Co-Evolution: The Imitation Game

Presented to Prof. Nawwaf Kharma (COEN6321 Final Project)

Vasken Dermardiros
Student ID: 26049516
Concordia University
Montreal, QC, Canada

***Abstract*— This paper describes the co-evolution of two game players. The goal of co-evolution is to remove the human from the loop and allow the program to evolve autonomously through self-play until it reaches and surpasses human-competitive levels.**

Two games are played. In the first, a predictor tries to predict the future output generated by the generator. The generator bases its output on its previous move. As the predictor correctly predicts the output, it gains points. As the predictor fails, it losses points. The second game is Rock-Paper-Scissors. Here, two predictors will attempt to predict their opponent's move and beat it.

Running both games with various parameters resulted in four distinguishable patterns: (1) chaotic, (2) semi-chaotic, (3) cyclical where the winner does not improve its strategy until it is beat, and (4) cyclical where both sides continually co-adapt. These patterns were observed irrespective of population size, however, a larger population size would better preserve diversity.

I. Introduction

PRACTICE makes perfect. For a computer player, there are a couple of options. Either the human programmer feeds the program pre-played games from which the program infers good plays, or the human plays against the program forcing it to get better over time in order to win, or the program is left to play against other better programs or against itself.

The last two options that do not require human intervention are the most interesting. By removing the human from the loop, the programs are left to learn and evolve on their own, and, unlike humans, computer programs can play multiple games simultaneously and continuously. Recently, a computer player – Google DeepMind's AlphaGo – beat the world's best Go player – Lee Se-dol – 4 to 1 [1]. DeepMind trained AlphaGo initially using a database of expert games, but then let AlphaGo play against itself and its own variants. AlphaGo eventually became an expert player, beating all expectations and predictions on the advancement of machine learning and artificial intelligence. A computer program to defeat a human professional Go player was thought to be at least a decade away [2].

AlphaGo relies in part on reinforcement learning to improve its game play [2]. In this paper, we will be using another approach that still relies on self-play: co-evolutionary genetic algorithms.

Genetic algorithms are inspired by the natural evolution of species [3]. Instead of having an individual program playing a game, or, generally, optimizing a design, a population of individuals are all playing the game at the same time. The individuals that perform best will be selected as parents and recombined to form children – new solutions. A small mutation is applied to the children to allow the exploration of the solution space. The whole population is evaluated and the fittest individuals are selected to form the next generation. The process of parent selection, recombination, mutation and survivor selection is repeated until the overall population's fitness reaches a certain *design* fitness or exhausts computational resources.

Looking back to nature once again, we seldom observe populations living in complete isolation. They live with other populations and all of them are trying to survive and adapt to one another. The adaptation falls within two categories: cooperation and competition. Through cooperation, two populations help each other survive and improve their health much more than if they were independent of one another. Through competition, two populations prey on each other and try to exploit each other's weaknesses. Competition based co-evolution has made faster rabbits and smarter fox.

In this paper and term project, a co-evolutionary based genetic algorithm is considered. Here, two computer players will try to outmanoeuvre each other. In the first part, **a predictor-generator game** is played where a predictor population tries to predict the generator's future pattern and is rewarded for every accurate prediction. Whereas the generator tries to fool the predictor and is rewarded for every inaccurate prediction made. In the second part, **a rock-paper-scissor (RPS) game** is played by two different predictor populations. Each predictor tries to beat the other player following the RPS rules.

The goal of co-evolution is to remove the human from the loop and allow the program to evolve autonomously through self-play until it reaches and surpasses human-competitive levels.

II. Predictor-Generator Game

Two players, the *generator* and *predictor*, are playing a very simple game. The generator generates a bit (0 or 1) per *round* based on what he had generated during the previous round – in this section, the generator will be referred to as *he* and the predictor as *she*. The predictor tries to predict the bit the generator will generate. As the game progresses, the generator will try to generate bits that the predictor will fail to predict. Whereas the predictor will try to correctly predict the generated bits. For each bit the predictor correctly predicts, she gets a point and the generator losses a point; conversely, for every bit she mispredicts, she losses a point and the generator gains a point.

The generator and predictor that are playing and writing the bits into the play history are the *champions*. The champions

are appointed to serve a *term*. During the term, a population of generators and predictors are "looking" at the game and "contemplating" strategies that will score them more points. At the end of the champions' term, the individuals (one from each side) with the best scores will be named champions for the next term. During the champions' term, at each round, a *tournament* is held where individual generators, in small groups, are pitted against each other. The winners of the tournament are selected as *parents* and to be mated to produce *children*. Children are either a result of *crossing over* the parents, by *mutating* a single parent, or are created randomly.

The way the generator generates a bit is by looking in his *rulebook* and finding a *rule* that matches the previously generated bit(s) in the play history. The rulebook contains many *rules*, *e.g.* {'0':'1'} which returns a '1' if it reads a '0' in the play history. Each rule assigns an output to a string of bits. If two strings match the play history pattern, the longer string is chosen. If two equal strings match the pattern, the string with the most *wildcards* is chosen. The predictor predicts a bit in a similar way except that she reads the generators last generated bit(s).

The game goes on for many rounds. As the game progresses, older generated and predicted bits are removed from the play history.

Having gone over the general idea, we will continue by describing the *individual* in the *population*. Then, the whole Co-Evolutionary Process of Generator versus Predictor is explained with results.

A. The Individual and the Population

The individual consists of 3 parts:

1) Rulebook, contains all the individual's rules
2) Allowance
3) Fitness

The rulebook contains the rules which are used to generate or predict the future bits in the play history. The allowance is simply how many rules an individual can still add to his/her rulebook. It is fixed at 1000. The fitness is the score given to the individual based on how correctly it can fool the predictor or guess the generator's moves. The fitness is evaluated on a certain number of past rounds.

The population consists of a group of individuals.

Example of an individual:

```
{'allowance': 999, 'fit': 10, 'rulebook': {'0.000': '1'}}
```

Example of a population of 5 individuals:

```
[{'allowance': 998, 'fit': 3,
    'rulebook': {'0':'0','01':'0'}},
 {'allowance': 997, 'fit': 2,
    'rulebook': {'0001':'1','001':'0','01':'1'}},
 {'allowance': 997, 'fit': 1,
    'rulebook': {'1': '1','10' '0','11':'1'}},
 {'allowance': 999, 'fit': 2,
    'rulebook': {'10':'1'}},
 {'allowance': 997, 'fit': 2,
    'rulebook': {'01':'0','1':'0','11':'1'}}]
```

B. Co-Evolution Process

The co-evolution process used in this project has 8 steps:

1) Evolution Configuration
2) Population Initialization
3) Champion Selection
4) Pattern Generation and Prediction
5) Population Evaluation (Fitness Evaluation)
6) Parent Selection
7) Children Creation
8) Survivor Selection

C. Evolution Configuration

Here, the competition overseer (the user) declares the population size, the initial play history, the length of the champions' term, the size of the initial rules and rulebook (more or less), the ratio of children to parents, the probabilities of crossover, mutation or creation of a new individual, the size of the tournament fighting pit, the length of the histories and the number of rounds to be played. There is also a possibility to fix either champion for the whole duration of the simulation.

D. Population Initialization

The generator and predictor populations are populated with random individuals. Although rule and rulebook sizes are given, the final rule and rulebook size follows a Poisson random distribution with its mean centered on the provided sizes. Initial rule and rulebooks cannot be empty/void.

E. Champion Selection

During the first round, champions are chosen randomly. Afterwards, at the end of the champions' term, individuals with the highest fitness and allowance become the new champions. If two such individuals exist, one is chosen at random.

F. Pattern Generation and Prediction

The generator champion looks at his past history and searches through his rulebook to find the longest rule that matches the pattern; he then plays the corresponding output. If no rule matches the pattern, the generator plays its previous move.

The predictor champion begins by looking at the generator's past history and plays her move, trying to predict the generator's current move. If no rule matches the pattern, the predictor plays a "?" which results in a loss for that round.

As the match progresses, older bits are transferred into the *past history* which has become a string of bits used by the generator and predictor to generate new bits, but their fitness is not evaluated on the *past history*. The oldest bits in the "past history" are deleted.

Here's an example, the newest bits are inserted on the left:

```
{history = 10, prehistory = 5}

** == [ Round 10 ] == **
len = history    | len = prehistory |
generatedPattern | past history     | old/deleted
-------------------------------------------------
      1010110101 | 10101   -->      | 10
       11000?1?1 |         -->      |
-------------------------------------------------
predictedPattern |                  |

** == [ Round 11 ] == **
len = history    | len = prehistory |
```

```
generatedPattern | past history     | old/deleted
-----------------------------------------------
      0101011010 | 11010   -->      | 110
       111000?1? |         -->      |
-----------------------------------------------
predictedPattern |                  |
```

G. Population Evaluation (Fitness Evaluation)

The population is evaluated on the leftmost string of bits with length equal to, at most, the value of variable *history*. For every match, the predictor gains a point and the generator losses one; and vice-versa. It's a zero-sum game.

H. Parent Selection

The population is shuffled and pitted against each other in small groups. The individual in the group with the highest fitness and allowance wins. If two such individuals exist, one is chosen by random. Given the *ratio* of children to parents, the pit size ensuring a constant population size throughout is determined:

$$PitSize = \sqrt{ratio + 1} \tag{1}$$

I. Children Creation

Parents can't be called parents without first having children!

The children are created from the parents through crossover, mutation or randomly – irrespective of the parents. The introduction of random individuals follows the same process as the individuals creation during population initialization. Children are created from parents until the child to parent ratio is satisfied.

1) Crossover: The only crossover used in the project is the uniform type. It consists of taking two parents randomly, let's call them the mother and father, and creating two children, let's call them the daughter and son. The children are created by distributing the parents' rulebook's rules to the children. Starting with the mother, we flip a coin, if it's tails, the first rule goes to the daughter, otherwise, it goes to the son. This process is repeated for every rule in the mother's rulebook. And then, the same process is applied to the father.

If, by chance, either daughter or son end up without rules, a random rule is given to her or him.

2) Mutation: Six mutation types are used in this project and are described:

1) Rule bit flip: {'100': '0'} –> {'101': '0'}
2) Rule bit to wildcard: {'100': '0'} –> {'1.0': '0'}, the point is a wildcard
3) Rule order shuffle: {'100': '0'} –> {'010': '0'}
4) Rule add bits: {'100': '0'} –> {'01010': '0'}
5) Rule flip output bit: {'100': '0'} –> {'100': '1'}
6) Add rule: {'100': '0'} –> {'100': '0', '11': '1'}

J. Survivor Selection

The childrens' fitness is evaluated. The children are then reunited with their parents to form a family. And then they are all pitted against each other in a deathmatch. The tournament selection is exactly the same as what was done during the parent selection process. The survivors become the new population.

This process is repeated until the end of the simulation.

K. Results

During the simulation, results are printed out for the overseer to track the champions. The population fitnesses and allowances are also tracked. The allowance is inversely proportional to the complexity of the individuals.

Here is an output for Rounds 5 and 10 showing how the new champion predictor was able to predict every move of the weak champion generator:

```
**----  The New Champs are Here!  ----*
Round 5
Generator Champ Rulebook: {'.': '1'}
Predictor Champ Rulebook: {'.': '1'}
Generated Pattern: 1111110 (older)
Predicted Pattern: 1????0
Fittest Generator: 5
Fittest Predictor: 6

**----  The New Champs are Here!  ----*
Round 10
Generator Champ Rulebook: {'.': '0'}
Predictor Champ Rulebook: {'.': '1'}
Generated Pattern: 011111111110 (older)
Predicted Pattern: 111111????
Fittest Generator: 10
Fittest Predictor: 9
```

The newest bits (left) are generated/predicted by the new champions. *I.e.*: looking at Round 10, the generator champion compares the bit string '11111111110' with the only rule in his rulebook {'.': '0'} and outputs '0'. Whereas the predictor champion compares the same bit string, reads her rulebook and outputs '1' (which is incorrect).

In later matches, the rulebooks get longer – albeit not *very* long:

```
**----  The New Champs are Here!  ----*
Round 90
Generator Champ Rulebook: {'.1': '0'}
Predictor Champ Rulebook: {'0': '0', '1.': '1'}
Generated Pattern: 000000111111111000000000 (older)
Predicted Pattern: 0111111111
Fittest Generator: 9
Fittest Predictor: 9

**----  The New Champs are Here!  ----*
Round 95
Generator Champ Rulebook: {'.1.': '0', '..1': '1'}
Predictor Champ Rulebook: {'..': '0'}
Generated Pattern: 000000000001111111110000 (older)
Predicted Pattern: 0000001111
Fittest Generator: 9
Fittest Predictor: 10
```

In the Predictor-Generator game, looking at the average fitness over time, we can observe two trends: (1) a chaotic evolution, and (2) a cyclical behaviour. For a population of 360 individuals, the two trends are plotted for average fitness: (1) Figure 1, (2) Figure 3; and for average rulebook size (1) Figure 2, (2) Figure 4.

As the champion's term is extended, the population can adapt to the champions and exploit their strategies, *i.e.* the average population fitness increases until the champions are switches. If the champions are switched every round, the population cannot adapt and so the average fitness is noisy.

III. Rock-Paper-Scissor Game

Two players, Alice and David, are playing a slightly more complex game: rock, paper, scissor (RPS). Unlike the previous

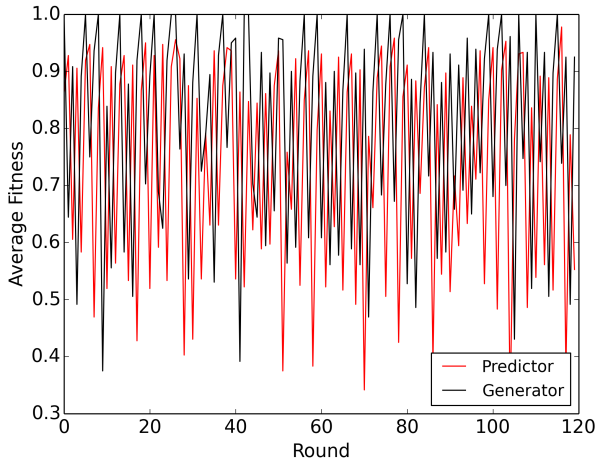

Fig. 1. **Chaotic** (Average Fitness): Population Size: 360, Champion's Term: 1, Pre-History: 16

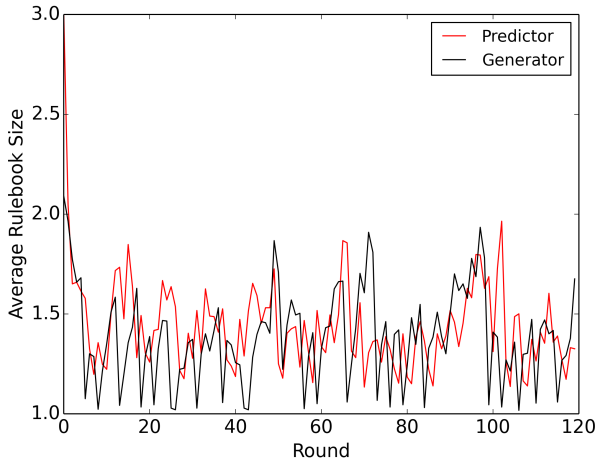

Fig. 2. **Chaotic** (Average Rulebook Length): Population Size: 360, Champion's Term: 1, Pre-History: 16

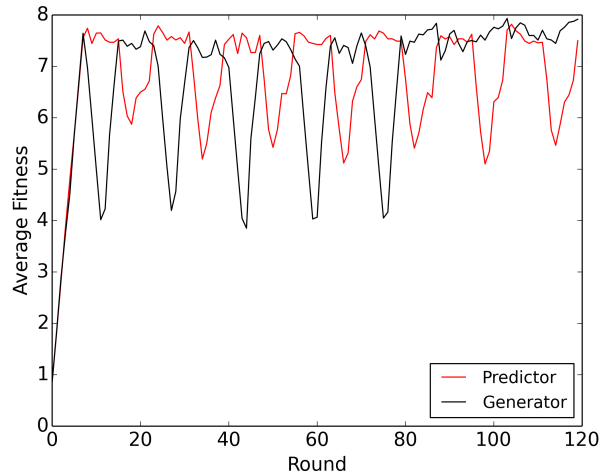

Fig. 3. **Cyclical** (Average Fitness): Population Size: 360, Champion's Term: 8, Pre-History: 16

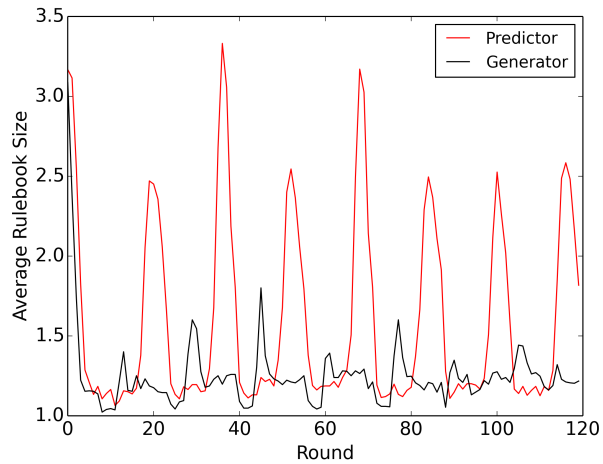

Fig. 4. **Cyclical** (Average Rulebook Length): Population Size: 360, Champion's Term: 8, Pre-History: 16

game, both players here are predictors observing their opponent's moves and trying to predict their next move. The payout structure is described in Table I.

TABLE I
PAYOUT STRUCTURE OF ROCK, PAPER, SCISSOR GAME, *e.g.* ROCK BEATS SCISSOR, SCISSOR BEATS PAPER, ROCK TIES WITH ROCK, ETC.

	R	P	S
R	(0, 0)	(+1,-1)	(-1,+1)
P	(-1,+1)	(0, 0)	(+1,-1)
S	(+1,-1)	(-1,+1)	(0, 0)

The algorithms for selection, crossover and mutation remain relatively unchanged. Instead of the possible choices being {0, 1}, for this game they're {R, P, S}. Here is an example of an individual:

```
{'allowance': 998, 'fit': 4, 'rulebook': {'R':'P','PP':'S'}}
```

When either Alice or David have no matching rules to their opponent's play, they will play the opponent's last move.

A. Results

Alice and David were left to play 120 rounds of RPS under different settings. Table II summarizes the different game parameters.

Having run the simulation for the different cases, we can observe 4 distinguishable patterns: (1) chaotic (See Figure 5), (2) semi-chaotic (See Figure 6), (3) cyclical where the winner does not improve his/her strategy until he/she is beat (See Figure 7), and (4) cyclical where both sides continually co-adapt (See Figure 8).

In the chaotic pattern, no clear trend can be observed. This was the result from when the champion's term was 1 round long. As the champions' terms were made longer, a more orderly pattern can be observed. Since the champions would remain fixed, the rest of both populations are able to adapt to the two. Changing the population size or pre-history length did not have a noticeable difference on the outcome.

Looking now at the length of the population's rulebooks, the average length is proportional to how chaotic the behaviours are. For a very chaotic trend, the number of rules is high,

TABLE II
RPS SIMULATION PARAMETERS

Parameter	Value
Population Size	{180, 360, 720, 1440}
Rounds	120
Champions' Term	{1, 2, 4, 8}
History Length	'= Champion's Term'
Pre-History Length	{4, 8, 16}
Initial Rule Length	3
Initial Rulebook Length	10
Child/Parent Ratio	8
P. Crossover	0.25
P. Crossover:Uniform	1.00
P. Mutation	0.65
P. Mutation:Add Rule	0.35
P. Mutation:Change Output	0.10
P. Mutation:Add Bit	0.10
P. Mutation:Shuffle	0.20
P. Mutation:Wildcard	0.05
P. Mutation:Change Bit	0.20
P. Pass Through	-
P. New Individual	0.10

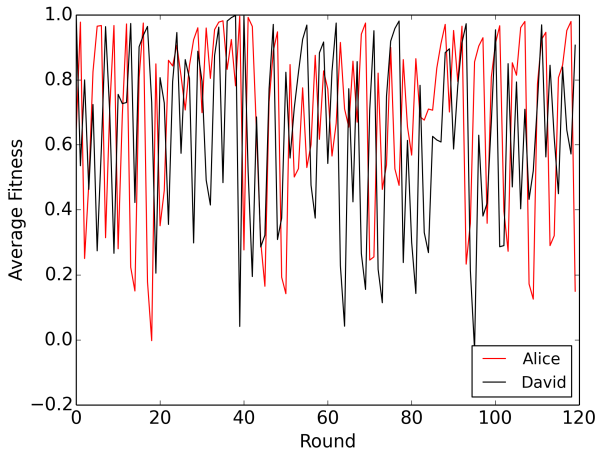

Fig. 5. **Chaotic** (Average Fitness): Population Size: 1440, Champion's Term: 1, Pre-History: 16

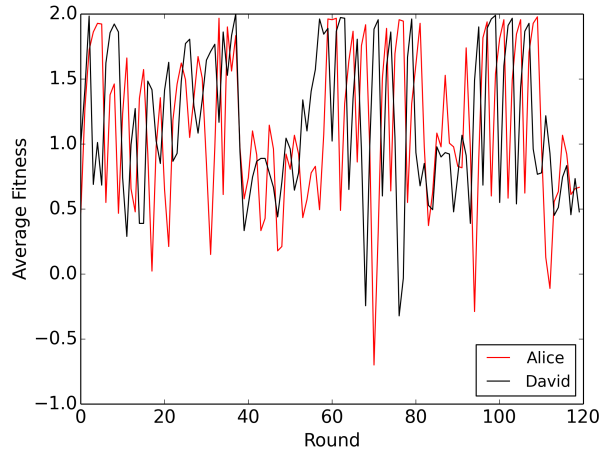

Fig. 6. **Semi-Chaotic** (Average Fitness): Population Size: 180, Champion's Term: 2, Pre-History: 8

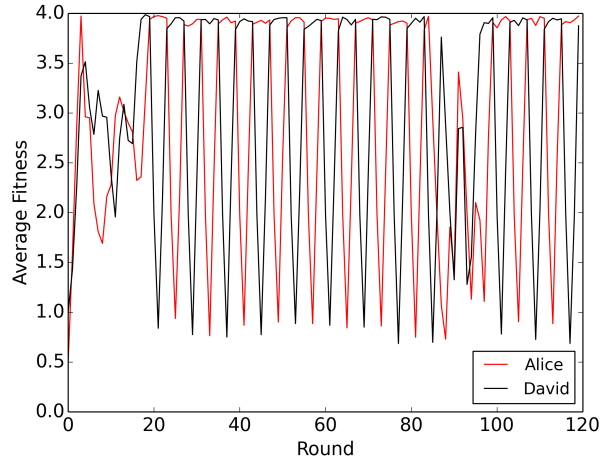

Fig. 7. **Cyclical Loser Adapts** (Average Fitness): Population Size: 720, Champion's Term: 4, Pre-History: 16

and vice-versa for the cyclical cases. The rulebook from the previous pattern (2) is shown in Figure 9.

For the last two patterns, we can observe the two trends in the average rulebook lengths. The first, when a player finds a winning strategy, its populations rulebook length collapses because all the individuals end up playing the winning strategy and the individuals who had losing strategies are eliminated (See Figure 10). The opponent, however, will encourage its population to diversify and try out many different strategies in order to find a winning strategy. Once this strategy is found, now its population diversity collapses. This can be due to a high selection pressure that is resulting in a drastic loss of diversity. The second rulebook pattern is observed in (4) (See Figure 11). Here, both sides simultaneously alter their strategies and no one side dominates the other.

The actual game played by Alice and David that generated pattern (4) is the following. After a short number of moves, both players can no longer beat each other and draw every hand. Since Alice won the first few hands, she ends up winning the match.

```
--------------
 Match Played
--------------
Alice (begin.): PPPPPPPPPRRRRRRRRSSSSSSSSPPPPPPPPRRRRRRRR
David (begin.): PPPPPPPPPRRRRRRRRSSSSSSSSPPPPPPPPRRRRRRRR

Alice (cont'd): SSSSSSSSPPPPPPPPRRRRRRRRSSSSSSSSPPPPPPPP
David (cont'd): SSSSSSSSPPPPPPPPRRRRRRRRSSSSSSSSPPPPPPPP

Alice (cont'd): RRRRRRRRSSSSSSSSPPPPPPPPRRRRRRRRSSSSSSSRP
David (cont'd): RRRRRRRRSSSSSSSSPPPPPPPPRRRRRRRRPPPPPPPR

Alice WINS!
Final Score: Alice:5 David:-5
```

Running the RPS game for longer rounds and for longer pre-history lengths, yields the results shown in Figure 12 and Figure 13.

The results are cyclical and co-evolving, similar to pattern (4) described previously.

IV. CONCLUSION

The paper demonstrated a co-evolutionary genetic algorithm through two simple games. In the first, a predictor population would try to predict the generator's future pattern whereas the

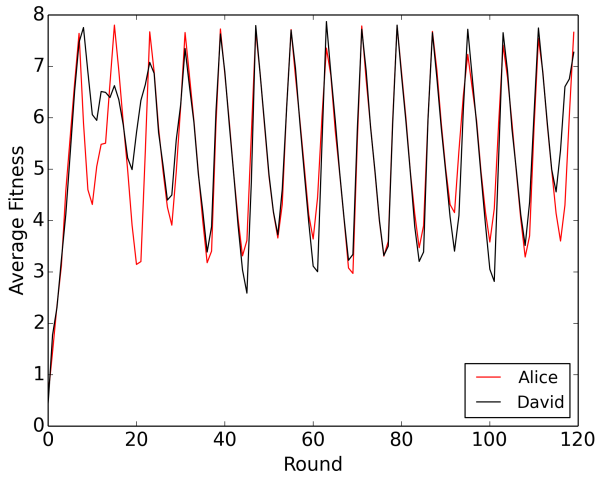

Fig. 8. **Cyclical Co-Adapt** (Average Fitness): Population Size: 360, Champion's Term: 8, Pre-History: 16

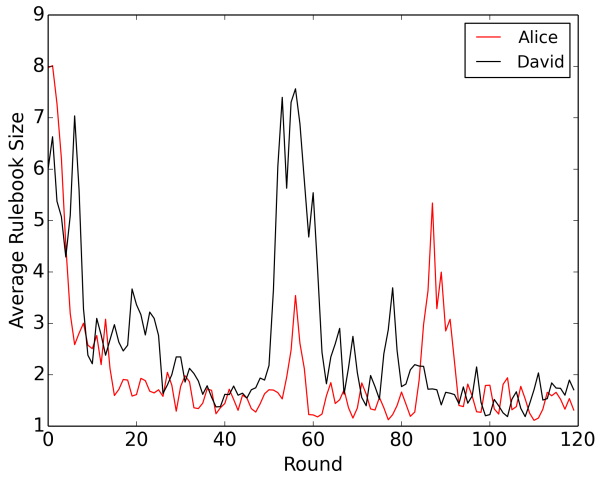

Fig. 9. **Semi-Chaotic** (Average Rulebook Length): Population Size 180, Champion's Term: 2, Pre-History: 8

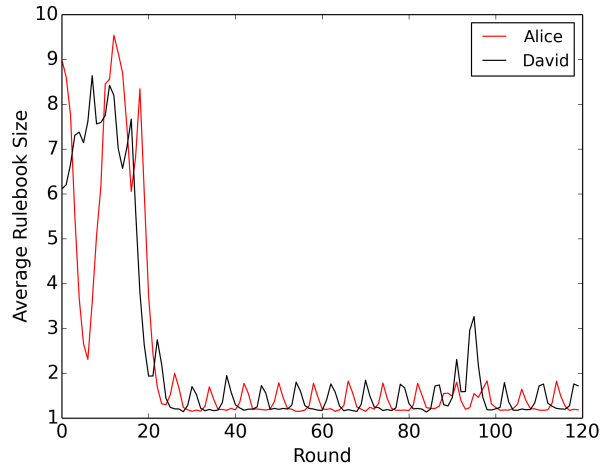

Fig. 10. **Cyclical Loser Adapts** (Average Rulebook Length): Population Size: 720, Champion's Term: 4, Pre-History: 16

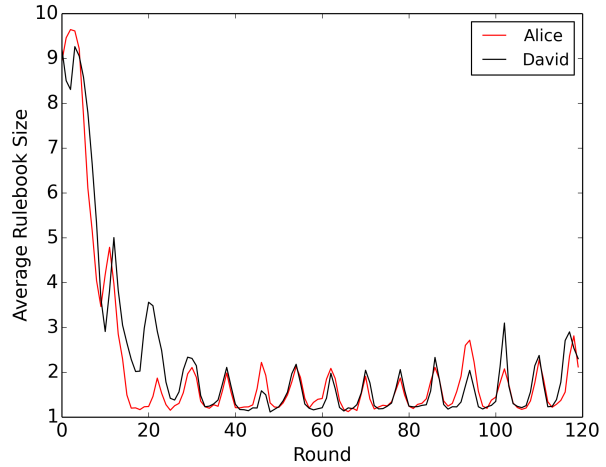

Fig. 11. **Cyclical Co-Adapt** (Average Rulebook Length): Population Size: 360, Champion's Term: 8, Pre-History: 16

generator would generate based on its own previous pattern. For every correct prediction, the predictor would gain a point and the generator would lose one; conversely, for every wrong prediction, the opposite points would be awarded. In the second game, which was rock-paper-scissor, two predictor populations would try to predict each other's moves and play the winning hand. The two games run similarly except the possible moves were different and the fitness rewarding schedule was different.

The results showed that co-evolution occurred for both games. As the champion's terms got longer, a more stable and cyclical trend resulted, whereas for single round terms, chaotic patterns resulted. In the cyclical regime, either the winner would end up not changing its strategy until it starts losing or either both players would continuously update their play strategies. When a winning rule was found, often the population diversity would collapse. Diversity should be explicitly controlled to disallow this outcome. Due to this collapse, many individuals would end up being copies. For cases where evaluating the fitness is computationally demanding, a database look-up approach should be considered instead of re-evaluating every individual one by one.

In a future study, we will try a self-adaptive approach for the RPS game parameters that will evolve depending on the person's playing style. Additionally, we would like to study if a hybrid memetic approach using artificial neural networks would yield better results.

References

[1] S. Byford, "Google's AlphaGo AI beats Lee Se-dol again to win Go series 4-1," Mar. 2016. [Online]. Available: http://www.theverge.com/2016/3/15/11213518/alphago-deepmind-go-match-5-result

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: http://www.nature.com/doifinder/10.1038/nature16961

[3] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*, ser. Natural Computing Series. Berlin, Heidelberg: Springer Berlin

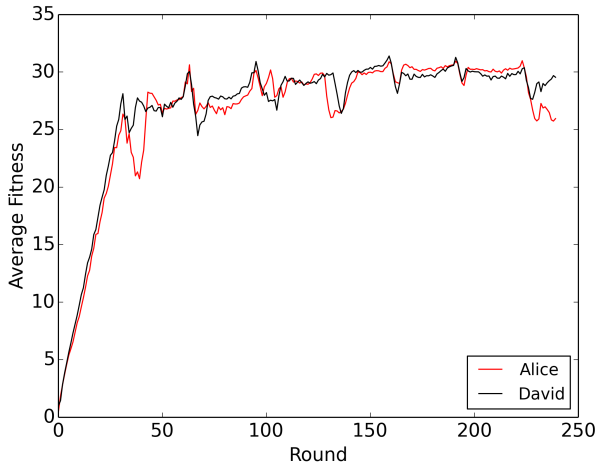

Fig. 12. **Longer Run** (Average Fitness): Population Size: 1440, Champion's Term: 32, Pre-History: 32

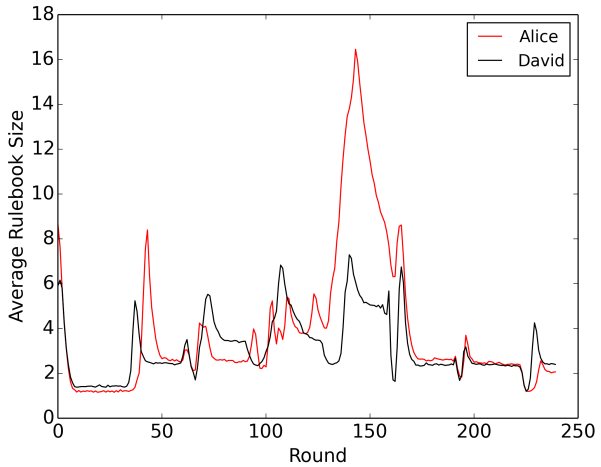

Fig. 13. **Longer Run** (Average Rulebook Length): Population Size: 1440, Champion's Term: 32, Pre-History: 32

Heidelberg, 2015. [Online]. Available: http://link.springer.com/10.1007/978-3-662-44874-8

Appendix

This section shows the evolution of the champions' rulebooks per round for the game shown in Figure 8.

A. Alice

```
1 {'PRP.': 'P', 'S': 'P', 'RRRSS': 'S', '.SRPR': 'P'}
2 {'S': 'P', 'R': 'P', 'PRP': 'P', 'PSRR': 'P',
    'P.PR': 'R'}
3 {'PRRSPP': 'R', 'SS.': 'R', 'RS': 'S', '.RS': 'S',
    '.': 'P'}
4 {'P': 'S', 'S': 'P', 'R': 'S', 'SR': 'P',
    'SRP': 'S', 'SSP': 'R'}
5 {'P': 'S', 'S': 'P', '.SS': 'P', 'R': 'S'}
6 {'P': 'R', 'S': 'P', '.S': 'S', 'SRP': 'P'}
7 {'SPR': 'S', 'RP': 'P', '..': 'S', '.': 'P'}
8 {'P': 'S', 'S': 'P', 'RS': 'S'}
9 {'..': 'S'}
10 {'P': 'S', 'R': 'P'}
11 {'P': 'S', 'S': 'S', 'R': 'S', '.': 'S'}
12 {'P': 'S', 'R': 'P'}
13 {'R': 'P', '.': 'S'}
14 {'S': 'R', 'R': 'R'}
15 {'R': 'P'}
16 {'R': 'P'}
17 {'R': 'P'}
18 {'P': 'S', '.': 'P'}
19 {'P': 'P', 'R': 'S'}
20 {'P': 'S', 'R': 'P'}
21 {'P': 'P', 'R': 'P'}
22 {'.': 'S'}
23 {'P': 'S', 'R': 'P'}
24 {'.': 'S'}
25 {'.': 'S'}
26 {'P': 'S', 'S': 'R'}
27 {'P': 'S', 'S': 'P'}
28 {'P': 'S', 'S': 'R'}
29 {'P': 'S', 'S': 'R'}
30 {'P': 'S', 'S': 'R', '.': 'R'}
31 {'P': 'S', 'S': 'R'}
32 {'S': 'S', '.': 'P'}
33 {'.': 'R'}
34 {'S': 'R', 'R': 'P', '.': 'P'}
35 {'P': 'R', 'S': 'R', '.': 'S'}
36 {'S': 'R', 'R': 'S'}
37 {'S': 'R', 'R': 'R', '.': 'P'}
38 {'S': 'R', 'R': 'R'}
39 {'S': 'R', 'R': 'P'}
40 {'P': 'S', 'R': 'P', '.': 'P'}
41 {'.': 'P'}
42 {'R': 'R', '.': 'S'}
43 {'P': 'S', 'R': 'P'}
44 {'P': 'S', 'R': 'P', '.': 'P'}
45 {'P': 'S', '.': 'P'}
46 {'R': 'P', '.': 'S'}
47 {'P': 'S', 'R': 'P'}
48 {'.': 'S'}
49 {'.': 'S'}
50 {'P': 'S', '.': 'R'}
51 {'P': 'S', 'S': 'S', 'R': 'R', '.': 'R'}
52 {'S': 'R', 'R': 'P', '.': 'S'}
53 {'P': 'S', '.': 'R'}
54 {'P': 'S', 'S': 'R'}
55 {'P': 'S', 'S': 'R'}
56 {'.': 'R'}
57 {'P': 'R', '.': 'R'}
58 {'S': 'R', '.': 'P'}
59 {'S': 'R', 'R': 'S', '.': 'P'}
60 {'S': 'R', '.': 'P'}
61 {'P': 'P', 'S': 'R', '.': 'P'}
62 {'S': 'R', 'R': 'P'}
63 {'S': 'R', 'R': 'P'}
64 {'R': 'S', '.': 'P'}
65 {'S': 'P', 'R': 'R', '.': 'P'}
66 {'.': 'P'}
67 {'R': 'P', '.': 'S'}
68 {'P': 'P', 'R': 'P', '.': 'S'}
69 {'R': 'P', '.': 'S'}
70 {'P': 'S', 'R': 'S'}
71 {'P': 'S', 'R': 'P'}
72 {'S': 'S', 'R': 'P', '.': 'S'}
73 {'.': 'S'}
74 {'P': 'R', 'S': 'R', '.': 'S'}
75 {'P': 'S', 'S': 'R', 'R': 'R', '.': 'R'}
76 {'P': 'P', '.': 'R'}
77 {'P': 'S', '.': 'R'}
78 {'P': 'S', 'S': 'R', '.': 'P'}
79 {'P': 'S', 'S': 'R'}
80 {'.': 'R'}
81 {'S': 'R', 'R': 'P', '.': 'R'}
82 {'R': 'R', '.': 'R'}
83 {'P': 'P', 'S': 'S', 'R': 'P', '.': 'R'}
84 {'S': 'R', '.': 'P'}
85 {'R': 'S', '.': 'R'}
86 {'P': 'R', 'S': 'S', '.': 'P'}
87 {'S': 'R', 'R': 'P'}
88 {'R': 'R', '.': 'P'}
89 {'.': 'P'}
90 {'P': 'S', 'R': 'P'}
91 {'P': 'S', '.': 'P'}
92 {'P': 'S', 'S': 'S', '.': 'P'}
93 {'R': 'P', '.': 'R'}
94 {'P': 'S', 'R': 'P'}
95 {'P': 'S', 'R': 'P'}
96 {'P': 'P', 'S': 'P', '.': 'P'}
97 {'.': 'S'}
98 {'P': 'S', 'S': 'S'}
99 {'P': 'S', '.': 'R'}
100 {'P': 'S', 'S': 'R', 'R': 'S'}
```

```
101 {'P': 'P', '.': 'R'}
102 {'S': 'P', 'R': 'P', '.': 'S'}
103 {'P': 'S', 'S': 'R'}
104 {'R': 'R', '.': 'R'}
105 {'.': 'R'}
106 {'P': 'P', 'S': 'R', '.': 'R'}
107 {'S': 'R', 'R': 'P', '.': 'R'}
108 {'S': 'R', '.': 'P'}
109 {'P': 'P', 'R': 'P', '.': 'R'}
110 {'S': 'S', 'R': 'P', '.': 'R'}
111 {'S': 'R', 'R': 'P'}
112 {'S': 'S', '.': 'P'}
113 {'.': 'P'}
114 {'P': 'S', 'R': 'P'}
115 {'R': 'S', '.': 'S'}
116 {'P': 'S', 'S': 'R', 'R': 'P'}
117 {'P': 'S', '.': 'P'}
118 {'P': 'S', 'R': 'S'}
119 {'P': 'S', 'R': 'P'}
120 {'.': 'S'}
```

B. David

```
  1 {'S': 'S', 'PSPRRRSR': 'P', 'RSR': 'P', '.': 'S'}
  2 {'SRR': 'S', 'PP': 'R', 'R': 'S'}
  3 {'R': 'S', 'RP': 'R'}
  4 {'R': 'S', 'S': 'R', '..': 'R'}
  5 {'R.': 'R', 'S': 'P', '.': 'S'}
  6 {'S': 'R', 'R': 'P', 'RP': 'R'}
  7 {'R.': 'R', 'R': 'R', 'S': 'R'}
  8 {'P': 'S', 'S': 'R', '..': 'R', '.': 'P'}
  9 {'PR': 'S', 'S': 'R', 'RSPRRSRPR': 'R', 'SSPRS': 'R'}
 10 {'RPR': 'S', '..': 'R', 'RS': 'S'}
 11 {'S': 'R', 'R': 'P', 'S.R': 'P', '..': 'R',
     'RS': 'P', 'R.S': 'P'}
 12 {'PR': 'P', 'PS.': 'S', '..': 'R', 'SS': 'R',
     'RSP': 'R', 'P': 'S', 'R': 'R', '.R': 'P',
     'RRS': 'P', 'SRSSP': 'R'}
 13 {'S': 'R', '..': 'R', 'SR': 'P', 'RSR': 'P',
     '.': 'P'}
 14 {'P': 'R', 'R': 'P', '..': 'R', 'SR': 'S',
     'SP': 'R', 'RRS': 'P'}
 15 {'SR': 'P', 'R': 'P', 'RRS': 'P', 'S.': 'R'}
 16 {'P': 'P', 'S.': 'R', 'SR': 'P', 'RRS': 'P',
     '.': 'P'}
 17 {'.': 'P'}
 18 {'P': 'S', 'R': 'P'}
 19 {'S': 'R', 'R': 'R', '.': 'S'}
 20 {'R': 'P', '.': 'S'}
 21 {'.': 'R'}
 22 {'.': 'P'}
 23 {'P': 'S', 'R': 'P'}
 24 {'.': 'S'}
 25 {'P': 'S', 'S': 'P', 'R': 'S', '.': 'S'}
 26 {'P': 'S', 'S': 'R'}
 27 {'P': 'S', 'S': 'R'}
 28 {'P': 'S', 'S': 'R', 'R': 'P'}
 29 {'S': 'R', 'P.': 'S'}
 30 {'P.': 'S', '.': 'R'}
 31 {'S': 'R', '.': 'S'}
 32 {'S': 'P', '.': 'R'}
 33 {'.': 'R'}
 34 {'S': 'R', 'R': 'P'}
 35 {'.': 'P'}
 36 {'S': 'R', 'R': 'P'}
 37 {'S': 'P', 'R': 'R', '.': 'P'}
 38 {'P': 'S', 'S': 'R', '.': 'P'}
 39 {'S': 'R', 'R': 'P'}
 40 {'.': 'P'}
 41 {'R': 'S', '.': 'P'}
 42 {'S': 'P', '.': 'P'}
 43 {'.': 'P'}
 44 {'.': 'P'}
 45 {'R': 'S', '.': 'R'}
 46 {'P': 'S', 'S': 'S', 'R': 'S', '.': 'R'}
 47 {'P': 'P', 'S': 'S', 'R': 'S', '.': 'P'}
 48 {'.': 'S'}
 49 {'R': 'R', '.': 'S'}
 50 {'P': 'P', 'S': 'R'}
 51 {'P': 'S', 'S': 'R', 'R': 'R'}
 52 {'P': 'P', '.': 'P'}
 53 {'P': 'R', 'R': 'P', '.': 'S'}
 54 {'P': 'R', 'S': 'P', '.': 'R'}
 55 {'P': 'S', 'S': 'R', 'R': 'R'}
 56 {'S': 'R', 'R': 'R', '.': 'R'}
 57 {'.': 'R'}
 58 {'P': 'R', 'S': 'R', 'R': 'P', '.': 'P'}
 59 {'S': 'P', 'R': 'R', '.': 'R'}
 60 {'P': 'P', '.': 'R'}
 61 {'P': 'R', 'S': 'R', 'R': 'P'}
 62 {'P': 'S', 'S': 'S', 'R': 'P'}
 63 {'.': 'R'}
 64 {'S': 'R', '.': 'P'}
 65 {'.': 'P'}
 66 {'P': 'S', 'S': 'P', 'R': 'S'}
 67 {'P': 'S', 'R': 'P'}
 68 {'P': 'S', 'S': 'R', 'R': 'R'}
 69 {'P': 'S', '.': 'S'}
 70 {'P': 'S', 'R': 'R'}
 71 {'P': 'S', 'R': 'P'}
 72 {'P': 'P', 'S': 'P', '.': 'P'}
 73 {'.': 'S'}
 74 {'.': 'S'}
 75 {'S': 'R', 'R': 'S', '.': 'R'}
 76 {'P': 'S', 'S': 'R'}
 77 {'P': 'S', 'S': 'S', 'R': 'S'}
 78 {'P': 'P', '.': 'R'}
 79 {'P': 'S', 'S': 'R'}
 80 {'.': 'R'}
 81 {'R': 'R', '.': 'S'}
 82 {'S': 'S', 'R': 'P'}
 83 {'R': 'P', '.': 'P'}
 84 {'R': 'P', '.': 'R'}
 85 {'P': 'P', 'R': 'P', '.': 'R'}
 86 {'P': 'S', 'S': 'R', 'R': 'R', '.': 'S'}
 87 {'S': 'R', 'R': 'P'}
 88 {'.': 'P'}
 89 {'.': 'P'}
 90 {'P': 'R', 'S': 'S', '.': 'P'}
 91 {'S': 'S', 'R': 'S', '.': 'R'}
 92 {'P': 'S', 'S': 'R', '.': 'S'}
 93 {'P': 'S', 'R': 'S', '.': 'S'}
 94 {'.': 'P'}
 95 {'P': 'S', 'R': 'P'}
 96 {'.': 'S'}
 97 {'.': 'S'}
 98 {'P': 'S', 'S': 'R', 'R': 'P'}
 99 {'P': 'P', 'S': 'P', '.': 'P'}
100 {'S': 'R', 'R': 'P', '.': 'S'}
101 {'P': 'S', '.': 'R'}
102 {'P': 'R', 'S': 'R', 'R': 'S', '.': 'S'}
103 {'P': 'S', 'S': 'R'}
104 {'.': 'R'}
105 {'P': 'P', 'S': 'S', 'R': 'P', '.': 'R'}
106 {'S': 'R', 'R': 'P', '.': 'P'}
107 {'S': 'R', 'R': 'P'}
108 {'S': 'S', '.': 'S'}
109 {'.': 'P'}
110 {'S': 'R', 'R': 'P'}
111 {'S': 'S', 'R': 'R'}
112 {'P': 'S', 'R': 'R', '.': 'P'}
113 {'.': 'P'}
114 {'RRRS': 'S', 'RR': 'P', '.': 'S'}
115 {'RR': 'P', '.': 'S'}
116 {'RR': 'P', '.': 'S'}
117 {'.R': 'P', 'PSP': 'S', '.': 'S'}
118 {'P': 'R', '.': 'P'}
119 {'RR': 'P', '.': 'S'}
120 {'.': 'S'}
```