```cpp
#include <iostream>
#include "game_assets/game.h"
#include "ai/aiRandom.h"
#include "ai/aiOwn.h"
#include <thread>
#include <vector>
#include "random.h"
#include "hand_eval/2+2/2P2Evaluator.h"
#include <sys/time.h>
#include <random>
#include "hand_eval/handEvalCalc.h"
#include "ai/aiCaller.h"
#include "ai/aiFolder.h"
#include "ai/aiRaiser.h"

bool sScoreGreaterThan(const std::pair<long,long>& a, const std::pair<long,long>& b) {
    return a.first > b.first;
}

bool sMoneyGreaterThan(const std::pair<double,long>& a, const std::pair<double,long>& b) {
    return a.first > b.first;
}

inline double my_clock(void) {
    struct timeval t;
    gettimeofday(&t, NULL);
    return (1.0e-6*t.tv_usec + t.tv_sec);
}

// Forward Declaration
Rules* defineRules();

std::vector<Player*> runOneGeneration(std::vector<Player*> &players, Rules* rules, int numElitePlayersToKeep,
                                      int numTourney, std::vector<Player*> &hallOfFame);
std::vector<Player*> runOneGenerationWithMultiThreading(std::vector<Player*> players, int numElitePlayersToKeep,
                                                        int numTourney, int numThreads);
std::vector<long> startOneGameCicle(Game *game, std::vector<std::vector<long>> &resultVector, int id);
void evolvePlayers(std::vector<Player*> players, const std::vector<Player*> elitePlayers, double mutationLikelihood);
bool saveNeuralNetworkWeightsToFile(std::vector<double> weights);
bool loadNeuralNetworkWeightsIntoArray();
void initializeHallOfFame(std::vector<Player*> &players, std::vector<Player*> &hallOfFame);
void initializeHallOfFameWithOwnAi(std::vector<Player*> &players, std::vector<Player*> &hallOfFame);
void updateHallOfFame(std::vector<Player*> &players, std::vector<Player*> elite, std::vector<Player*> &hallOfFame);

```

```
45 double weightsLoaded[NUM_OF_NN_WEIGHTS];
46 int generation = 0;
47
48 int NUM_AI_AGENTS = 52;
49 int NUM_HALL_OF_FAME = GLOBAL_NUM_PLAYERS - NUM_AI_AGENTS;
50
51 int NUM_AI_FOLD = 12;
52 int NUM_AI_CALL = 8;
53 int NUM_AI_RAISE = 7;
54 int NUM_AI_RANDOM = 7;
55
56
57 int NUM_ELITE_PLAYERS = 5;  // TODO Some calculations in the evolve methode use hardcoded values so change it
58 std::vector<unsigned> topo {16,8,3};
59
60 int numOfGenerations = 500;
61 int numOfTournaments = 100;
62
63 int main() {
64
65     ////////////// SETUP ///////////////
66
67     seedRandomFastWithRandomSlow();
68     // Setup 2+2 method
69     TPTEvaluator::initEvaluator();
70
71     //Load Preflop Lookup Tables
72     TPTEvaluator::loadPreflopTables();
73
74     /////////// PLAYER CREATION ///////////
75
76     // Creating the players for the first generation
77     std::vector<Player*> players;
78     std::vector<Player*> hallOfFame;
79
80     for (int i = 0; i < NUM_AI_AGENTS; i++) {
81         Player *player;
82         AIOwn *ai = new AIOwn();
83         ai->setTopology(topo);
84         player = new Player(ai);
85         player->setID(i);
86         players.push_back(player);
87     }
88
```

```
 89     //initializeHallOfFame(players, hallOfFame);
 90     initializeHallOfFameWithOwnAi(players, hallOfFame);
 91
 92     ///////////////// RUN GAMES /////////////////
 93     double start_time, end_time;
 94     start_time = my_clock();
 95
 96     for (int i = 0; i < numOfGenerations; i++) {
 97
 98         // Setup the rules
 99         Rules* rules = defineRules();
100
101         // Keep x% of the best players
102         int numOfElitePlayersToKeep = NUM_ELITE_PLAYERS;
103
104         // elite players per generation
105
106         std::vector<Player*> elite;
107         // Run one Generation
108         elite = runOneGeneration(players, rules, numOfElitePlayersToKeep, numOfTournaments, hallOfFame);
109
110         // DO THE EVOLUTION
111         double mutationLikelihood = 0.10;
112
113         std::vector<Player*> playersToEvolve(players.begin(), players.begin() + NUM_AI_AGENTS);
114         evolvePlayers(playersToEvolve, elite, mutationLikelihood);
115
116         saveNeuralNetworkWeightsToFile(dynamic_cast<AIOwn *>(elite.at(0)->getAI())->net_.getOutputWeights());
117         //saveNeuralNetworkWeightsToFile(dynamic_cast<AIOwn *>(players.at(0)->getAI())->net_.getOutputWeights());
118         //loadNeuralNetworkWeightsIntoArray(); //This fills the global array
119
120         delete rules;
121         std::cout << "GENERATION-NR: " << i << " DONE" << std::endl;
122         generation++;
123     }
124
125     end_time = my_clock();
126     double totalTime = end_time - start_time;
127     std::cout << "Full " << numOfGenerations << " generations took: " << totalTime  << "seconds" << std::endl;
128
129     return 0;
130 }
131
132 void evolvePlayers(std::vector<Player*> players, const std::vector<Player*> elitePlayers, double mutationLikelihood) {
```

```
133     // Should I use this as generator setup?
134     //std::random_device rd;  //Will be used to obtain a seed for the random number engine
135     //std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
136
137     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
138     std::default_random_engine generator(seed);
139     std::discrete_distribution<int> distribution {2,10,4,2,2}; //TODO change this values when elite number players changes
140     std::uniform_int_distribution<int> uniformDistribution(0,4); //TODO change this values when elite number players changes
141     std::uniform_real_distribution<> uniformRealDistribution(0,1);
142     std::normal_distribution<double> normalDistribution(0, 0.1);
143
144     //Before creating children save the elite players at the beginning of the players vector and
145     // swap them with the current players there so you just mutate the rest
146     std::vector<Player*> tempPlayers;
147
148     for (auto &elitePlayer: elitePlayers) {
149         tempPlayers.push_back(elitePlayer);
150     }
151
152     for (auto &player: players) {
153         if(std::find(tempPlayers.begin(), tempPlayers.end(), player) == tempPlayers.end())
154         {
155             //player does not exist in tempPlayers, add him
156             tempPlayers.push_back(player);
157         }
158     }
159
160     //What if elitePlayers count = 1?
161     for (long p = elitePlayers.size(); p < tempPlayers.size(); p++) {
162         // get random number of parents
163         // should be an exponential distribution so that 2 is most likely but 1 and more is also likely
164         //TODO look this up how it should be
165         // for now i just hardcode my values
166         int parentCount = distribution(generator) + 1; // because range is 0...3
167         std::vector<Player *> parents;
168         std::vector<double> parentsEvolutionWeights;
169
170
171         for (int i = 0; i < parentCount; i++) {
172             //TODO should the same player be able to be selected twice as parent?
173             //TODO In the paper they are not allowed to be selected twice
174             int index = uniformDistribution(generator);
175             parents.push_back(elitePlayers.at(index));
176
```

```
177             double weight = uniformRealDistribution(generator);
178             parentsEvolutionWeights.push_back(weight);
179         }
180 
181         //normalize weights
182         double sum = 0.0;
183         for (auto &weight: parentsEvolutionWeights)
184             sum += weight;
185 
186         for (auto &weight: parentsEvolutionWeights)
187             weight = weight / sum;
188 
189         // for each link of the neural network calculate the new value
190         std::vector<double> childWeights = dynamic_cast<AIOwn *>(tempPlayers.at(p)->getAI())->net_.getOutputWeights();
191 
192         for (int w = 0; w < childWeights.size(); w++) {
193             double value = 0.0;
194             for (int par = 0; par < parents.size(); par++) {
195                 std::vector<double> parentWeights = dynamic_cast<AIOwn *>(parents.at(
196                         par)->getAI())->net_.getOutputWeights();
197                 value += parentWeights[w] * parentsEvolutionWeights[par];
198             }
199 
200             //add random noise with a likelihood
201             double random = uniformRealDistribution(generator);
202             if (random < mutationLikelihood) {
203                 //Choose a normal distribution with mean = 0 and std = 0.1
204                 // This promotes many small changes as opposed to a few large changes
205                 // TODO Check if this works by calculating it by hand and breakpoint
206                 // TODO if the values are too big
207                 double noiseValue = normalDistribution(generator);
208                 //TODO check if noise would yiels > 1 or < -1 value, then just add enough to cap it
209                 value += noiseValue;
210             }
211             childWeights.at(w) = value;
212         }
213 
214         // give child the new weights
215         dynamic_cast<AIOwn *>(tempPlayers.at(p)->getAI())->net_.setOutputWeights(childWeights);
216     }
217 }
218 
219 std::vector<Player*> runOneGeneration(std::vector<Player*> &players, Rules* rules, int numElitePlayersToKeep,
220                                       int numTourney, std::vector<Player*> &hallOfFame) {
```

```
221
222     // Create a game
223     Game game;
224     game.setRules(rules);
225
226     // Add players to the game
227     for (auto &player: players) {
228         game.addPlayer(player);
229     }
230
231     double start_time, end_time;
232     start_time = my_clock();
233
234     // Play #num of games with the same agents
235     for (long i = numTourney; i > 0; i--) {
236         game.playGame();
237         game.cleanUpGame();
238         std::cout << "game " << i << " finsihed!" << std::endl;
239     }
240     end_time = my_clock();
241     double time = end_time - start_time;
242     std::cout << numTourney << " games took: " << time << std::endl;
243
244
245     ///////////////// EVALUATE FITNESSFUNCTIONS //////////////
246
247     /// 1.) AVERAGE PLACEMENT PLACED
248     // Sort players by performance
249     std::vector<std::pair<long,long>> gameResultsPaired;
250     std::vector<long> gameResults = game.getOverallGameResults();
251     for (int i = 0; i < gameResults.size(); i++) {
252         gameResultsPaired.push_back(std::make_pair(gameResults[i], i));
253     }
254
255     // now gameResultsPaired holds an ascending list of pairs - pair.first = value, pair.second = index
256     std::sort(gameResultsPaired.begin(), gameResultsPaired.end());
257
258     float total = 0.0;
259     for (auto &pair : gameResultsPaired) {
260         std::cout << "Player " << pair.second << " average place:  " << pair.first / (float)numTourney << std::endl;
261         total +=  pair.first / (float)numTourney;
262     }
263     std::cout << "Total: " << total << std::endl;
264
```

```
265     /// 2.) HANDS WON PLACED
266     std::vector<std::pair<long,long>> handsWonPaired;
267     std::vector<long> handsWonResults = game.getOverallHandsWonResults();
268     for (int i = 0; i < handsWonResults.size(); i++) {
269         handsWonPaired.push_back(std::make_pair(handsWonResults[i], i));
270     }
271 
272     // now handsWonResults holds a list where .second is the playerID and .first is his hands won
273     std::sort(handsWonPaired.begin(), handsWonPaired.end(), sScoreGreaterThan);
274 
275     float totalHandsWon = 0.0;
276     for (auto &pair : handsWonPaired) {
277         std::cout << "Player " << pair.second << " hands won:  " << pair.first << std::endl;
278         totalHandsWon +=  pair.first;
279     }
280     std::cout << "Total hands won by all players: " << totalHandsWon << std::endl;
281 
282     /// 3.) MEAN MONEY WON
283     std::vector<std::pair<double,int>> meanMoneyWonPaired;
284     std::vector<double> meanMoneyWonResults = game.getMeanMoneyWonResults();
285     for (int i = 0; i < meanMoneyWonResults.size(); i++) {
286         meanMoneyWonPaired.push_back(std::make_pair(meanMoneyWonResults[i], i));
287     }
288 
289     std::sort(meanMoneyWonPaired.begin(), meanMoneyWonPaired.end(), sMoneyGreaterThan);
290 
291     for (auto &pair : meanMoneyWonPaired) {
292         std::cout << "Player " << pair.second << " mean money won:  " << pair.first << std::endl;
293     }
294 
295     /// COMBINE ALL 3 TOGETHER
296     std::vector<std::pair<double,int>> placeAndHandsWonPaired;
297 
298     for (int p = 0; p < gameResultsPaired.size(); p++) { // This ranks them according to their fitness
299         gameResultsPaired[p].first = p + 1;
300         handsWonPaired[p].first = p + 1;
301         meanMoneyWonPaired[p].first = p + 1;
302     }
303 
304     for (int i = 0; i < gameResultsPaired.size(); i++) {
305         for (int j = 0; j < handsWonPaired.size(); j++ ) {
306             for (int k = 0; k < meanMoneyWonPaired.size(); k++) {
307                 int index1 = gameResultsPaired[i].second;
308                 int index2 = handsWonPaired[j].second;
```

```
309                 int index3 = meanMoneyWonPaired[k].second;
310
311                 if (index1 == index2 && index2 == index3) {
312                     double fitnessScore = gameResultsPaired[i].first * FITNESS_WEIGHT_1 +
313                                           handsWonPaired[j].first * FITNESS_WEIGHT_2 +
314                                           meanMoneyWonPaired[k].first * FITNESS_WEIGHT_3;
315
316                     placeAndHandsWonPaired.push_back(std::make_pair(fitnessScore, index1));
317                 }
318             }
319         }
320     }
321     // sort descendingly (best player, lowest score)
322     std::sort(placeAndHandsWonPaired.begin(), placeAndHandsWonPaired.end());
323
324
325     for (auto &pair : placeAndHandsWonPaired) {
326         std::cout << "Player " << pair.second << " fitnessScore:  " << pair.first << std::endl;
327         totalHandsWon +=  pair.first;
328     }
329
330     ////////////// GET BEST PLAYERS OF OWN-AI ////////////////
331     // keep #num of elite players and return them
332     std::vector<Player*> elitePlayers;
333     elitePlayers.resize(numElitePlayersToKeep);
334     int eliteIndex = 0;
335     int hallOfFameIndex = 0;
336     for (int i = 0; i < players.size(); i++) {
337         //Set players overall ranking
338         players[placeAndHandsWonPaired[i].second]->overallRanking_ = i + 1;
339
340         //Check if agant is hallOfFame member or not
341         // If it is sort the hallOfFame
342         if (!players[placeAndHandsWonPaired[i].second]->hallOfFameMember_ && eliteIndex < NUM_ELITE_PLAYERS) {
343             elitePlayers.at(eliteIndex) = players[placeAndHandsWonPaired[i].second];
344             eliteIndex++;
345         } else if (players[placeAndHandsWonPaired[i].second]->hallOfFameMember_){ // This sorts the hallOfFame
346             hallOfFame.at(hallOfFameIndex) = players[placeAndHandsWonPaired[i].second];
347             hallOfFameIndex++;
348         }
349     }
350
351     //UPDATE HALL OF FAME
352     updateHallOfFame(players, elitePlayers, hallOfFame);
```

```
353
354     // RESETTING THE GAME STATISTICS AFTER EACH GENERATION
355     game.resetStatistics();
356
357     // Returning the top players that we want to keep
358     return elitePlayers;
359 }
360
361 void initializeHallOfFame(std::vector<Player*> &players, std::vector<Player*> &hallOfFame) {
362
363     for (int i = NUM_AI_AGENTS; i < NUM_AI_AGENTS + NUM_AI_RANDOM; i++) {         // random
364         Player *player;
365         player = new Player(new AIRandom());
366         player->setID(i);
367         player->hallOfFameMember_ = true;
368         players.push_back(player);
369         hallOfFame.push_back(player);
370     }
371
372     for (int i = NUM_AI_AGENTS + NUM_AI_RANDOM; i < NUM_AI_AGENTS + NUM_AI_RANDOM + NUM_AI_CALL; i++) {         // call
373         Player *player;
374         player = new Player(new AICall());
375         player->setID(i);
376         player->hallOfFameMember_ = true;
377         players.push_back(player);
378         hallOfFame.push_back(player);
379     }
380
381     for (int i = NUM_AI_AGENTS + NUM_AI_RANDOM + NUM_AI_CALL; i <
382             NUM_AI_AGENTS + NUM_AI_RANDOM + NUM_AI_CALL + NUM_AI_RAISE; i++) {         // raiser
383         Player *player;
384         player = new Player(new AIRaiser());
385         player->setID(i);
386         player->hallOfFameMember_ = true;
387         players.push_back(player);
388         hallOfFame.push_back(player);
389     }
390
391     for (int i = NUM_AI_AGENTS + NUM_AI_RANDOM + NUM_AI_CALL + NUM_AI_RAISE; i <
392             NUM_AI_AGENTS + NUM_AI_RANDOM + NUM_AI_CALL + NUM_AI_RAISE + NUM_AI_FOLD; i++) {         // fold
393         Player *player;
394         player = new Player(new AIFold());
395         player->setID(i);
396         player->hallOfFameMember_ = true;
```

```
397         players.push_back(player);
398         hallOfFame.push_back(player);
399     }
400 }
401
402 void initializeHallOfFameWithOwnAi(std::vector<Player*> &players, std::vector<Player*> &hallOfFame) {
403
404     for (int i = NUM_AI_AGENTS; i < GLOBAL_NUM_PLAYERS; i++) {         // random
405         AIOwn *ai = new AIOwn();
406         Player *player;
407         player = new Player(ai);
408         player->setID(i);
409         player->hallOfFameMember_ = true;
410         players.push_back(player);
411         hallOfFame.push_back(player);
412     }
413 }
414
415 void updateHallOfFame(std::vector<Player*> &players, std::vector<Player*> elite, std::vector<Player*> &hallOfFame) {
416     // Check if elite players performed better than worst hallOfFamePlayers, if so replace them
417     // Clone the player that replaces the worst player and set its id to the replaced player's id.
418     // Also set the hallOfFame flag to TRUE
419     // also don't forget to replace the player in the players vector
420     int j = 0;
421     for (int i = 0; i < elite.size(); i++) {
422         if (hallOfFame.at(NUM_HALL_OF_FAME - elite.size() + i)->overallRanking_ > elite.at(j)->overallRanking_) {
423             // Copy elite player and replace it with current hallOfFame player
424             Player *copyElitePlayer = new Player(*elite.at(j));
425
426             // set id to old players id and set the hallOfFame flag
427             copyElitePlayer->setID(hallOfFame.at(NUM_HALL_OF_FAME - elite.size() + i)->getID());
428             copyElitePlayer->hallOfFameMember_ = true;
429             hallOfFame.at(NUM_HALL_OF_FAME - elite.size() + i) = copyElitePlayer;
430             players.at(copyElitePlayer->getID()) = copyElitePlayer;
431             j++;
432         }
433     }
434 }
435
436 Rules* defineRules() {
437     // Set the rules
438     Rules *rules = new Rules();
439     rules->buyIn_ = 1500;
440     rules->blindLevel_ = BlindLevel(10, 20, 3);
```

```
441
442     // How many players do compeat in the game
443     rules->totalNumberOfPlayers_ = GLOBAL_NUM_PLAYERS;
444
445     // How many players should participate?
446     // 9 players per table are allowed
447     rules->maxNumPlayersPerTable_ = 9;
448
449     return rules;
450 }
451
452 bool saveNeuralNetworkWeightsToFile(std::vector<double> weights) {
453     ////////////////// SAVING //////////////////////////
454     // save weights array to file
455
456     double weightsArray[NUM_OF_NN_WEIGHTS] = {0};
457     for (int i = 0; i < weights.size(); i++) {
458         weightsArray[i] = weights.at(i);
459     }
460     std::string filename = "../LookupTables/nn_weightsGen" + std::to_string(generation) + ".dat";
461     FILE * fout = fopen(filename.c_str(), "wb");
462     if (!fout) {
463         printf("Problem creating the Output File!\n");
464         return 1;
465     }
466     fwrite(weightsArray, sizeof(weightsArray), 1, fout);  // big write, but quick
467
468     fclose(fout);
469     return true;
470 }
471
472 bool loadNeuralNetworkWeightsIntoArray() {
473     printf("Loading nn_weights.DAT file...");
474     memset(weightsLoaded, 0, sizeof(weightsLoaded));
475     FILE * fin = fopen("../LookupTables/nn_weights.dat", "rb");
476     if (!fin)
477         return false;
478     size_t bytesread = fread(weightsLoaded, sizeof(weightsLoaded), 1, fin); // get the HandRank Array
479     fclose(fin);
480     printf("complete.\n\n");
481     return true;
482 }
483
484
```

```
485 /// MULTITHREADED
486
487 std::vector<Player*> runOneGenerationWithMultiThreading(std::vector<Player*> players, int numElitePlayersToKeep, int numTourney, int
    numThreads) {
488
489     std::vector<std::vector<long>> resultVector;
490     resultVector.resize(numTourney);
491
492     // Create a game
493     std::vector<Game *> gameVector;
494     std::vector<Rules *> rulesVector;
495
496     for (int i = 0; i < numTourney; i++) {
497         Game *game = new Game();
498         gameVector.push_back(game);
499         Rules *rules = defineRules();
500         rulesVector.push_back(rules);
501         game->setRules(rules);
502     }
503
504     // Add players to the game
505     // Be patient while multithreading... we can't give the players vector to all threads because they would
506     // manipulate all the pointers simultaniously!
507     // SOLUTION: we want to create a new players pointer vector for each game and also new AI but copy the values of
508     // the old ones -> copy constructor
509     for (auto &game: gameVector) {
510         for (auto &player: players) {
511             Player *newPlayer = new Player(*player); //Copy the players
512             newPlayer->setID(player->getID());
513             game->addPlayer(newPlayer);
514             newPlayer->getAI()->setName("New");
515         }
516     }
517
518     double start_time, end_time;
519     start_time = my_clock();
520
521     // Play #num of games with the same agents
522     std::vector<std::thread> t_vector;
523     int numGames = 0;
524     while (numGames < numTourney) {
525         for (int i = 0; i < numThreads; i++) {
526             t_vector.push_back(std::thread(startOneGameCicle, gameVector[numGames], std::ref(resultVector), numGames));
527             numGames++;
```

```
528         }
529
530         for (auto &thread: t_vector) {
531             thread.join();
532         }
533         if (numGames + t_vector.size() > numTourney) {
534             numThreads = numTourney - numGames;
535         }
536
537         t_vector.clear();
538     }
539
540
541     //TODO when to delete what?
542     for (auto &i : gameVector) {
543         delete i;
544     }
545
546     end_time = my_clock();
547     double time = end_time - start_time;
548     std::cout << numTourney << " games took: " << time << std::endl;
549
550     std::cout << "Got all results - ready to print" << std::endl;
551     std::vector<int> gameResults;
552     for (int i = 0; i < 45; i++) {
553         gameResults.push_back(0);
554     }
555
556     for (int i = 0; i < numTourney; i++) {
557         for (int j = 0; j < 45; j++) {
558             gameResults[j] += resultVector[i][j];
559         }
560     }
561     // Sort players by performance
562     std::vector<std::pair<int,int>> gameResultsPaired;
563     for (int i = 0; i < gameResults.size(); i++) {
564         gameResultsPaired.push_back(std::make_pair(gameResults[i], i));
565     }
566
567     // now gameResultsPaired holds an ascending list of pairs - pair.first = value, pair.second = index
568     std::sort(gameResultsPaired.begin(), gameResultsPaired.end());
569
570     // keep #num of elite players and return them
571     std::vector<Player*> elitePlayers;
```

```
572     elitePlayers.resize(numElitePlayersToKeep);
573     for (int i = 0; i < numElitePlayersToKeep; i++) {
574         elitePlayers.at(i) = players[gameResultsPaired.at(i).second];
575     }
576 
577     float total = 0.0;
578     for (auto &pair : gameResultsPaired) {
579         std::cout << "Player " << pair.second << " average place:  " << pair.first / (float)numTourney << std::endl;
580         total +=  pair.first / (float)numTourney;
581     }
582 
583     std::cout << "Total: " << total << std::endl;
584 
585     // Returning the top players that we want to keep
586     return elitePlayers;
587 }
588 
589 std::vector<long> startOneGameCicle(Game *game, std::vector<std::vector<long>> &resultVector, int id) {
590     game->playGame();
591     game->cleanUpGame();
592 
593     resultVector[id] = game->getOverallGameResults();
594     std::cout << "Thread " << id << "finished game" << std::endl;
595 
596     return game->getOverallGameResults();
597 }
598 
599 
```