RFO ALGOTYTHM ARYICLE

There are many approaches to solving optimization problems, among which genetic algorithms occupy a special place due to their ability to effectively explore the search space by simulating the processes of natural evolution. The conventional genetic algorithm uses binary encoding of solutions, which requires converting real numbers into binary format. This transformation not only introduces additional complexity, but also significantly weighs down the algorithm. In the modern world, minimizing calculation costs plays a decisive role, and the productivity of a method tends to be directly proportional to its speed. While solving this problem, I came up with an idea of how to preserve genetic operators while replacing the most difficult calculations of converting real numbers with simpler and less energy-consuming operations.

My algorithm, Royal Flush Optimization (RFO), is a new approach to solving optimization problems that retains the main advantages of genetic algorithms but uses a more direct way of representing solutions. The key idea is to split each coordinate of the search space into sectors, similar to how a poker hand is made up of individual cards of a certain rank. Instead of working with bit strings, the algorithm manages map ranks (sector numbers), which allows the topology of the search space to be naturally preserved.

In my opinion, the main advantages of the proposed approach are both its simplicity of implementation and intuitive clarity (working with "maps" is more visual than with bit strings), and the absence of the need to encode and decode real numbers while preserving the combinatorial properties of the genetic algorithm. In this article, we will consider in detail the implementation of the algorithm and the features of the decision modification operators.

The poker metaphor not only gives the algorithm its name, but also describes its essence well: just as in poker a player strives to collect the best combination of cards, the algorithm combines sectors of different solutions, gradually forming optimal "hands." Just as in poker each card has its own rank and suit, so in the algorithm each sector has its own value and position in the search space. In this case, as in a real game, it is not only the value of individual cards that is important, but also their interaction in the overall combination.

It is worth noting that this approach can be considered as a generalization of the ideas of

discrete optimization to the case of continuous spaces, which opens up interesting prospects for theoretical analysis and practical application. Just as poker combines elements of chance and strategy, RFO combines random search with directed optimization, making it effective for complex multivariate problems.

Implementation of the algorithm The Royal Flush Optimization (RFO) algorithm is based on the idea of representing the search space as discrete sectors, similar to how cards in poker have certain ranks. In a conventional genetic algorithm, the values (optimized parameters) for all coordinates are converted into binary code and folded into a chromosome-like sequence, which requires additional computational costs. In RFO, we abandon this approach in favor of a simpler and more intuitive presentation.

Instead of binary encoding, we break each coordinate of the search space into sectors, assigning them values similar to poker cards - from jack to ace (J, Q, K, A). The number of ranks (sectors) is specified in the external parameter of the algorithm and can be any integer value. Thus, any point in the search space can be represented as a combination of maps, where each map corresponds to a specific sector according to its coordinate. This approach not only simplifies the calculations, but also preserves the combinatorial properties of the genetic algorithm.

During the optimization, the algorithm works with "hands" - sets of cards representing different solutions. Crossover and mutation operators are applied directly to "hands" (sets of cards), where each hand is analogous to a chromosome, allowing exploration of the search space without the need for binary conversion and back.

The illustration below (Figure 1) clearly demonstrates this principle. It shows a three-dimensional search space with X, Y, and Z coordinates, where each coordinate is divided into sectors corresponding to the ranks of the maps. On the right are examples of "hands" - various combinations of cards that the algorithm forms while searching for the optimal solution.

RFO

Figure 1. RFO algorithm and partitioning coordinates into card deck ranks

Let's move on to writing the pseudocode for the Royal Flush Optimization (RFO) algorithm:

Initialization: Set the number of players at the "poker table" (population size) Determine the size of the "deck" (the number of sectors for each dimension) Set the probability of "bluffing" (mutation) Create an initial "hand" - randomly generate card ranks for each coordinate Transform ranks into real values with a random offset within the sector The main game loop: For each position at the table: Select the "opponent" using quadratic selection (stronger "hands" have a higher chance of being selected) Copy the current "hand" (solution) Perform a three-point exchange of cards: Randomly select three "cutting" points Sort cutting points Randomly select a starting point (0 or 1) Exchange cards between the current hand and the opponent's hand A "bluff" (mutation probability) is possible - a random change in the rank of a card with a given probability Transform the obtained map ranks into real coordinate values Rating and update: Calculate the value of each "hand" (the value of the objective function) Remember the best combination found (global best solution) Combine the current hands with the previous deck Sort all the "hands" by their value The best "hands" pass on to the next generation Termination of the algorithm upon reaching the stopping criterion Move on to writing the algorithm code. Write the S_RFO_Agent structure representing an object containing information about the "hand" in the context of gameplay.

Structure fields:

card [] — array for storing the real value of the card. f — hand value (fitness function value). cardRanks [] — array of integers representing the "card ranks" (sector numbers). Init () method initializes the structure, takes a single "coords" parameter, which specifies the number of cards in the "hand".

//—— // Structure for representing a single "hand" struct S_RFO_Agent { double card []; // cards double f; // value of the fitness function ("hand value") int cardRanks []; // sector numbers ("map ranks")

void Init (int coords)

{

```
  ArrayResize (cardRanks, coords);

  ArrayResize (card,     coords);

  f = -DBL_MAX;      // initialization with minimum value

}

};
```
//——— The C_AO_RFO class implements the algorithm and inherits properties and methods from the C_AO base class. Let's look at it in more detail.

C_AO_RFO () constructor sets values for class variables, initializes parameters: popSize — population size (poker table) is set to 50. deckSize — number of cards in the deck (or sectors) - 1000. dealerBluff — probability of bluffing (mutation) is set at 3% (0.03). 'params' array is used to store parameters, is resized to 3 and filled with values corresponding to popSize, deckSize and dealerBluff. SetParams () method — the method extracts values from the "params" array and assigns them to the corresponding class variables.

Init ()method is designed to initialize the class with the passed parameters, such as the minimum and maximum values of the parameters to be optimized and their step.

The Moving() and Revision() methods are used to perform operations related to shuffling cards in your hand and revising their best combination.

Class fields: deckSize — number of sectors in the dimension. dealerBluff — mutation probability. deck [], tempDeck [], hand [] — arrays of S_RFO_Agent type representing the main deck, the temporary deck for sorting, and the current hand (descendants), respectively. Private class members: cutPoints — number of "cutting" points of the card set at hand that are used to combine card set variants. tempCuts [] and finalCuts [] — arrays for storing temporary and final "cutting" point indices. The methods used are Evolution() - responsible for performing the basic evolution of card permutations, and DealCard() - responsible for converting a sector to its real value. The ShuffleRanks () method is responsible for mutating ranks (randomly choosing among available ranks).
//——— class C_AO_RFO : public C_AO { public: C_AO_RFO () { ao_name = "RFO"; ao_desc = "Royal Flush Optimization"; ao_link =

"https://www.mql5.com/en/articles/17063";

popSize   = 50;   // "poker table" (population) size

deckSize   = 1000;   // number of "cards" in the deck (sectors)

dealerBluff = 0.03;   // "bluff" (mutation) probability

ArrayResize (params, 3);

params [0].name = "popSize";   params [0].val = popSize;

params [1].name = "deckSize";   params [1].val = deckSize;

params [2].name = "dealerBluff"; params [2].val = dealerBluff;

}

void SetParams () { popSize = (int)params [0].val; deckSize = (int)params [1].val; dealerBluff = params [2].val; }

bool Init (const double &rangeMinP [], // minimum values const double &rangeMaxP [], // maximum values const double &rangeStepP [], // step change const int epochsP = 0); // number of epochs

void Moving (); void Revision ();

//-- int deckSize; // number of sectors in the dimension double dealerBluff; // mutation probability

S_RFO_Agent deck []; // main deck (population) S_RFO_Agent tempDeck []; // temporary deck for sorting S_RFO_Agent hand []; // current hand (descendants)

private: //-- int cutPoints; // number of cutting points int tempCuts []; // temporary indices of cutting points int finalCuts []; // final indices taking the beginning and end into account

void Evolution (); // main process of evolution double DealCard (int rank, int suit); // convert sector to real value void ShuffleRanks (int &ranks []); // rank mutation };
//——— The Init method is intended to initialize an object of the C_AO_RFO class.

The method begins by calling a function that performs standard initialization of the given parameters, such as minimum and maximum values, and parameter change steps. If this initialization fails, the method terminates and returns 'false'. After successfully initializing the parameters, the method proceeds to prepare data structures for storing information about "hands" and "decks". This involves resizing the arrays for storing "hands" and "decks" to match the population size.

The method then initializes each element of the "hands" array using a special method that configures them based on the specified coordinates. Similarly, the "deck" and "temp deck" arrays are also prepared and initialized. The method sets the number of cutting points required for the crossover algorithm. In this case, three cutting points are set (this is the best value found after experiments). Then, arrays are set up to store temporary and final cutting points. At the end, the method returns the value "true", which confirms that the initialization was successfully completed.

//——— bool C_AO_RFO::Init (const double &rangeMinP [], const double &rangeMaxP [], const double &rangeStepP [], const int epochsP = 0) { if (!StandardInit (rangeMinP, rangeMaxP, rangeStepP)) return false;

//-- // Initialize structures for storing "hands" and "decks" ArrayResize (hand, popSize); for (int i = 0; i < popSize; i++) hand [i].Init (coords);

ArrayResize (deck, popSize * 2); ArrayResize (tempDeck, popSize * 2); for (int i = 0; i < popSize * 2; i++) { deck [i].Init (coords); tempDeck [i].Init (coords); }

// Initialize arrays for cutting points cutPoints = 3; // three cutting points for a "three-card" crossover ArrayResize (tempCuts, cutPoints); ArrayResize (finalCuts, cutPoints + 2);

return true; }
//——— The Moving method is responsible for the process of "moving" or updating the state of the population within the RFO algorithm.

Checking the status — method begins execution by checking the condition that determines whether the initial initialization "dealing" of cards has been completed. If this is not the case (revision == false), the method performs initialization.

Initializing the initial distribution — the method iterates through all elements of the population, for each element of the population (each "hand") a set of cards is created. The inner loop iterates through each required number of cards and performs the following actions:

A card rank is randomly selected from the deck. The method is then called to deal the card based on the generated rank. The resulting map is adjusted using a function that checks whether it is within a given range and makes the necessary changes depending on the specified parameters. Finally, the value of the received map is set to the "a" array. Update the state — after initialization is complete, "revision" is set to "true", indicating that the initial distribution is complete and no further reinitialization is required.

Calling the Evolution () method — if the initial deal has already been made, the method proceeds to perform the evolutionary process of shuffling and dealing cards into hands.

//——— void C_AO_RFO::Moving () { //-- if (!revision) { // Initialize the initial "distribution" for (int i = 0; i < popSize; i++) { for (int c = 0; c < coords; c++) { hand [i].cardRanks [c] = u.RNDminusOne

(deckSize); hand [i].card [c] = DealCard (hand [i].cardRanks [c], c); hand [i].card [c] = u.SeInDiSp (hand [i].card [c], rangeMin [c], rangeMax [c], rangeStep [c]);

```
    a [i].c [c] = hand [i].card [c];
  }
}
```

```
revision = true;
return;
}
```

//---------- Evolution (); }
//—— The Revision method is responsible for finding the best "combination" of "hands", evaluating their suitability and updating the overall deck.

Finding the best combination:

The method starts by initializing the bestHand variable that will store the index of the best hand among all members of the population. Then a loop is performed that iterates through all elements of the population (from 0 to popSize). Inside the loop, the method compares the fitness value of each "a" hand with the current best value of fB. If the fitness value of the current hand is greater than fB, an update is made with the new best value and the index of that "hand" is assigned to the bestHand variable. If the best hand is found, its cards are copied into the cB array, which allows the state of the best combination (the best global solution) to be saved. The method then updates the fitness values for each hand in the "hand" array to be equal to the corresponding values in the "a" array. This is necessary to ensure that the suitability data for each hand is up to date. After updating the fitness values, the current hands from the "hand" array are added to the general "deck" array, starting at the popSize position (that is, at the end of the population, its second half).

Finally, the method sorts the "deck" array using a separate tempDeck temporary array to arrange the deck by combination value. This allows us to take advantage of the increased probability of selecting valuable card combinations during selection with their subsequent combination.

```
//————————————————————————————————————————————————————————————————————————— void C_AO_RFO::Revision () { // Search for the best "combination" int bestHand = -1;

for (int i = 0; i < popSize; i++) { if (a [i].f > fB) { fB = a [i].f; bestHand = i; } }

if (bestHand != -1) ArrayCopy (cB, a [bestHand].c, 0, 0, WHOLE_ARRAY);

//---------------------------------------------------------------------------- // Update fitness values for (int i = 0; i < popSize; i++) { hand [i].f = a [i].f; }

// Add current hands to the general deck for (int i = 0; i < popSize; i++) { deck [popSize + i] = hand [i]; }

// Sort the deck by combination value u.Sorting (deck, tempDeck, popSize * 2); }
//————————————————————————————————————————————————————————————————————————— The Evolution method is responsible for the main logic of the algorithm, in which cards are exchanged between the "hands" of players at the table, "bluffing" takes place and the real values of the cards are updated.

The method begins with a loop that iterates through all elements of the population. The following actions are performed:

Selecting an opponent:

To select an opponent, a random number is generated, which is then squared to increase the
```

selection (the probability of selecting an opponent is intersected with its rating). This makes it more likely that you will choose the best combination of hands. The random number is scaled using the u.Scale function to obtain the opponent's index. The current hand (from the "deck" array) is copied to the "hand" array. The method generates random "cutting" points for the hand of cards. These points determine which cards will be exchanged between the two hands. The cutting points are sorted and bounds are added to them; the first bound is set to "0" and the last bound is set to "coords - 1". The method selects a random starting point to begin exchanging cards using u.RNDbool (). After the exchange of cards is completed, a chance to "bluff" is given.

Converting ranks to real values: In the final loop, the card ranks are converted to their values using the DealCard method and checked for compliance with the established boundaries. After this, the values in the "a" array, which contains the final cards, are updated.
//——— void C_AO_RFO::Evolution () { // For each position at the table for (int i = 0; i < popSize; i++) { // Select an opponent based on their rating (probability squared to enhance selection) double rnd = u.RNDprobab (); rnd *= rnd; int opponent = (int)u.Scale (rnd, 0.0, 1.0, 0, popSize - 1);

// Copy the current hand

hand [i] = deck [i];

// Define cutting points for card exchange

for (int j = 0; j < cutPoints; j++)

{

  tempCuts [j] = u.RNDminusOne (coords);

}

// Sort cutting points and add borders

ArraySort (tempCuts);

ArrayCopy (finalCuts, tempCuts, 1, 0, WHOLE_ARRAY);

finalCuts [0] = 0;

```
finalCuts [cutPoints + 1] = coords - 1;


// Random selection of a starting point for exchange

int startPoint = u.RNDbool ();


// Exchange cards between hands

for (int j = startPoint; j < cutPoints + 2; j += 2)

{

  if (j < cutPoints + 1)

  {

    for (int len = finalCuts [j]; len < finalCuts [j + 1]; len++) hand [i].cardRanks [len] = deck
[opponent].cardRanks [len];

  }

}


// Possibility of "bluffing" (mutation)

ShuffleRanks (hand [i].cardRanks);


// Convert ranks to real values

for (int c = 0; c < coords; c++)

{

  hand [i].card [c] = DealCard (hand [i].cardRanks [c], c);

  hand [i].card [c] = u.SeInDiSp (hand [i].card [c], rangeMin [c], rangeMax [c], rangeStep [c]);


  a [i].c [c] = hand [i].card [c];

}
```

}}
//——— The DealCard method is a key element of the Royal Flush Optimization algorithm, transforming discrete sectors of the search space into continuous coordinate values. The method takes two parameters as input: "rank" — the rank of the card and "suit" — the coordinate index (suit).

The transformation consists of two stages. First, the size of one sector (suitRange) is calculated by dividing the entire search range by the number of sectors. Then a specific value is generated within the selected sector. The u.RNDprobab() random offset ensures uniform exploration of the space within each sector, and "rank" defines the base position in the search space.

This approach allows combining a discrete representation of solutions through sectors with a continuous search space, providing a balance between global and local search.

//——— double C_AO_RFO::DealCard (int rank, int suit) { // Convert the map rank to a real value with a random offset within the sector double suitRange = (rangeMax [suit] - rangeMin [suit]) / deckSize; return rangeMin [suit] + (u.RNDprobab () + rank) * suitRange; }
//——— The ShuffleRanks method implements the mutation mechanism in the Royal Flush Optimization algorithm, acting as a "bluff" when working with card ranks. Given an array of ranks by reference, the method iterates through each coordinate and, with the dealerBluff probability, replaces the current rank with a random value from the range of valid ranks in the deck. This process can be compared to a situation in poker when a player unexpectedly changes the card in his hand, introducing an element of unpredictability into the game. This mutation mechanism is intended to help the algorithm avoid getting stuck in local optima and maintain a diversity of possible solutions during the optimization.

//——— void C_AO_RFO::ShuffleRanks (int &ranks []) { // Rank shuffle (mutation) for (int i = 0; i < coords; i++) { if (u.RNDprobab () < dealerBluff) ranks [i] = (int)MathRand () % deckSize; } }

//——

Test results RFO algorithm test results:

RFO|Royal Flush Optimization|50.0|1000.0|0.03|

5 Hilly's; Func runs: 10000; result: 0.8336125672709654 25 Hilly's; Func runs: 10000; result: 0.7374210861383783 500 Hilly's; Func runs: 10000; result: 0.34629436610445113

5 Forest's; Func runs: 10000; result: 0.8942431024645086 25 Forest's; Func runs: 10000; result: 0.7382367793268382 500 Forest's; Func runs: 10000; result: 0.24097956383750824

5 Megacity's; Func runs: 10000; result: 0.6315384615384616 25 Megacity's; Func runs: 10000; result: 0.5029230769230771 500 Megacity's; Func runs: 10000; result: 0.16420769230769366

All score: 5.08946 (56.55%)

The final score of 56.55% is a very respectable result. In the visualization, the algorithm does not demonstrate any specific behavior; it looks like a chaotic wandering of individual points.

Hilly

RFO on the Hilly test function

Forest

RFO on the Forest test function

Megacity

RFO on the Megacity test function

Based on the test results, the RFO optimization algorithm ranks 15th, joining the list of the strongest known algorithms.

AO Description Hilly Hilly final Forest Forest final Megacity (discrete) Megacity final Final result % of MAX

10 p (5 F) 50 p (25 F) 1000 p (500 F) 10 p (5 F) 50 p (25 F) 1000 p (500 F) 10 p (5 F) 50 p (25 F) 1000 p (500 F) 1 ANS across neighbourhood search 0.94948 0.84776 0.43857 2.23581 1.00000 0.92334 0.39988 2.32323 0.70923 0.63477 0.23091 1.57491 6.134 68.15 2 CLA code lock algorithm (joo) 0.95345 0.87107 0.37590 2.20042 0.98942 0.91709 0.31642 2.22294 0.79692 0.69385 0.19303 1.68380 6.107 67.86 3 AMOm animal migration ptimization M 0.90358 0.84317 0.46284 2.20959 0.99001 0.92436 0.46598 2.38034 0.56769 0.59132 0.23773 1.39675 5.987 66.52 4 (P+O)ES (P+O) evolution strategies 0.92256 0.88101 0.40021 2.20379 0.97750 0.87490 0.31945 2.17185 0.67385 0.62985 0.18634 1.49003 5.866 65.17 5 CTA comet tail algorithm (joo) 0.95346 0.86319 0.27770 2.09435 0.99794 0.85740 0.33949 2.19484 0.88769 0.56431 0.10512 1.55712 5.846 64.96 6 TETA time evolution travel algorithm (joo) 0.91362 0.82349 0.31990 2.05701 0.97096 0.89532 0.29324 2.15952 0.73462 0.68569 0.16021 1.58052 5.797 64.41 7 SDSm stochastic diffusion search M 0.93066 0.85445 0.39476 2.17988 0.99983 0.89244 0.19619 2.08846 0.72333 0.61100 0.10670 1.44103 5.709 63.44 8 AAm archery algorithm M 0.91744 0.70876 0.42160 2.04780 0.92527 0.75802 0.35328 2.03657 0.67385 0.55200 0.23738 1.46323 5.548 61.64 9 ESG evolution of social groups (joo) 0.99906 0.79654 0.35056 2.14616 1.00000 0.82863 0.13102 1.95965 0.82333 0.55300 0.04725 1.42358 5.529 61.44 10 SIA simulated isotropic annealing (joo) 0.95784 0.84264 0.41465 2.21513 0.98239 0.79586 0.20507 1.98332 0.68667 0.49300 0.09053 1.27020 5.469 60.76 11 ACS artificial cooperative search 0.75547 0.74744 0.30407 1.80698 1.00000 0.88861 0.22413 2.11274 0.69077 0.48185 0.13322 1.30583 5.226 58.06 12 DA dialectical algorithm 0.86183 0.70033 0.33724 1.89940 0.98163 0.72772 0.28718 1.99653 0.70308 0.45292 0.16367 1.31967 5.216 57.95 13 BHAm black hole algorithm M 0.75236 0.76675 0.34583 1.86493 0.93593 0.80152 0.27177 2.00923 0.65077 0.51646 0.15472 1.32195 5.196 57.73 14 ASO anarchy society optimization 0.84872 0.74646 0.31465 1.90983 0.96148 0.79150 0.23803 1.99101 0.57077 0.54062 0.16614 1.27752 5.178 57.54 15 RFO royal flush optimization (joo) 0.83361 0.73742 0.34629 1.91733 0.89424 0.73824 0.24098 1.87346 0.63154 0.50292 0.16421 1.29867 5.089 56.55 16 AOSm atomic orbital search M 0.80232 0.70449 0.31021 1.81702 0.85660 0.69451 0.21996 1.77107 0.74615 0.52862 0.14358 1.41835 5.006 55.63 17 TSEA turtle shell evolution algorithm (joo) 0.96798 0.64480 0.29672 1.90949 0.99449 0.61981 0.22708 1.84139 0.69077 0.42646 0.13598 1.25322 5.004 55.60 18 DE differential evolution 0.95044 0.61674 0.30308 1.87026 0.95317 0.78896 0.16652 1.90865 0.78667 0.36033 0.02953 1.17653 4.955 55.06 19

CRO chemical reaction optimization 0.94629 0.66112 0.29853 1.90593 0.87906 0.58422 0.21146 1.67473 0.75846 0.42646 0.12686 1.31178 4.892 54.36 20 BSA bird swarm algorithm 0.89306 0.64900 0.26250 1.80455 0.92420 0.71121 0.24939 1.88479 0.69385 0.32615 0.10012 1.12012 4.809 53.44 21 HS harmony search 0.86509 0.68782 0.32527 1.87818 0.99999 0.68002 0.09590 1.77592 0.62000 0.42267 0.05458 1.09725 4.751 52.79 22 SSG saplings sowing and growing 0.77839 0.64925 0.39543 1.82308 0.85973 0.62467 0.17429 1.65869 0.64667 0.44133 0.10598 1.19398 4.676 51.95 23 BCOm bacterial chemotaxis optimization M 0.75953 0.62268 0.31483 1.69704 0.89378 0.61339 0.22542 1.73259 0.65385 0.42092 0.14435 1.21912 4.649 51.65 24 ABO african buffalo optimization 0.83337 0.62247 0.29964 1.75548 0.92170 0.58618 0.19723 1.70511 0.61000 0.43154 0.13225 1.17378 4.634 51.49 25 (PO)ES (PO) evolution strategies 0.79025 0.62647 0.42935 1.84606 0.87616 0.60943 0.19591 1.68151 0.59000 0.37933 0.11322 1.08255 4.610 51.22 26 TSm tabu search M 0.87795 0.61431 0.29104 1.78330 0.92885 0.51844 0.19054 1.63783 0.61077 0.38215 0.12157 1.11449 4.536 50.40 27 BSO brain storm optimization 0.93736 0.57616 0.29688 1.81041 0.93131 0.55866 0.23537 1.72534 0.55231 0.29077 0.11914 0.96222 4.498 49.98 28 WOAm wale optimization algorithm M 0.84521 0.56298 0.26263 1.67081 0.93100 0.52278 0.16365 1.61743 0.66308 0.41138 0.11357 1.18803 4.476 49.74 29 AEFA artificial electric field algorithm 0.87700 0.61753 0.25235 1.74688 0.92729 0.72698 0.18064 1.83490 0.66615 0.11631 0.09508 0.87754 4.459 49.55 30 AEO artificial ecosystem-based optimization algorithm 0.91380 0.46713 0.26470 1.64563 0.90223 0.43705 0.21400 1.55327 0.66154 0.30800 0.28563 1.25517 4.454 49.49 31 ACOm ant colony optimization M 0.88190 0.66127 0.30377 1.84693 0.85873 0.58680 0.15051 1.59604 0.59667 0.37333 0.02472 0.99472 4.438 49.31 32 BFO-GA bacterial foraging optimization - ga 0.89150 0.55111 0.31529 1.75790 0.96982 0.39612 0.06305 1.42899 0.72667 0.27500 0.03525 1.03692 4.224 46.93 33 SOA simple optimization algorithm 0.91520 0.46976 0.27089 1.65585 0.89675 0.37401 0.16984 1.44060 0.69538 0.28031 0.10852 1.08422 4.181 46.45 34 ABHA artificial bee hive algorithm 0.84131 0.54227 0.26304 1.64663 0.87858 0.47779 0.17181 1.52818 0.50923 0.33877 0.10397 0.95197 4.127 45.85 35 ACMO atmospheric cloud model optimization 0.90321 0.48546 0.30403 1.69270 0.80268 0.37857 0.19178 1.37303 0.62308 0.24400 0.10795 0.97503 4.041 44.90 36 ADAMm adaptive moment estimation M 0.88635 0.44766 0.26613 1.60014 0.84497 0.38493 0.16889 1.39880 0.66154 0.27046 0.10594 1.03794 4.037 44.85 37 ATAm artificial tribe algorithm M 0.71771 0.55304 0.25235 1.52310 0.82491 0.55904 0.20473 1.58867 0.44000 0.18615 0.09411 0.72026 3.832 42.58 38 ASHA artificial showering algorithm 0.89686 0.40433 0.25617 1.55737 0.80360 0.35526 0.19160 1.35046 0.47692 0.18123 0.09774 0.75589 3.664 40.71 39 ASBO adaptive social behavior optimization 0.76331 0.49253 0.32619 1.58202 0.79546 0.40035 0.26097 1.45677 0.26462 0.17169 0.18200 0.61831 3.657 40.63 40 MEC mind evolutionary computation 0.69533 0.53376 0.32661 1.55569 0.72464 0.33036 0.07198 1.12698 0.52500 0.22000 0.04198 0.78698 3.470 38.55 41 IWO invasive weed optimization 0.72679 0.52256 0.33123 1.58058 0.70756 0.33955 0.07484 1.12196 0.42333 0.23067 0.04617 0.70017 3.403 37.81 42 Micro-AIS micro artificial immune system 0.79547 0.51922 0.30861 1.62330 0.72956 0.36879 0.09398 1.19233 0.37667 0.15867 0.02802 0.56335 3.379 37.54 43 COAm cuckoo optimization algorithm M 0.75820 0.48652 0.31369 1.55841 0.74054 0.28051 0.05599 1.07704 0.50500 0.17467 0.03380 0.71347

3.349 37.21 44 SDOm spiral dynamics optimization M 0.74601 0.44623 0.29687 1.48912 0.70204 0.34678 0.10944 1.15826 0.42833 0.16767 0.03663 0.63263 3.280 36.44 45 NMm Nelder-Mead method M 0.73807 0.50598 0.31342 1.55747 0.63674 0.28302 0.08221 1.00197 0.44667 0.18667 0.04028 0.67362 3.233 35.92 RW random walk 0.48754 0.32159 0.25781 1.06694 0.37554 0.21944 0.15877 0.75375 0.27969 0.14917 0.09847 0.52734 2.348 26.09

Summary While researching and developing new optimization methods, we often face the need to find a balance between efficiency and implementation complexity. Work on the Royal Flush Optimization (RFO) algorithm has yielded interesting results that raise questions about the nature of optimization and ways to improve them.

By observing the algorithm performance as it reaches 57% of its theoretical maximum, we see an interesting phenomenon: sometimes simplification can be more valuable than complication. RFO demonstrates that abandoning complex binary coding in favor of a more straightforward sector-based approach can lead to significant acceleration in the algorithm performance while maintaining a sufficiently high quality of solutions. This is reminiscent of the situation in poker, where sometimes a simpler but faster strategy can be more efficient than a more complex one that requires lengthy calculations.

When thinking about the place of RFO in the family of optimization algorithms, one can draw an analogy with the evolution of vehicles. Just as there is a need for fuel-efficient city cars alongside powerful sports cars, so too in the world of optimization algorithms there is room for methods that focus on different priorities. RFO can be viewed as a "low-cost" variant of the genetic algorithm, offering a reasonable trade-off between performance and resource efficiency.

In conclusion, it is worth noting that the development of RFO opens up interesting prospects for further research. This may be just the first step in the development of a whole family of algorithms based on a sector-based approach to optimization. The simplicity and elegance of the method, combined with its practical effectiveness, can serve as a source of inspiration for the creation of new algorithms that balance performance and computational efficiency.

It is worth noting that the division into sectors occurs virtually, without allocating memory in the form of arrays. This RFO framework is an excellent starting point for further development of improved versions of the poker algorithm.

Tab

Figure 2. Color gradation of algorithms according to the corresponding tests

Chart

Figure 3. Histogram of algorithm testing results (scale from 0 to 100, the higher the better, where 100 is the maximum possible theoretical result, in the archive there is a script for calculating the rating table)

RFO pros and cons:

Pros:

There are few external parameters, only two, not counting the population size. Simple implementation. Fast. Well-balanced, good performance on tasks of various dimensions. Disadvantages:

Average convergence accuracy. The article is accompanied by an archive with the current versions of the algorithm codes. The author of the article is not responsible for the absolute accuracy in the description of canonical algorithms. Changes have been made to many of them to improve search capabilities. The conclusions and judgments presented in the articles are based on the results of the experiments.

github: https://github.com/JQSakaJoo/Population-optimization-algorithms-MQL5 Programs used in the article

Name Type Description

1 #C_AO.mqh Include Parent class of population optimization algorithms 2 #C_AO_enum.mqh Include Enumeration of population optimization algorithms 3 TestFunctions.mqh Include Library of test functions 4 TestStandFunctions.mqh Include Test stand function library 5 Utilities.mqh Include Library of auxiliary functions 6 CalculationTestResults.mqh Include Script for calculating results in the comparison table 7 Testing AOs.mq5 Script The unified test stand for all population optimization algorithms 8 Simple use of population optimization algorithms.mq5 Script A simple example of using population optimization algorithms without visualization 9 Test_AO_RFO.mq5 Script RFO test stend Translated from Russian by MetaQuotes Ltd. Original article: https://www.mql5.com/ru/articles/17063