lesserwrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0#sBXHKHFRDmQnLB4kN`.

Rank	Bot	Score
1	N	1930
2	P	1841
3	S	1776
4	R	1733
5	F	1691
6	O	1626
7	K	1440
8	B	1429
9	Q	1385
10	C	1342
11	A	1285
12	E	1139

Table 2: The results of a tournament of the best 12 contenders of the first round of the program tournament. P refers to ϵGroundedFairBot, where $\epsilon = 0.5\%$. 100 rounds were played.

Rank	Bot	Score
1	R	803
2	F	800
2	N	800
4	P	793
5	O	502
6	S	500

Table 3: The results of a tournament of the best 6 contenders of the second round of the program tournament. P refers to ϵGroundedFairBot, where $\epsilon = 0.5\%$. 100 rounds were played.

by taking advantage of weak submissions. If this is true, then ϵGroundedFairBot should rank higher in the second round and some of the successful programs from the first round should fall as the programs on the shoulders of which their success is built disappear. Second, it is interesting to see whether P is part of the set of programs that survive until the end of such an evolutionary process. For example, even if the final "equilibrium" is one in which all programs cooperate with one another, it seems plausible that they do so by some mechanism that does not allow for cooperation with ϵGroundedFairBot.

For the second round of the tournament, I included only the best 12 programs from the first round. The results can be found in table 2.[11] We mostly see our hypotheses confirmed. The ϵGroundedFairBot rises in rank and some of the previously successful programs – in particular K – perform much worse in comparison than in the first round.

I conducted a third round of the tournament, in which I only included the 6 most successful programs from round two. The results can be found in table 3.[12] Now a clear picture emerges, where one group of "survivors" manages to cooperate with one another, whereas a second group defects and elicits defection from everyone else.

In conclusion, ϵGroundedFairBot's non-exploitability and ability to cooperate with a wide range of opponents help it to be among the top programs and to make it into the final population. But to be a contender for first place in the first round, it lacks the means to exploit weak bots that are eliminated in later rounds. Thus, to modify ϵGroundedFairBot into a more successful program, one would have to add, for example, a mechanism for detecting opponent programs (like CooperateBot) that do not analyze their opponent or include patterns like the word "quine" into the source code.

[11] For the detailed results, see `https://foundational-research.org/files/typescript11cleaner.txt`.
[12] For the detailed results, see `https://foundational-research.org/files/typescript12cleaner.txt`.

References

Axelrod, Robert (2006). *The Evolution of Cooperation*. Basic Books.

Slany, Wolfgang and Wolfgang Kienreich (2007). "On some winning strategies for the Iterated Prisoner's Dilemma or Mr. Nice Guy and the Cosa Nostra". In: *The Iterated Prisoner's Dilemma. 20 Years On*. Ed. by Graham Kendall, Xin Yao, and Siang Yew Chong. Vol. 4. Advances in Natural Computation. World Scientific. Chap. 8, pp. 171–204.