

Rock Paper Scissor Game Algorithm

*** Swarm mind implementation.

1.0 Class diagram

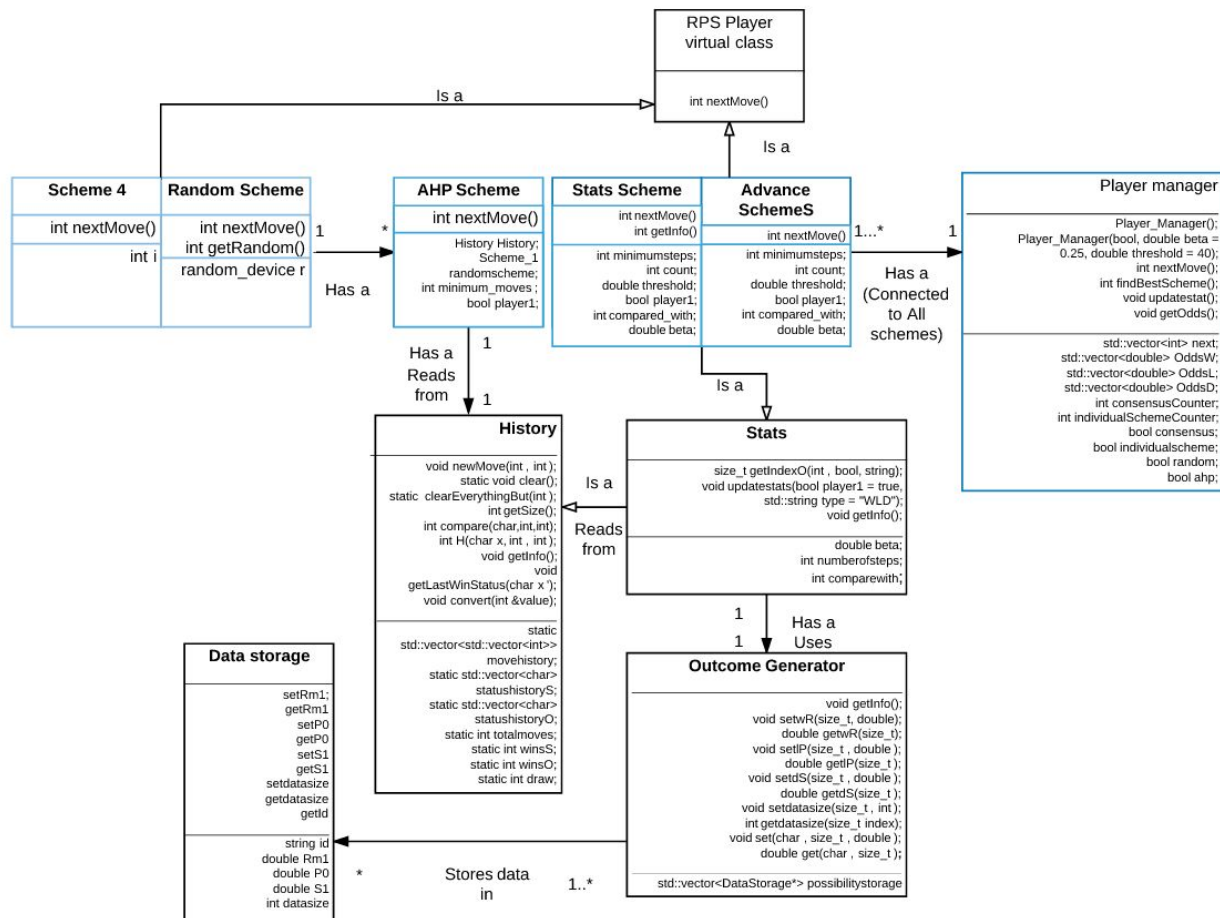


Figure 1. Class diagram

2.0 General algorithm description

In its essence, the system uses a swarm mind algorithm which combines 15 schemes built using 5 different classes.

The objective of the swarm mind is to predict the opponent's (human or other AI) next move in an iterative Rock-Paper-Scissor game of 1000 rounds through majority consensus: each algorithm, out of the 15 total schemes, generates 1 prediction per round (Rock, paper or

scissor) and the most overwhelmingly common majority prediction is used as the next move to be played. In the absence of a majority consensus from the system OR if previous swarm consensus did not result in sufficient victories, the most overwhelmingly successful individual algorithm out of the 15 schemes is used to predict the next game move to be played. Lastly, in the absence of an overwhelmingly successful algorithm, a random move is picked to be played. The 15 schemes are grouped and orchestrated by one single umbrella Class, called Player_Manager.

Note: As the requirements of the project focuses on predicting previous-to-next move patterns, the system is purposely biased towards the requirement by having an over-representation of schemes possessing the ability to detect previous-to-next move correlations in the swarm. The statistical and practical impact of such bias has not been thoroughly examined and will be outside the scope of the project.

Reasoning behind architecture decision:

From a statistical perspective, the advantage of using a swarm mind is two fold:

- 1) Increase confidence interval when picking the next move to be played.
- 2) Potential for emerging pattern detection abilities: by giving each scheme the ability to detect a specific but different type of pattern or learning formula, the combined system may detect emerging/non-hardcoded pattern not detectable by individual schemes.

The disadvantage of a swamp mind is the following:

- 1) a majority does not guarantee an adequate move and may be subject to majority bias. However, as mentioned earlier, this bias is fully attenuated since the system may pick the best performing individual scheme over the majority consensus under some circumstances.

Furthermore, in the case of when an overwhelmingly successful scheme is selected, the RPS system can also be viewed as a primitive-evolutionary algorithm in which the best strategy is picked while the least performing are relegated, but kept in the "gene pool" in case these genes become advantageous when facing future environmental changes.

Algorithm descriptions for each of the 15 algorithms:

Basic schemes:

- Scheme 1: Random play. No detection abilities nor adaptive abilities. Used as default when no consensus is reached and no scheme is overwhelmingly successful. Predictions made by scheme 1 are excluded from swarm majority voting.
- Scheme 2: Hardcoded psychological RPS play pattern of an average human player based on published research studies. No adaptive abilities.
- Scheme 3: Detects correlations between the opponent's last move and the new move. Results are further subdivided depending on either the opponent's last move was a Lost, a Draw or a Win for the opponent.

Advance schemes 1:

- Scheme 4-5: Same as scheme 3, but with different beta values for their respective learning formula.

- Adv1 Scheme 6-8: Detects correlations between the opponent's TWO last moves in identical fashion as scheme 3. Each scheme has different beta values for their respective learning formula and different play threshold values.
- Adv1 Scheme 9-10: Detects correlations between the opponent's last TWO moves. Results are however further subdivided depending on either the second last move was a Lost, a Draw or a Win for the opponent. Each scheme has different beta values for their respective learning formula and different play threshold values.
- Adv1 Scheme 11: Detects correlations between the opponent's three last moves in identical fashion as scheme 3.
- Adv1 Scheme 12: Detects correlations between the opponent's three last moves. Results are further subdivided depending on either the opponent's second last move was a Lost, a Draw or a Win for the opponent.
- Adv1 Scheme 13: Detects correlations between the opponent's three last moves. Results are however further subdivided depending on either the third last move was a Lost, a Draw or a Win for the opponent.
- Adv1 Scheme 14: Detects correlations between the opponent's four last moves in identical fashion as scheme 3.
- Schemed 15: Hardcoded pattern scheme for which basic scheme 3 sometimes has no counter.

*(Codes that are present but not activated) Adv2 Scheme 16-25: Detect raw Rock Paper Scissor patterns correlation between opponents from 1 up to 10 last moves. Some schemes are identical, but have different beta values and play minimum thresholds. **Due to time constraint and unsatisfactory performances in preliminary testings, these schemes have been omitted in the final design-segments of its codes can however still be seen in the program*******

2.1 White box implementation

2.1.0 Class DataStorage.h

Contains storage capability for -1, 0, +1 probabilities as well as get & set functions for them. Used by object of class outcomegen.

2.1.1 Class Outcomegen.h

Overview

The outcome generator is a pseudo hash-table with a 1-to-1 element-to-index correspondence.

The constructor takes the number of desire steps to look back. It then generates the set of all possible outcome combinations for the given data and steps by creating an appropriate data storage location in `vector std::vector<DataStorage*> possibilitystorage` of size `steps_back^3` using `vector push_back(new DataStorage)`.

As seen from the vector's type, each index of vector possibility storage contains a pointer to an object of class DataStorage which can store information about the odds for the given index.

Furthermore, each index of the vector is also abstractly associated with a unique outcome. Although not needed, each outcome is given an appropriate ID for clarity sake and while testing.

For example, for an outcomegen with parameter: 3 steps to hold type WLD (win-loss-draw) data, outcomegen will generate the set of all possible 3 steps win loss draw combinations with IDs set as follow:

www
wwl
wwd
wlw
wll
wld

...
Ddd

Using number theorem and base 3 number representations applied, each combination is associated with a unique (1-to-1) storage index in the following convention w/r = 0, l/p = 1, d/s = 2

For example,

outcome www will have index = $3^2*0 + 3^1*0 + 3^0*0 = 0$,
outcome wwl will have index = $3^2*0 + 3^1*0 + 3^0*1 = 1$
etc.

This will be discussed in greater details in Stat_Analysis.h section of this document.

Notes:

In the basic stat scheme, outcomegen is set to only look at 1 step back, thus generating the following set: w,l,d.

In advance stat scheme1 (composite scheme consisting of several sub-AIs), outcomegen is set to look up to 2-5 steps back.

OutcomeGenerator(int a) (constructors) & void generate(int
numberofsteps)

Possibilitystorage is first expanded to size `pow(3.0, numberofsteps to look back)`

using `push_back` and a for loop. This size corresponds to the total number of possibilities.

For loop with termination condition of `pow(3, number of step)` is then applied to set the id of each object pointed by the storage elements of possibilitystorage to reflect its appropriate index-outcome correspondance.

Due to its exponential nature, the maximum number of steps to look back is arbitrarily limited to 10 steps to avoid memory/cpu bottleneck through a throw exception in outcomegen's constructor.

Set and Get functions

As name indicates, get or set values at of a datastorage value through `->get` or `set` of `DataSource*`. Appropriate index need to be passed using the above index convention.

2.1.2History.h

Overview:

- Stores each move from beginning of the round. Automatically calculates win loss and stores it as well.
- Contains memory housekeeping fncs `clear()` and `clearEverythingbut(int)`: since it is highly unlikely that a scheme will need to look back at more than 10 moves back, `clearEverythingbut(10)` is periodically called by the player manager (`player1= true`) to clear all history except the last 10 moves.
- All values are stored as static since only 1 history is needed for every scheme. If `player_manager` plays against itself, the opposing player manager will invert the following storage parameters:
- Storage parameters:
`static std::vector<std::vector<int>> movehistory;`
`static std::vector<char> statushistoryS;`
`static std::vector<char> statushistoryO;`
`static int totalmoves;`
`static int winsS;`
`static int winsO;`
`static int draw;`

void newMove(int self, int opponent);

Takes 2 arguments: self and opponent. Arguments should follow the abstract convention:

0 = rock
1 = paper
2 = scissor

When passed, stores the moves in movehistory using push_back. Movehistory is a matrix vector of column size 2. First column = self, second = opponent (for player_manager = player1).

Function then subtract self from opponent and passes the result through a series of filters that determine whether it represents a win, a loss or a draw. Stores the results in

```
static std::vector<char> statushistoryS;  
static std::vector<char> statushistoryO;
```

Increment counters

```
static int winsS;  
static int winsO;  
static int draw;
```

Accordingly.

static void clear();

Clears the values of

```
static std::vector<char> statushistoryS;  
static std::vector<char> statushistoryO;
```

And movehistory by using vector.clear()

Does not reset counters.

static void clearEverythingBut(int value);

First stores the last "int value" index of vector<char> statushistoryS; vector<char> statushistoryO and movehistory in 3 temporary vectors
Then calls clear() to clear all storage.

Finally passes temporary vectors value back to their respective storage vectors.

int getSize();

Return the size of vector statushistoryS, which is equal to the size = statushistoryO and move history by using vector.size()

int compare(char x, int range2 = -1, int range = -2);

Compare the difference in move type between two index range2 and range1 of the same vector (statushistoryS or statushistoryO, char x represent self if == 's', opponent if == 'o' and dictates which vector is picked.)

Get the value at range2 by using H(x, range2, 0)

Get the value at range by using H(x, range, 0)

Calculates the difference between the two to determine the type

If difference < 0 or > 2, converts it back to appropriate rock, paper, scissor representation

Returns the difference.

int H(char x, int range, int type);

Has access to `statushistoryS` or `statushistoryO`, char `x` ('s' self, anything else = opponent) dictates which vector is picked.
Returns the move that would defeat the selected move at range `int range` (if `type = +1`), the move identical to the selected move at range (if `type = 0`), or the move that will lose to the the move located at `int range`.

First `Func` locate the appropriate index for the range (`Index = movehistory.size() + range`.
Note: range is negative.)

Then it picks the appropriate vectors (`statushistoryS` or `statushistoryO`) according to char `x`; gets the value; adds the value by `int type`. If difference `<0` or `> 2`, converts it back to appropriate rock, paper, scissor representation
Returns the difference.

`void getLastWinStatus(char x = 's');`

`Cout` the status of `statushistoryx[its.size()-1]`

`void convert(int &value);`

 If difference `<0` or `> 2`, converts it back to appropriate rock, paper, scissor representation
 `3 => 0, -1=> 2`

2.1.3 Stat_Analysis.h

Overview

`Stat_Analysis` is a derived class of `history` and has an `outcomegen(numberofsteps)` object. Its main purpose is to contain advance statistical tool to be used by all `stat` schemes and combine outcome prediction with history.

 Contains parameters:

`double beta;`

`int numberofsteps;`

`int comparewith;`

`Stat(int steps, double beta, int compare = -2)`

 When the constructor is called, it creates an `outcomegen` object initialized with `numberofsteps` as the object's number of steps to look back, the `double beta` is the `beta` used by learning formula (see `global variable.h` for more info) and `compare` value, which meaning will be discussed shortly below.

`size_t getIndexO(int mode, bool player1 = 1, std::string type = "WLD");`

 When this function is called, it returns the index of `history::vector<DataStorage*>` `possibilitystorage` that correctly correspond to the the win-loss-draw outcome

combination of the current round. The following example illustrate further this concept and how it operates:

If a scheme is looking 3 steps back, stat, using class outcomegen, will generate the set of all possible 3 steps win loss draw combinations as follow:

```

Www [index 0]
Wwl [index 1]
Wwd [index 2]
Wlw [etc.]
    wll
    wld

...
Ddd

```

Suppose the last 3 moves of the opponent were WLD, this member function's purpose is to locate the index of WLD in outcomegen's `possibilitystorage` vector.

The index of that move in outcomegen for WLD thus should be $3^2 \cdot 0$ (W) + $3^1 \cdot 1$ (L) + $3^0 \cdot 2$ (D) = 5 by convention where W = 0, L = 1, D = 2

Function finds said value using a for low with `* pow(3,1)` and a simple for loop.

(Ignore if type != 'WLD'. This is for advance scheme2, which is not implemented by player manager.)

If opponent plays a new move, sayd WLDD, where D is the new move, but player manager have yet to update stats for WLD and needs to find the index of WLD not of ..LDD, "mode"= 1 is then used to "shift" the last 3 moves by one.

`void updatestats(bool player1 = true, std::string type = "WLD")`

Takes the move being played last, compares it with the *comparewith*-th move played by the opponent (by default second last move) using function `history::compare(player, -1, comparewith)` discussed earlier. Update the stats the -1, 0, +1 odds of outcomegen's `possibilitystorage` vector. Uses the learning formula to find new odds, `size_t getIndexO()` to find the appropriate index to update and outcomegen's `set` function to modify the values of each -1,0, +1 odds.

2.1.4 Stat_AdvanceStat.h

Overview

Is a derived class of Stat_analysis. As such, it contains all the member variable and functions above. Also contains member function `getNext()/nextMove()` and the added parameters:


```

int minimumsteps;
int count = 0;
double threshold;
bool player1;
int compared_with;
double beta;

```

Threshold is the percentage above which an odd is deemed high enough to be used.

Note: Since Scheme_3 (stat) is essentially an Stat_advanceStat with int minimumsteps set to 1, this section will cover both schemes.

Constructor():

```

Scheme_AStats(bool player1, int minimumsteps, double threshold, double beta, int
comparewith) : Stat(minimumsteps, beta, comparewith), player1(player1),
minimumsteps(minimumsteps), threshold(threshold), beta(beta),
compared_with(comparewith) {};

```

Sets the value as above.

int getNext()

First checks whether its player1 playing or not. If not, history is reversed.

If the number of round played is below the minimumsteps (= to total numbers of steps to look back, returns -10 since outcomegen requires more data to be used.

Else{

Calls updatestates described in section above to update the -1,0,+1 odds for the given outcome last played.

Checks the odds for the current outcome.

Based on this, sees which outcome -1,0,+1 has the highest probability to be played next. Calls History::H(player1 or not, step to compare with, mostlikelyoutcome found) to get the moved most likely to be played by opponent next based on the most likely -1,0,+1. Returns the move that beats the last.

2.1.5 Scheme_2.h

Overview

Hard-coded scheme that looks at history.

Contains scheme_1 object random.

And history object history.

nextMove()

if (History.statushistoryO[History.statushistoryS.size() - 1] is a loss, then Opponent's next move= History.H(player, -1, -1);

else if (History.statushistoryO[History.statushistoryS.size() - 1] is a win then opponent's next move = History.H(player, -1, 1);

Else if a draw
return random

Return opponentNext + 1, use history.convert(next) described previously to convert to equivalent move if opponentNext + 1 is over 2 or under 0; return opponentNext + 1.

2.1.6 Scheme_1.h

nextMove()

Uses a random generator with uniform distribution to output a random move between 0 to 2 corresponding to rock, paper or scissor.

2.1.7 Scheme_4.h

nextMove()

Plays a simple hardcoded $(i*i)\%3$ scheme where i is an int initialized at 0 and is incremented after each subsequent turn. The resulting move is difficult for an AI opponent such as Stat (scheme 3) to predict and could occasionally prove to be useful to have in the swarm mind.

2.1.8 Player_Manager.h

Member parameters:

Has a vector of type pointer pure virtual rps_player to store each of the individual scheme

Has 3 vectors that keep track of the win, loss, draw odds respectively for each of the 15 schemes + 1 for the consensus

Has 3 int storage that keep track of the win, loss, draw odds of the last move picked by player_manager.

Constructor:

Sets the initial values of each schemes to arbitrary values as described in the opening paragraph. Sets the odds initial odds to 33.333%. Opens the above csv files to be written. Pushes the schemes vector scheme <RPS_Player*>

Overview/nextMove()

Has 15 schemes. Each out of the 15 total schemes generates 1 prediction per round (Rock, paper or scissor) and the most overwhelmingly common majority prediction is used as the next move to be played. In the absence of a majority consensus from the system OR if previous swarm consensus did not result in sufficient victories, the most overwhelmingly successful individual algorithm out of the 15 schemes is used to predict the next game move to be played. Lastly, in the absence of an overwhelmingly successful algorithm, a random move is picked to be played.

Additional criteria and details: the program also tracks the win/loss/draw odds of the moved being picked and if it's below *threshold* said by user or default value, consensus will not be achieved and random will be played.

Vector pointer pure virtual `rps_player` allows to call `nextMove()` on different derived scheme with the same code. Default move = `rand()`, passes a series of filter to see if criterias are met -> if yes, move is changed to reflect the filter's criteria.

`int Player_Manager::findBestScheme()`

Looks through the scheme with the highest winOdds and returns its index in scheme vector.

`updatestat()`

Compares the last move made by each scheme and the move played by the opponent to find if it's a loss-win-or-loss and then Update the Win-Loss-Draw odds of the respective scheme in vector "next" based on the learning formula*/

if a scheme could not return the best next move (confidence level of the scheme below given threshold, insufficient data acquired, etc. -10 is return as the next move

In this case: The move is arbitrarily consider a Draw but updated with a small beta of 0.02. This is because I want to discourage the system from using a successful but rarely used scheme

The first index of next ([0]) always correspond to the move of the swarm consensus and is not associated with an individual scheme*

Uses learning formula to update the appropriate states for each data parameters and odds vectors.

3.0 Output files and format & Tests

- Result.csv: base value is 0, incremented by 1 if wins, decremented by 1 if loses, stays the same if draws. Value is outputted each time player_manager uses nextMove()
- Consensus.csv: output '0' if no consensus reached this round, output 1 if consensus is reached
- ConsensusCount.csv: output number associated with each of the scheme used by the consensus. If no consensus, output -10 15times, once for each scheme
- IndividualScheme.csv: output '0' if no individual scheme picked. Output 1, if individual scheme was used this round
- IndividualSchemeCounter.csv: output number associated with the individual scheme used by this round. If no individual scheme used, output -10)
- RandomScheme.csv: output '0' if no individual scheme picked. Output 1, if individual scheme was used this round

Compended graph are used to display the wins vs schemes. Pareto graph are used to display consensus vote distribution Format will be explained.

*****Note that all the test cases can be repeated in the user interface and individual moves per round can be printed out upon user request.**

In test case1() Player Manager vs constant move

Player manager is picked against an opponent that only plays a randomly picked constant move 1000 times

A for loop with `int i = 0; i < 1000; i++` is used to generate said opponent. `i%3` is feed as the opponent.

Results:

```
Player_Manager won!W
Consensus total: 992
Individual schemes total: 7
D: 0
W: 998
L: 2
Destructor runs
Destructor runs
Destructor runs
```

d:\documents\visual studio 2015\Proje

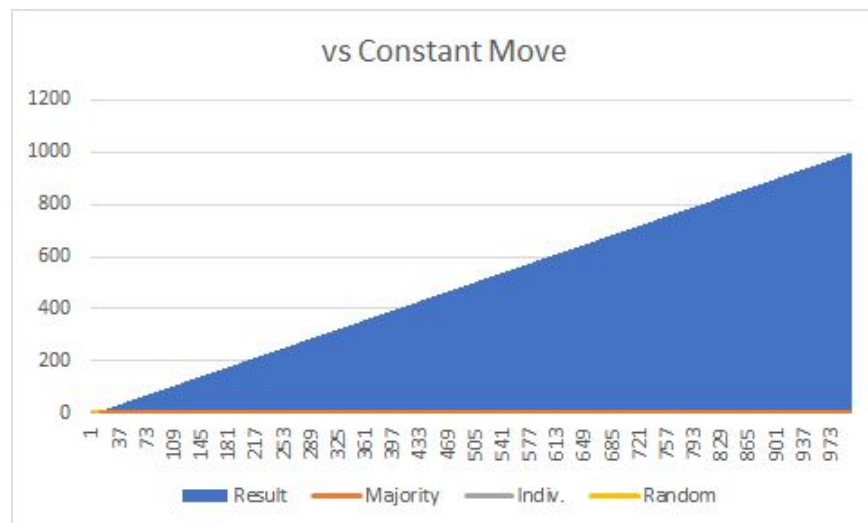
```
Round 993
Player_Manager plays: Rock
Opponent plays: Scissor
Player_Manager won!
```

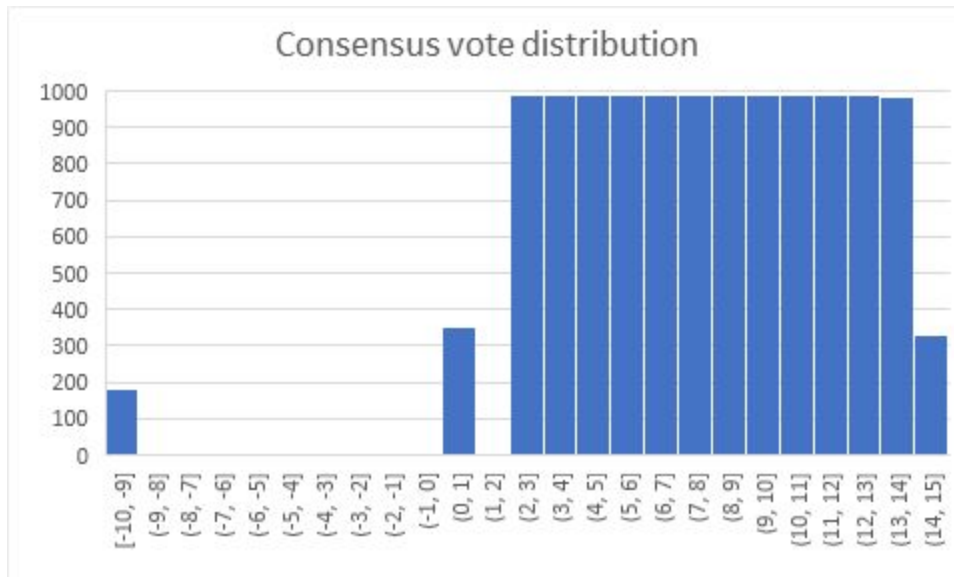
```
Round 994
Player_Manager plays: Rock
Opponent plays: Scissor
Player_Manager won!
```

```
Round 995
Player_Manager plays: Rock
Opponent plays: Scissor
Player_Manager won!
```

```
Round 996
Player_Manager plays: Rock
Opponent plays: Scissor
Player_Manager won!
```

```
Round 997
Player_Manager plays: Rock
Opponent plays: Scissor
Player_Manager won!
```






The results are conclusive. An unanimous consensus was reached 992 out of 1000 times (excluding random, AHP scheme and hardcoded scheme 15). From the last graph, one can see that:

- Scheme 1 (random) agreed with the consensus 33% of the time. This is to be expected as scheme 1 is random and an equal distribution of approximately 33.33% is expected from randomgen.
 - Scheme 2 (AHP) agreed with the consensus 0% of the time. This is to be expected as scheme 2 is hardcoded to expect the opponent to play H(player, -1, -1) when the opponent loses. As such, it is unsuited to play against an opponent that always plays the same move regardless of situations
 - Scheme 15 agreed with the consensus 30% of the time. This is also to be expected as scheme 15 is a hardcoded that returns $(i*i)\%3$
 - All stats schemed (3-14) were able to predict the pattern with unanimous and agree 99.9% and were effective at predicting this simple pattern.
- (note -10 represents the sum of all votes not used by player_manager and has no interesting significant in this case. [total = $(1000-992)*15$])

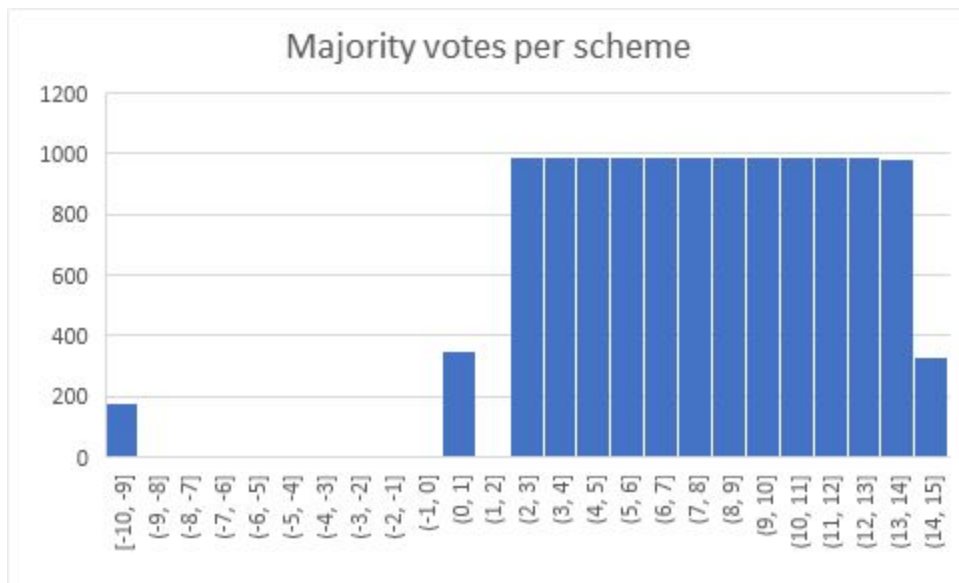
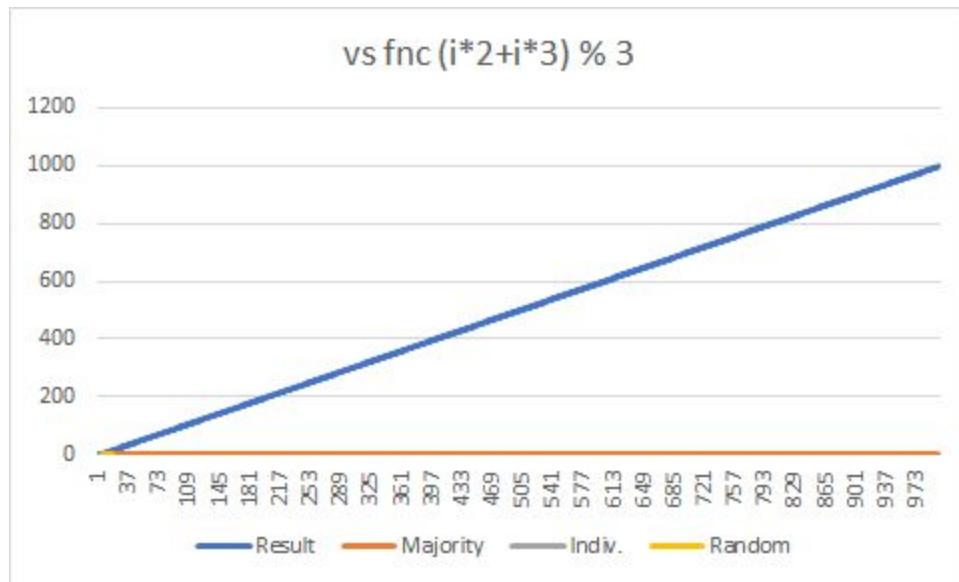
In test case2() Player Manager vs fnc $(i^2+i*3) \% 3$ where $i++$ after each round

Player manager is picked against an opponent that only plays a slightly more complex function

```
Individual scheme picked: 1  
Individual scheme picked: 15  
Individual scheme picked: 1  
Individual scheme picked: 3  
Individual scheme picked: 3  
Individual scheme picked: 3  
Consensus total: 990  
Individual schemes total: 6  
D: 4  
W: 996  
L: 0
```

 d:\documents\visual studio 2015\Projects

```
Round 993  
Player_Manager plays: Paper  
Opponent plays: Rock  
Player_Manager won!  
  
Round 994  
Player_Manager plays: Rock  
Opponent plays: Scissor  
Player_Manager won!  
  
Round 995  
Player_Manager plays: Scissor  
Opponent plays: Paper  
Player_Manager won!  
  
Round 996  
Player_Manager plays: Paper  
Opponent plays: Rock  
Player_Manager won!  
  
Round 997  
Player_Manager plays: Rock  
Opponent plays: Scissor  
Player_Manager won!  
  
Round 998  
Player_Manager plays: Scissor  
Opponent plays: Paper  
Player_Manager won!
```

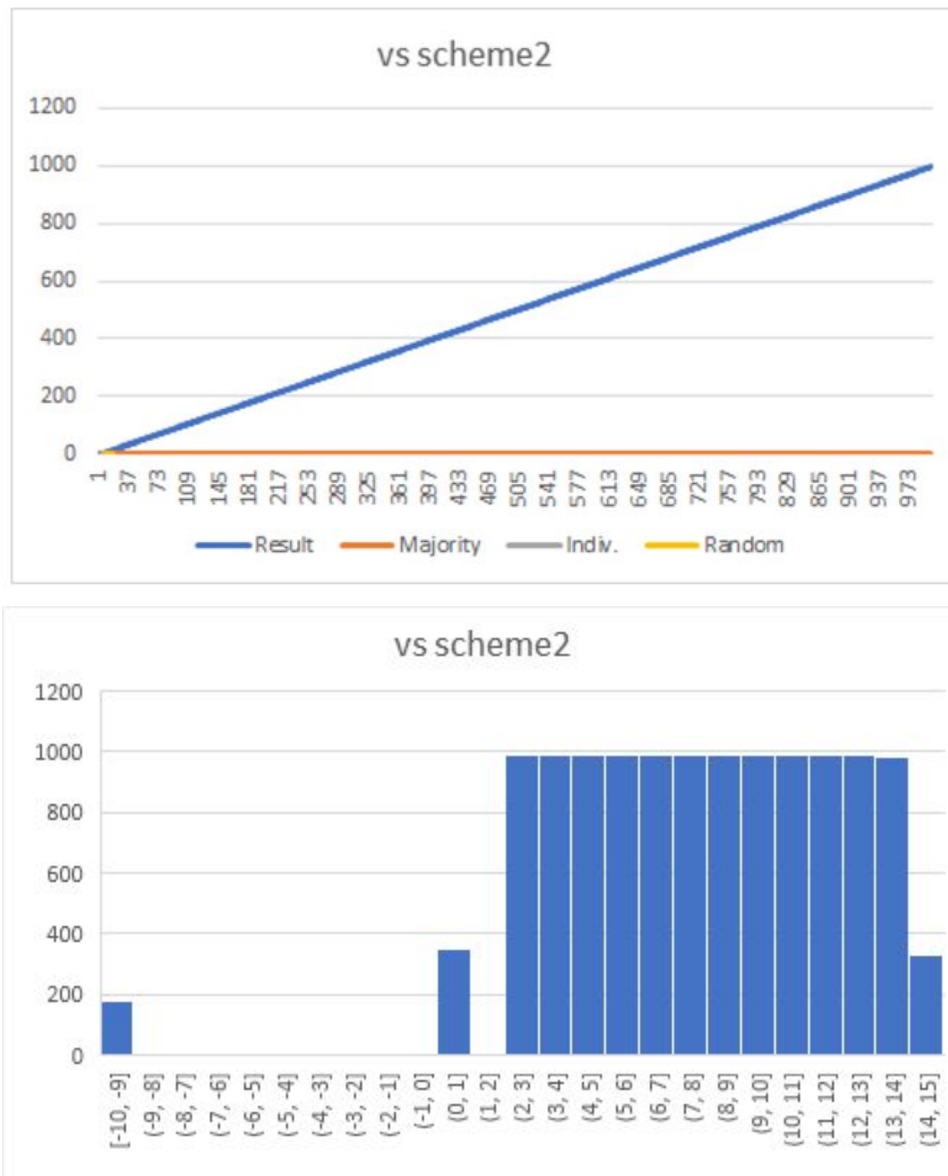


Similarly, the results are once again conclusive. An unanimous consensus was reached 996 out of 1000 times (excluding random, AHP scheme and hardcoded scheme 15). An interesting observation is that all stat scheme (scheme 3-14) were able to detect the pattern with 100% accuracy and precision despite the increase in pattern complexity of the opponent, giving the program a high degree of confidence in its predictions.

The same experiment can be repeated against several different function of similar nature with similar results with the exception of some exponentials or division functions involving number

rounding (acts as a random generator in this case since int will be “randomly” rounded up or down depending on certain situations). This success in pattern analysis will be shown during the live demonstration of the program for the marker.

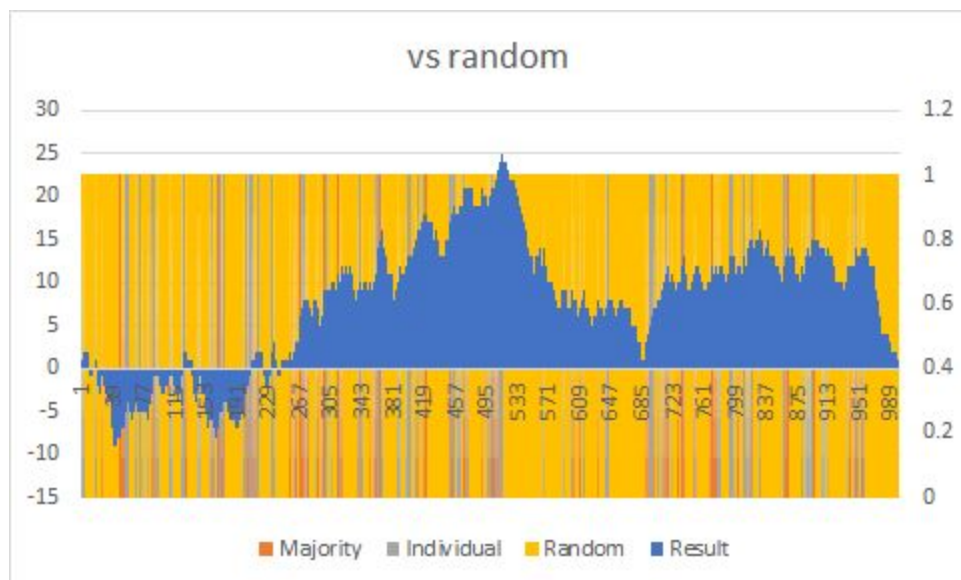
In test case3() Player Manager vs scheme_2 (AHP)



Similar results. Upon further investigating individual moves (accessible through the user interface), one can see that scheme_2 will eventually be stuck in a single move pattern - playing the same move over and over again against player_manager, which takes advantage of this.

In test case4() Player Manager vs random (scheme_1)

```
Consensus total: 95
Individual schemes total: 173
D: 323
W: 338
L: 339
```



The result is unsurprising: more than 80% of the time, random was played and the program ended in a draw against random. This shows that player_manager fairs very well against an unpredictable opponent and will play random against an opponent that does the same.

In test case5() Player Manager vs complex hardcoded pattern

In this test, player_manager is pitted against the following code

```
For (int i = 0; i < 1000; i++){
    If (i%3== 0){
        Opponent = (1000-i)%3;
    }
    Else if (i%3 == 1){
        Opponent = i%3;
    }
    Else{
        Opponent = 0;
    }
}
```

}

```
Individual scheme picked: 13  
Individual scheme picked: 13  
Consensus total: 2  
Individual schemes total: 993  
D: 3  
W: 994  
L: 3
```

```
Individual scheme picked: 13  
Individual scheme picked: 13  
Consensus total: 1  
Individual schemes total: 960  
D: 15  
W: 969  
L: 16
```

 d:\documents\visual studio 2015\Projects\Final project\C

Round 995

Player_Manager plays: Paper

Opponent plays: Rock

Player_Manager won!

Individual scheme picked: 13

Round 996

Player_Manager plays: Scissor

Opponent plays: Paper

Player_Manager won!

Individual scheme picked: 13

Round 997

Player_Manager plays: Scissor

Opponent plays: Paper

Player_Manager won!

Individual scheme picked: 13

Round 998

Player_Manager plays: Paper

Opponent plays: Rock

Player_Manager won!

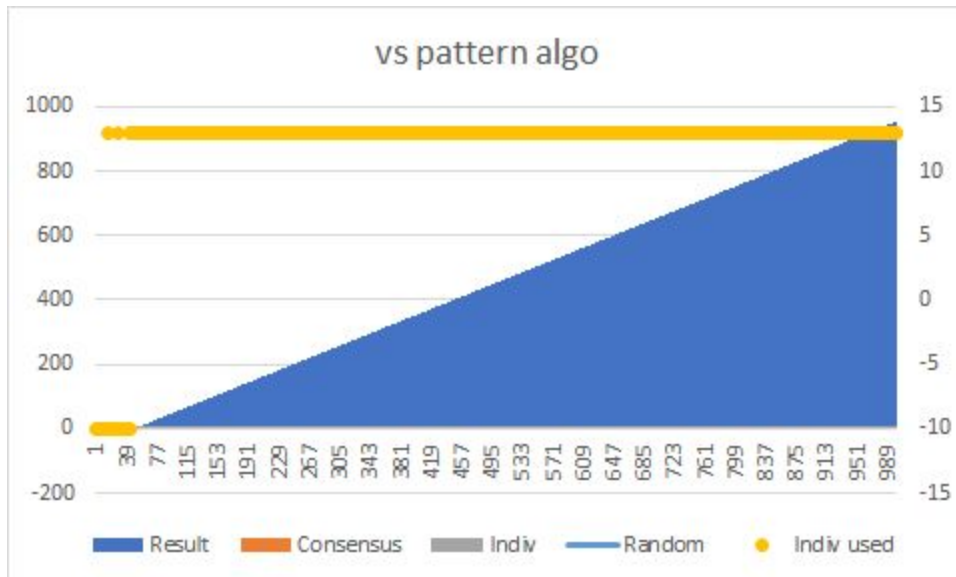
Individual scheme picked: 13

Round 999

Player_Manager plays: Scissor

Opponent plays: Paper

Player_Manager won!



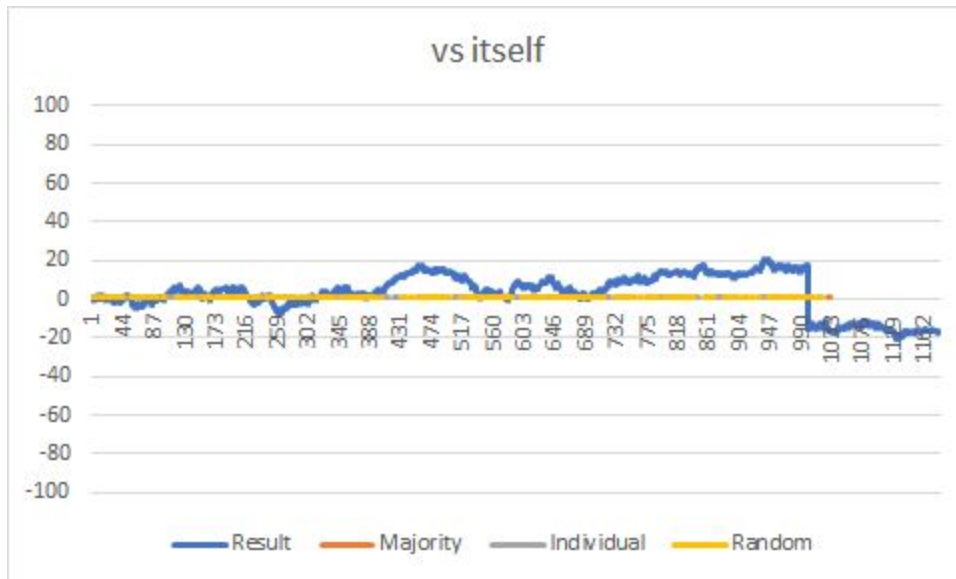
This example showcases the versatility of the advance stats schemes. In this case, as seen in yellow in the graph above, scheme 13 has proven to be overwhelmingly successful in detecting the above pattern:

Recall that Scheme 13 looks 3 moves back and compares the results with the last 3rd last step played by the opponent.

It just so happens that opponent generates a play pattern that follows closely this scheme detection range. Without it, player_manager would not had been detected and win against this opponent.

In test case6() Player Manager vs itself

```
Consensus total: 65
Individual schemes total: 162
D: 343
W: 319
L: 338
```



When playing itself, both player manager end up resorting to playing random and neither has the edge over the other. One thing that could be further explored but wasn't due to time constraint is the the effect on changing various beta in both player managers.

Note on tests codes

Please note all the test cases can be accessed at will in the interactive menu in the program walks. Additional testing can be conducted through the user interface which enables the user to play against ALL individual scheme_2 (AHP) and player_manager(swarm AI). Please refer to UI for more details on the tests results and select option 0 afterwards to see individual results for each move.

Testcodes is implemented as follows in header file Test_Cases.h

It tests several scenarios and allows for the user to play directly against player manager and scheme AHP. As seen from the csv graphs, the code performs exactly as expected and can detect, adapt and prevail against a wide range of patterns of increasing complexity with great success.

```
void case1(bool value, int choice) {
    int opp;
    if (choice == 1) {
        opp = rand() % 3;
    }
}
```

```
Scheme_2 B(false);
Scheme_1 C;
Player_Manager D(false);
Scheme_2 E(true);
```

```
int win = 0;
int loss = 0;
int draw = 0;
Player_Manager A;
History History;
bool print = value;
if (choice == 11) {
    History.newMove(0, 0);
}
for (int i = 0; i < 1000; i++) {
    int self = 0;
    if (choice != 11) {
        self = A.nextMove();
    }

    if (choice == 2) {
        opp = (i*2+i*3) % 3;
    }
    if (choice == 3) {
        opp = B.nextMove();
    }
    if (choice == 4) {
        opp = C.nextMove();
    }
    if (choice == 5) {

        if (i % 3 == 0) {
            opp = (1000 - i) % 3;
        }
        else if (i % 3 == 1) {
            opp = i % 3;
        }
        else {
            opp = 0;
        }
    }
    if (choice == 6) {
```

```

        opp = D.nextMove();
    }
    if (choice == 10) {
        cout << "\nplease enter your move [0 - Rock] [1 - Paper] [2- scissor] [-1 to
Exit]: ";

        while (!(cin >> opp))
        {
            throw "exception";
        }
        while (opp > 2 || opp < -1) {
            cout << "\nPlease enter a valide move";
            cin >> opp;
        }
        std::cout << "\nD: " << draw << "\nW: " << win << "\nL: " << loss <<
std::endl;
    }

    if (choice == 11) {
        self = E.nextMove();
        cout << "\nplease enter your move [0 - Rock] [1 - Paper] [2- scissor] [-1 to
Exit]: ";

        while (!(cin >> opp))
        {
            throw "exception";
        }
        while (opp > 2 || opp < -1) {
            cout << "\nPlease enter a valide move";
            cin >> opp;
        }
        std::cout << "\nD: " << draw << "\nW: " << win << "\nL: " << loss <<
std::endl;
    }

    if (opp == -1) {
        break;
    }

    History.newMove(self, opp);
    if (print == true) {
        std::cout << "\n\nRound " << i;
    }
    if (self == 0 && print == true) {

```



```

        std::cout << "\nPlayer_Manager plays: Rock";
    }
    else if (self == 1 && print == true) {
        std::cout << "\nPlayer_Manager plays: Paper";
    }
    else if (print == true) {
        std::cout << "\nPlayer_Manager plays: Scissor";
    }

    if (opp == 0 && print == true) {
        std::cout << "\nOpponent plays: Rock";
    }
    else if (opp == 1 && print == true) {
        std::cout << "\nOpponent plays: Paper";
    }
    else if (print == true) {
        std::cout << "\nOpponent plays: Scissor";
    }

    if (History.statushistoryO[History.getSize() - 1] == 'D') {
        draw++;
        if (print == true) {
            std::cout << "\nIt's a draw!";
        }
    }
    else if (History.statushistoryO[History.getSize() - 1] == 'L') {
        win++;
        if (print == true) {
            std::cout << "\nPlayer_Manager won!";
        }
    }
    else if (History.statushistoryO[History.getSize() - 1] == 'W') {
        loss++;
        if (print == true) {
            std::cout << "\nPlayer_Manager lost!";
        }
    }

}

std::cout << "\nConsensus total: " << A.consensusCounter;
std::cout << "\nIndividual schemes total: " << A.individualSchemeCounter;
std::cout << "\nD: " << draw << "\nW: " << win << "\nL: " << loss << std::endl;

```

}