Classical and Agent-Based Evolutionary Algorithms for Investment Strategies Generation

Rafał Dreżewski, Jan Sepielak, Leszek Siwik

Department of Computer Science
AGH University of Science and Technology, Kraków, Poland
drezew@agh.edu.pl

Abstract. The complexity of generating investment strategies makes it hard (or even impossible), in most cases, to use traditional techniques and to find the exact solution. In this chapter the evolutionary system for generating investment strategies is presented. The algorithms used in the system (evolutionary algorithm, co-evolutionary algorithm, and agent-based co-evolutionary algorithm) are verified and compared on the basis of the results coming from experiments carried out with the use of real-life stock data. The conclusions drawn from the results of experiments are such that co-evolutionary and agent-based co-evolutionary techniques better maintain population diversity and generate more general strategies than evolutionary algorithms.

1 Introduction

Investing on the stock market requires analyzing of the great number of possible strategies (which securities should be chosen, when they should be bought and when sold). The majority of the investment decisions are based on analyzing of present and historical data since it allows for predicting future trends. The problem however is that the anticipation of future trends depends on many assumptions, parameters and conditions. Taking into consideration so many assumptions, combinations of parameters and their values leads to the comparison of the great number of graphs. The evaluation of parameters of many securities is difficult and time consuming for the investor and the analysts. As a result, the investor or the analyst is able to analyze only the small subset of all possible strategies, so the optimal investment strategy is usually not found [15].

The set of the strategies which consists of indicator function is infinite because the complexity of the strategy can be unlimited. Formulas of the given strategy are functions of hundreds (or thousands) of parameters. Complexity of the problem makes it impossible to use direct search methods and instead of it a heuristic approach has to be used. For instance *evolutionary algorithms* can be applied here.

Evolutionary algorithms are optimization and search techniques, which are based on the Darwinian model of evolutionary processes [2]. One of the branches of evolutionary algorithms are *co-evolutionary algorithms* [12]. The most important difference between them is the way in which the fitness of the individual is evaluated. In the case of evolutionary algorithms the fitness of the individual depends only on how "valuable" is the solution of the given problem encoded within its genotype. In the case of

co-evolutionary algorithms the fitness of the individual depends on the values of other individuals' fitness. The value of fitness is usually based on the results of tournaments, in which the given individual and some other individuals from the population are engaged. Co-evolutionary algorithms are generally applicable in the cases in which it is difficult or even impossible to formulate an explicit fitness function. Co-evolutionary interactions between individuals have also other positive effects—for example maintaining population diversity [6]. Population diversity is especially important in the case of multi-modal optimization problems, multi-objective optimization problems, and dynamic environments.

Agent-based (co-)evolutionary algorithms have been proposed as the result of research on decentralized models of evolutionary computations. The basic idea of such an approach is the realization of evolutionary processes within a multi-agent system, which leads to very interesting class of systems: (co-)evolutionary multi-agent systems—(Co)EMAS [5]. Such systems have some features which radically differ them from "classical" evolutionary algorithms. The most important of them are the following: synchronization constraints of the computations are relaxed because the evolutionary processes are decentralized (individuals are agents), there exists the possibility of developing hybrid systems using many different soft-computing techniques within one single, coherent agent architecture, and finally there is the possibility of introducing new evolutionary and social mechanisms, which were hard or even impossible to introduce in the case of classical evolutionary algorithms. (Co)EMAS systems have been already applied to solving multi-modal and multi-objective optimization problems [7]. Another area of applications is the modeling and simulation of social and economical phenomena.

In the case of financial and economical computations (and also financial and economical modeling and simulation) we have to deal with dynamic environments and competing or co-operating economical and social agents. As it was said before, co-evolutionary techniques help maintaining population diversity, and agent-based co-evolutionary systems maintain the diversity even better. What is more, agent-based approach allows us to easily model social and economical agents and relations between them.

In the chapter the component-based system for generating investment strategies is presented. In the system three algorithms were implemented: "classical" evolutionary algorithm, co-evolutionary algorithm, and agent-based co-evolutionary algorithm. These algorithms were assessed and compared during the series of experiments, which results conclude the chapter.

2 Previous Research on Evolutionary Algorithms for Generating Investment Strategies

During recent years there can be observed the growing interests in applying biologically inspired algorithms to solving economic and financial problems [3, 4]. Below, only selected applications of evolutionary algorithms in systems supporting investment-related decision making are presented. To the authors' best knowledge, there were no attempts to apply agent-based (co-)evolutionary algorithms in such systems.

S. K. Kassicieh, T. L. Paez and G. Vora used the genetic algorithm for supporting the investment decisions making [10]. Their algorithm operated on historical stock data. The tasks of the algorithm included selecting company to invest in. The time series of the considered companies were given. In their system some logical operations were carried out on the data. The genetic algorithm was used to determine, which logical operators should be applied in a given situation.

O. V. Pictet, M. M. Dacorogna, R. D. Dave, B. Chopard, R. Schirru and M. Tomassini ([11]) presented the genetic algorithm for the automatic generation of trade models represented by financial indicators. Three algorithms were implemented: genetic algorithm (it converged to local minima and revealed the poor capability of generalization), genetic algorithm with fitness sharing technique developed by X. Yin and N. Germay [16] (it explored the search space more effectively and revealed better abilities to find diverse optima), and genetic algorithm with fitness sharing technique developed by authors themselves in order to prevent the concentration of the individuals around "peaks" of fitness function (it revealed the best capability of generalization). The proposed algorithms selected parameters for indicators and combined them to create new, more complex ones.

F. Allen and R. Karjalainen ([1]) used genetic algorithm for finding trading rules for S&P 500 index. Their algorithm was able to select the structures and parameters for rules. Each rule was composed of a function organized into a tree and a returned value (signal), which indicated whether stocks should be bought or sold at a given price. Components of the rules were the following: functions operating on historical data, numerical or logical constants, logical functions which allowed for combining individual blocks in order to build more complicated rules. Function in the root always returned logical value, which ensured the correctness of the strategy. Fitness measure was based on excess return from the buy-and-hold strategy, however the return did not take into consideration the transaction cost.

3 Evolutionary System for Generating Investment Strategies

In this section both: the architecture of the component-based system for generating investment strategies as well as three algorithms (evolutionary algorithm, co-evolutionary algorithm, and agent-based co-evolutionary algorithm) used as computational components are presented and discussed.

3.1 The Architecture of the System

In Fig. 1 there is presented the high-level architecture of the system. As one may see the system consists of the following basic components:

- *DataSource*—this component supplies the data to the strategy generator. Historical sessions' stock data are used.
- *Functions*—it contains all classes, which are necessary for creating formulas of strategies. It allows to carry out basic operations on formulas i.e.: initialization, exchange of single functions, adding new functions or removing existing ones. Formulas can be tested on data and, in such a case, the results will be returned.

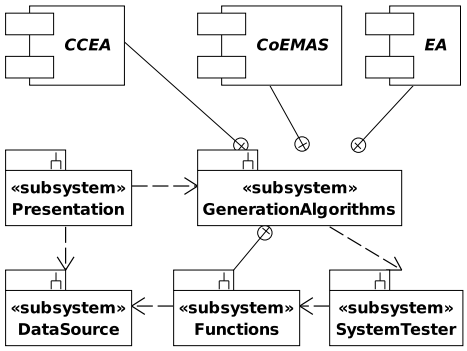


Fig. 1. The architecture of the system used in experiments

– *SystemTester*—this component allows for testing of proposed strategies. It is able to prepare reports concerning the transactions and containing the information about the gained profit. It is used by the generation algorithms to estimate the fitness.
– *GenerationAlgorithms*—it contains implementation of three algorithms responsible for generating strategies: evolutionary algorithm (EA), co-evolutionary algorithm (CCEA) and co-evolutionary multi-agent system (CoEMAS). This component includes the mutation and recombination operators as well as the fitness estimation mechanisms. Implementation of this subsystem was divided into packages shown in Fig. 2:
  - *algorithm* package—it is the most general package containing classes, which are the basis for implementation of other algorithms responsible for investment strategy generation.
  - *EA* package—it contains implementation of evolutionary algorithm.
  - *CCEA* package—it contains implementation of co-evolutionary algorithm.
  - *CoEMAS* package—it contains implementation of agent-based co-evolutionary algorithm.
– *Presentation*—it contains definitions of GUI forms responsible for instance for results presentation, algorithm monitoring etc.

3.2 Data Representation

In all three algorithms implemented and discussed in the course of this chapter the strategy is a pair of formulas. First formula indicates when one should enter the market and the second indicates when one should exit the market. Each formula, which is a part of strategy can be represented as a tree, which leaves and nodes are functions performing some operations. Such tree has a root and any quantity of child nodes. The function placed in the root always returns logical value. Fig. 3 shows the tree of the formula, which can be symbolically written in the following way: $STE(WillR(20), 30) > 10.0$.

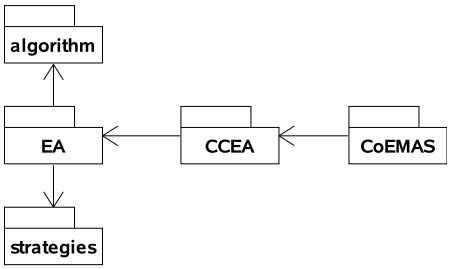


Fig. 2. Package diagram for generation strategies algorithms

When treating this expression as entry formula, the system will generate “buy” signal when the value of Standard Error indicator (STE) from 30 days and for data from Williams’ %R indicator for percentage period equal to 20, will be greater than 10.0.

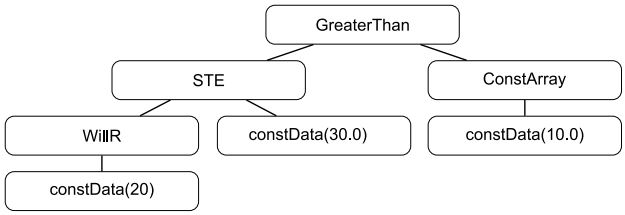


Fig. 3. Tree of exemplary formula

A formula tree is represented in memory as a tree. The root node is an object containing the references to the functions and the references to parameters. These parameters are also the same objects as the root. Leaves of the tree are objects which do not contain parameters. When formula is executed recursive calls occur. In the beginning, the root requires values of all parameters needed to invoke its function. Then the control flows to objects of the parameters. Objects representing parameters behave in the same way as the root object. Leaves do not contain parameters, so they can return the value required by the parent node.

Functions (which formulas are composed of) were divided into four categories:

- Functions returning data arrays (see Table 1). There are 6 such functions.
- Mathematical functions (for example *Abs*—each value in the returned data array is absolute value of corresponding element from the data array passed as an argument, *Cos*—calculates cosine for values from the data array passed as an argument, etc.) There are 40 such functions.
- Tool functions (for example *ProjBandBot*—returns the data array with values of the bottom Projection Band, *STEBandBot*—returns the data array with values of the bottom Standard Error Band, etc.) There are 33 of them.

– Indicator functions (see Table 2). There are 14 of them.

There are 93 functions altogether.

Table 1. Data array functions

Function name	Description
Close	Returns close prices.
ConstArray	Returns array of constant passed as argument.
High	Returns high prices.
Low	Returns low prices.
Open	Returns open prices.
Volume	Returns volumes.

Table 2. Indicator functions

Function name	Description
AD	Returns the data array with values of the Accumulation/Distribution indicator.
ADX	Returns the data array with values of Average Directional Movement indicator.
BBandBot	Returns the data array with values of the Bollinger Band Bottom indicator.
BBandTop	Returns the data array with values of the Bollinger Band Top indicator.
LinearReg	Returns the data array with values of the Linear Regression indicator.
Mov	Returns the data array with values of diverse moving averages.
ROC	Returns the data array with values of the Rate of Change indicator.
RSI	Returns the data array with values of the Relative Strength Index indicator.
STE	Returns the data array with values of the Standard Error indicator.
Stdev	Returns the data array with values of the Standard Deviation indicator.
TSF	Returns the data array with values of the Time Series Forecast indicator.
Var	Returns the data array with values of the Variance indicator.
WillR	Returns the data array with values of the Williams' %R indicator.
Zig	Returns the data array with values of the Zig Zag indicator.

Implemented functions accept the following types of parameters: constants (*integer*, *float* or *enum*), array of constant float values, and values returned by other function (array of logical values or array of float values). Logical constant does not exist because it was not needed for building formulas.

Classes containing implementations of mentioned functions are presented in Fig. 4. A *FunctionBase* class contains meta-data concerning all functions defined in descendent classes. To those meta-data belong for instance feasible values of functions arguments. *SecurityData* class contains session data of single stock. It supplies data to in-

dicator functions. It also accumulates errors from formulas runs (e.g. division by zero, attempt of computing logarithm of negative value, etc.) and allows reporting them when it is required. Other classes contain implementation of indicator functions belonging to the categories mentioned earlier.

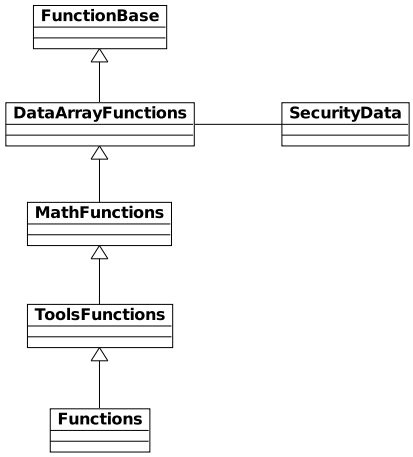


Fig. 4. Hierarchy of classes with functions used to build formulas

3.3 Evolutionary Algorithm (EA)

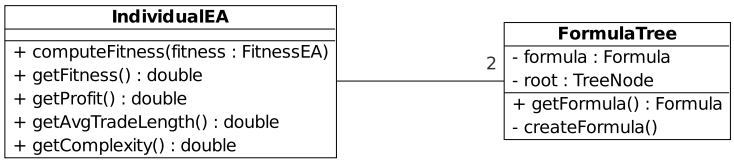


Fig. 5. Individual in EA and its genotype

In the evolutionary algorithm genetic programming approach was used. The genotype of individuals simultaneously contains formula for entering and exiting the market. Class diagram presented in Fig. 5 shows that each individual has (in its genotype) two formula trees which represent formulas for entering and exiting the market. Method *computeFitness* is used to estimate fitness of an individual. When fitness is estimated

its value can be obtained by *getFitness* method. It is also possible to get all components of fitness such as profit, average length of trades, and average complexity of formulas. An object of *FormulaTree* class contains root of tree and cached object of the formula, which is created from the tree using *createFormula* method.

Estimation of the fitness value of the strategies is carried out on the basis of loaded historical data. Two tables of logical values are created as a result of execution of the formulas. The first one relates to the purchase action (entering the market) and the second one relates to the sale action (exiting the market). Algorithm responsible for computing the profit processes tables and determines when a purchase and a sale occur. Entering the market occurs when a system is outside the market and there is the value of "true" in the enter table. Exiting the market occurs when the system is on the market and there is value of "false" in the exit table. In the other cases no operation is performed. When applying this algorithm, the system cannot enter the market once more (before any sale action takes place). In other words, after the sale—purchase action must take place, and after buying—the sale action must occur. The profit/loss from the given transaction is estimated when exiting the market and it is accumulated. The cost of each transaction is included—the commission is calculated by subtracting a certain constant from the transaction value.

Apart from the profit/loss there are also other criteria, which are included in the fitness estimation. The first one is the formula complexity—formulas which are too complex can slow down the computations and increase the memory usage. The complexity of the formulas is determined by summing up of all component functions. The second criteria is the length of the transaction—it depends on the preference of the investor.

Recombination Three kinds of recombination operators are used: *returned value* recombination, *arguments* recombination, and *function* recombination. *Returned value* recombination is performed when there are two functions with different arguments but the same returned values within the formula tree. These functions are exchanged between individuals (functions are moved with their arguments).

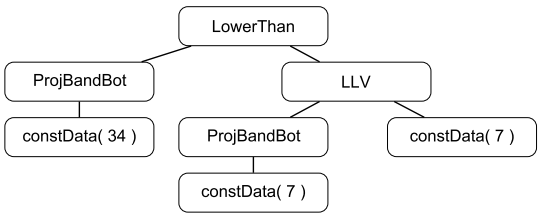


Fig. 6. Parent 1 for *returned value* recombination

Let us consider two formulas:

- $ProjBandBot(34) < LLV(ProjBandBot(7), 7)$ (see Fig. 6), and
- $ProjBandTop(7) > STEBandBot(High(), 7, 7)$ (see Fig. 7),

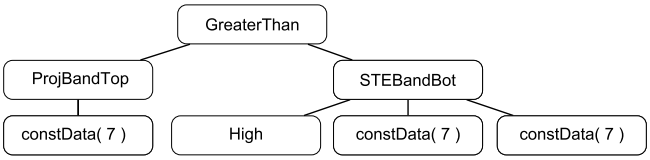


Fig. 7. Parent 2 for *returned value* recombination

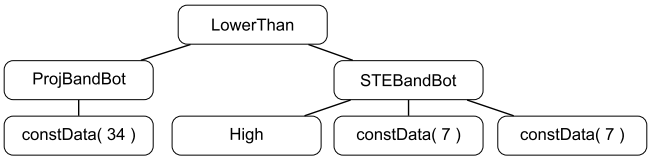


Fig. 8. Descendant 1 after *returned value* recombination

where $ProjBandBot$ is function which calculates the bottom Projection Band indicator, $ProjBandTop$ calculates the top Projection Band indicator, LLV (Lowest Low Value) is indicator, which calculates the lowest value in the data array over the preceding period, and $STEBandBot$ calculates the bottom Standard Error Band indicator. Functions $ProjBandBot$ and $STEBandBot$ have different arguments, but the types of their return values are the same (they both return double values). As a result two new formulas will be created:

- $ProjBandBot(34) < STEBandBot(High(), 7, 7)$ (see Fig. 8), and
- $ProjBandTop(7) > LLV(ProjBandBot(7), 7)$ (see Fig. 9).

Arguments recombination occurs when there are two functions with the same arguments within the parents. These arguments are exchanged between individuals during the recombination.

Let us consider two parents:

- $BBandBot(Close(), 10, Triangular, 4.3)$ (see Fig. 10), and
- $BBandTop(Open(), 3, Simple, 2.9)$ (see Fig. 11),

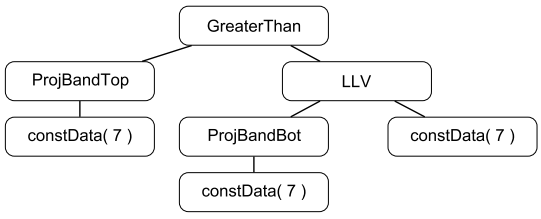


Fig. 9. Descendant 2 after *returned value* recombination

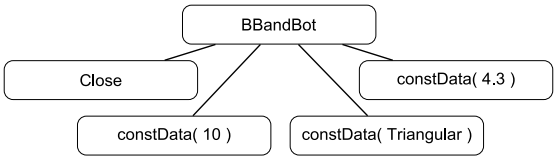


Fig. 10. Parent 1 for *arguments* recombination

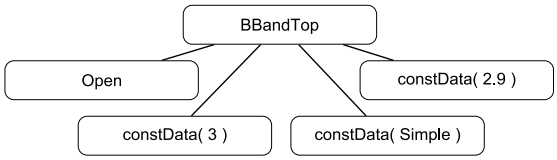


Fig. 11. Parent 2 for *arguments* recombination

where $BBandBot$ and $BBandTop$ calculate respectively bottom and top Bollinger Band indicator. Functions $BBandBot$ and $BBandTop$ have the same arguments. After exchanging these arguments two new descendants will be created:

- $BBandBot(Open(), 3, Simple, 2.9)$ (see Fig. 12), and
- $BBandTop(Close(), 10, Triangular, 4.3)$ (see Fig. 13).

Function recombination can take place when two functions have the same arguments and the same returned values. Let us consider two parents from the previous example. Functions $BBandBot$, $BBandTop$, $Close$ and $Open$ fulfill the assumption which requires return values and parameters to be the same. As a result of applying *function* recombination two new individuals will be created:

- $BBandTop(Open(), 10, Triangular, 4.3)$ (see Fig. 14), and
- $BBandBot(Close(), 3, Simple, 2.9)$ (see Fig. 15).

Only one kind of recombination can be used to create an offspring. Probability of recombination determines which kind of recombination is chosen. Each kind of recombination has also a probability which controls carrying out particular recombination. It determines how often particular elements of formulas are exchanged between parents.

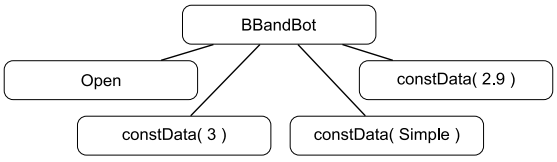


Fig. 12. Descendant 1 after *arguments* recombination

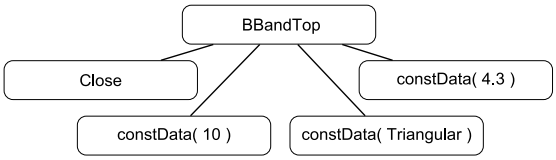


Fig. 13. Descendant 2 after *arguments* recombination

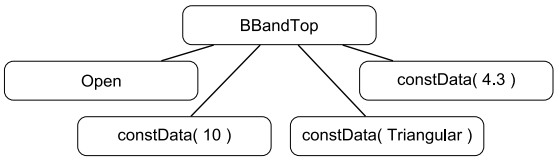


Fig. 14. Descendant 1 after *function* recombination

Parameter *reproduction factor* determines how many offsprings are generated. The size of population is multiplied by this coefficient. The outcome defines the number of offsprings to be created.

During the recombination stage, in the first place, type of recombination is chosen using *RecombinationArray* class. Next, parents are chosen using a tournament selection. When two parents are selected, a certain type of recombination is performed in accordance with, previously determined, type of recombination. Sequence diagram for single recombination is presented in Fig. 16.

Mutation Two types of mutation were used: *function arguments* mutation and *function* mutation. In *function argument* mutation the argument of the function must be a constant value. This constant value is exchanged with the other one coming from the allowed range. For instance, having a function $Add(1,2)$ operator can modify it to $Add(5,2)$.

There are three variants of the *function* mutation:

1. If a given function should be mutated, a list of the functions taking the same arguments is found. If such functions exist, the exchange is performed. If there are no

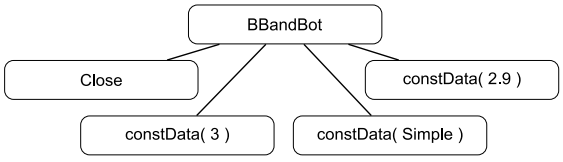


Fig. 15. Descendant 2 after *function* recombination

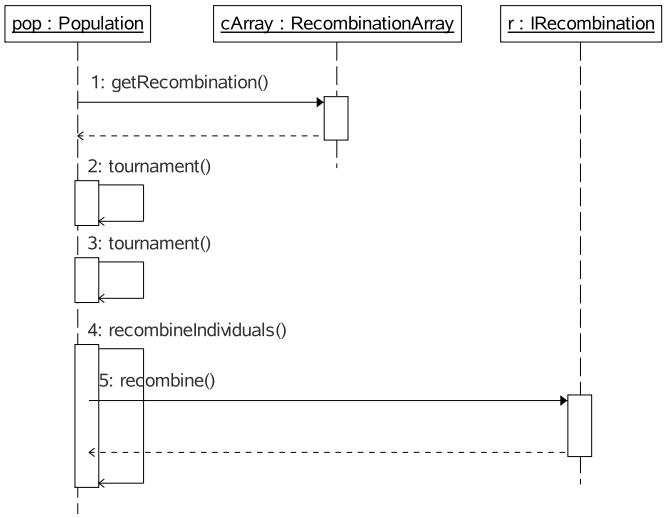


Fig. 16. Carrying out a single recombination

such functions, mutation is not carried out. For instance $And(a, b)$ can be changed to $Or(a, b)$. It was named as function mutation.

2. A given function can be exchanged with the other, completely new one, regardless of arguments—only returned types must match. Arguments of such function are created randomly. For instance:
   - $BBandTop(Month(), 13, E, 4.714)$—before mutation,
   - $BBandTop(HHV(Close(), 3), 13, E, 4.714)$—after mutation.

   Month function returns day of the month, from which data came from, and *HHV* function calculates the highest value in the data array over the preceding period. Function *Month* was exchanged for *HHV*, and its arguments were created randomly. It was named as the entire function mutation.
3. Similarly as in (2), but, if it is possible, parameters of the replaced function are copied to a new one. For instance, there is $Mod(Low(), AD())$—before mutation, and $STE(Low(), 14)$—after mutation. *Mod* function (which calculates modulus) was exchanged for STE function and parameter *Low* was moved. Second parameter had to be created randomly, because STE function requires integer constant as an argument. It was named as the partial function mutation.

Probability of mutation depends on the depth in the formula tree. If it is closer to leaves the mutation probability is greater. Thanks to this, there is a greater chance that the mutation will cause small changes in the genotype.

Many kinds of mutations could be performed on the same part of formula. Mutation factors are used while calculating mutation probability. These coefficients let the user specify how mutation likelihood changes when depth in formula of the mutated element changes. If such coefficient is greater than 1, then the greater depth causes the greater

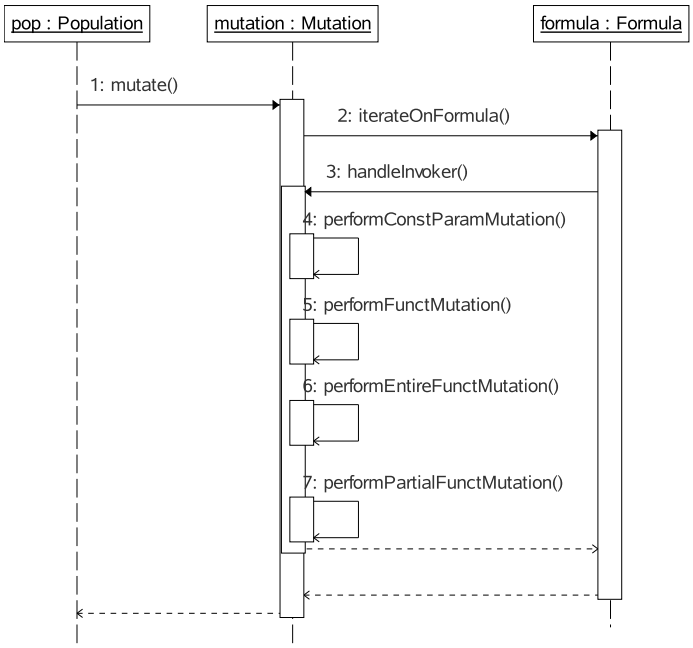


Fig. 17. Carrying out mutation of single formula element

mutation probability. But if coefficient is in the $[0, 1]$ range then the greater depth causes the smaller mutation probability.

Fig. 17 shows mutation sequence of single formula element. *Mutate* method from *Mutation* class is carried out on genotype of an individual. Next, iteration on functions of genotype is performed. *HandleInvoker* method is invoked on each function of genotype. Particular mutations are performed with probability specified by the user.

Selection The tournament selection ([2]) was used in the EA algorithm. In the tournament selection a group of N individuals ($N \geq 2$) is selected. The individual which has the highest fitness is chosen from the group.

After creating the offspring, it was added to the population of parents (the reinsertion mechanism). From such enlarged population the new base population was chosen also with the use of tournament mechanism.

Algorithm 1 shows the scheme of evolutionary algorithm. At the beginning the population is created randomly and evaluated. Next, reproduction, recombination and mutation are performed in each generation. When the individuals are created their fitness value is estimated. Afterwards, new population is selected applying tournament reinsertion on parent and children individuals.

Algorithm 1. Scheme of evolutionary algorithm (EA)

```
1  gen ← 0;
2  random initialization of population Pop(gen);
3  evaluate Pop(gen);
4  while not stop condition do
5  |  Pop^1(gen) ← reproduction Pop(gen);
6  |  Pop^2(gen) ← recombination Pop^1(gen);
7  |  Pop^3(gen) ← mutation Pop^2(gen);
8  |  evaluate Pop^3(gen);
9  |  Pop(gen+1) ← reinsertion (Pop^3(gen) ∪ Pop(gen));
10 |  gen ← gen+1;
11 end
```

3.4 Co-Evolutionary Algorithm (CCEA)

Co-evolution is a process of mutual adaptation of a set of individuals which interact with each other [8]. When an individual becomes better adapted, other individuals also have the opportunity to improve their fitness.

All of evolutionary and co-evolutionary algorithms consist in searching of the population of solutions in accordance with the concept of natural selection. Co-evolutionary algorithms however, differ from ordinary evolutionary algorithms. Individual may interact with other individuals. Partners of interactions may be members of the same population, or may belong to different populations, depending on the nature of the problem being solved.

Many real problems are difficult to solve using standard evolutionary techniques. However, such problems can be decomposed into many sub-problems, and finding their solutions will result in solving an initial problem. *Co-operative co-evolution* is designed to solve the problem X through co-evolving the sets of solutions of sub-problems, for which X was decomposed into [8].

While developing the co-evolutionary algorithm for generating investment strategies, co-operative approach proposed by M. A. Potter and K. A. De Jong ([14]) was used. There are two species in the implemented algorithm: individuals representing entering the market strategies and individuals representing exiting the market strategies. Interactions between these species rely on co-operation.

Class diagram presented in Fig. 18 shows that each individual has in its genotype one formula tree representing the formula for entering or exiting the market depending on population to which the individual belongs. To estimate the fitness, formula is obtained using *getFormula* method and passed to an object of *FitnessCCEA* class. In *setFitness* method—fitness value and its elements are taken from the *FitnessCCEA* object. Likewise in EA fitness and its component can be obtained by proper methods.

During the fitness estimation process individuals are selected into pairs, which form the complete solution. In the first generation, for each evaluated individual from the first species a partner for co-operation from the second species is chosen randomly. For the complete solution created in this way, the fitness is computed and assigned to the individual that is being evaluated.

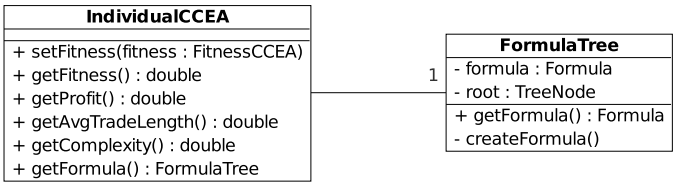


Fig. 18. Individual in CCEA and its genotype

In the next generations, the best individual from the opposite species and from the previous generation is chosen for the evaluated individual. For instance, if second generation has finished then at the beginning of third generation reproduction starts. After recombination and mutation new individuals are created and then they can be evaluated. To do this the best individual from the second generation of exit the market population is retrieved. Then pairs of previously selected individual and each individual from third generation from enter the market population are created. Fitnesses from such formed pairs are assigned to individuals which came from enter the market population. In the similar manner fitnesses of individuals from the second population are established. The best individual from second generation of enter the market population is selected. Pairs composed of this individual and each individual from exit the market population (from third generation) are created. Estimated fitnesses are assigned to individuals from exit the market population.

Algorithm 2 shows the scheme of co-evolutionary algorithm. At the beginning populations of two species are created randomly and fitness of each individual is estimated. Next, reproduction, recombination and mutation are applied on both species in each iteration. When offspring individuals are created, fitness estimation occurs. Both species interact at the stage of fitness computing . After that, reinsertion is performed and single step of CCEA is finished. The pairs of individuals gaining the best profit are the result of the algorithm run.

3.5 Co-Evolutionary Multi-Agent System (CoEMAS)

Co-evolutionary multi-agent system used in the experiments is the agent-based realization of the co-evolutionary algorithm. Its general principles of functioning are in accordance with the general model of co-evolution in multi-agent system [5]. The CoEMAS system is composed of the environment (which include computational nodes—islands—connected with paths) and agents, which can migrate within the environment. The selection mechanism is based on the resources, which are defined in the system. The general rule is such, that in the each time step the environment gives more resources to “better” agents and less resources to “worse” agents. The agents use the resources to perform each activity, like migration, reproduction, and so on. Each time step (the agents can live more than one generation), individuals lose some constant amount of the possessed resources, which is given back to the environment. The agents make all

Algorithm 2. Scheme of co-evolutionary algorithm (CCEA) [13]

```
1  gen ← 0;
2  foreach species s do
3  |  random initialization of population Pop_s(gen);
4  |  evaluate Pop_s(gen);
5  end
6  while not stop condition do
7  |  foreach species s do
8  |  |  Pop_s^1(gen) ← reproduction Pop_s(gen);
9  |  |  Pop_s^2(gen) ← recombination Pop_s^1(gen);
10 |  |  Pop_s^3(gen) ← mutation Pop_s^2(gen);
11 |  |  evaluate Pop_s^3(gen);
12 |  |  Pop_s(gen+1) ← reinsertion (Pop_s^3(gen) ∪ Pop_s(gen));
13 |  end
14 |  gen ← gen+1;
15 end
```

their decisions independently—especially those concerning reproduction and migration. They can also communicate with each other and observe the environment.

In the CoEMAS algorithm realized in the system for generating investment strategies each co-evolutionary algorithm (CCEA) is an agent which is located on one of the islands and independently carries out the computations. The population of each co-evolutionary algorithm also consists of the agents (there are two species of agents within each population, like in the case of CCEA).

Genetic operators and fitness estimation mechanism are the same as in the case of previously described algorithms. Selection mechanism is different—it works on the basis of resources (the agent possessing the greater amount of resource wins the tournament).

On the basis of the amount of the possessed resource each individual decides whether it is ready for the reproduction. It occurs when the level of resource is greater or equal to $r_{min}^{rep,\gamma}$ (see Algorithm 3). Parents are chosen using a tournament. At the mutation stage the amount of resource does not change. During the recombination parents give the offspring certain amount of their resources.

The tournament during the reinsertion phase is also based on the resources—the agent that possesses more resource wins the tournament. At the reinsertion stage better individuals receive more resource from the environment and the worse ones receive less. If the agent possesses less resource than r_{die}^{γ}, it dies.

The possibility of migration of agent-individuals from one population to another was added as well. During the migration the resource possessed by the given agent is reduced by a constant amount.

The *reproduction factor* parameter in CoEMAS determines only the maximum number of offsprings to be created. Practically, the amount of created individuals is smaller and depends on how many individuals have enough resource to reproduce. If it turns out that there are no such individuals any more, reproduction will be stopped.

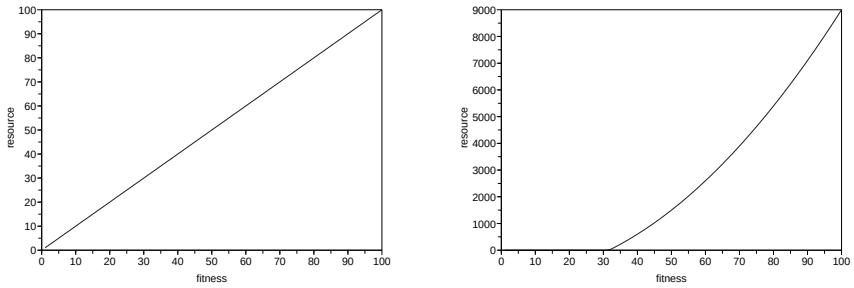
Algorithm 3. Basic activities of agent a in *CoEMAS*

input: index i of profile to perform

switch *profile* $pr_i \in PR^a$ **do** ; /* PR^a is the set of profiles of agent a */
    case *initialization profile*
        $r^\gamma \leftarrow r^\gamma_{init}$; /* r^γ is the amount of resource γ possessed by the agent a, r^γ_{init} is the initial amount of resource given to the agent */
    case *resource profile*
        get from the environment some resource correspondingly to fitness;
        if $r^\gamma \leq r^\gamma_{die} \vee t - t_{create} \geq t_{maxage}$ **then** ; /* r^γ_{die} is the minimal amount of resource needed by the agent to live, t is number of generation, t_{create} is number of generation in which agent was created, and t_{maxage} is maximum number of generations through which agent can live */
            execute $\langle die \rangle$ strategy;
        end
    case *reproduction profile*
        if $r^\gamma > r^{rep,\gamma}_{min}$ **then** ; /* $r^{rep,\gamma}_{min}$ is the minimal amount of resource needed for reproduction */
            execute $\langle seek \rangle$ strategy; /* find second agent to reproduce with */
            execute $\langle recombination \rangle$ strategy;
            execute $\langle mutation \rangle$ strategy;
        end
    case *migration profile*
        if $r^\gamma > r^{mig,\gamma}_{min}$ **then** ; /* $r^{mig,\gamma}_{min}$ is the minimal amount of resource needed for migration */
            execute $\langle migration \rangle$ strategy;
            give $r^{mig,\gamma}_{min}$ amount of resource to the environment;
        end
    end
end

Individuals die when they possess too few resources. However, this approach is insufficient because after a few generations there will be many individuals in population which vegetate (do nothing and do not give back resource to environment) and therefore their existence is useless. For that reason, when individuals pass to new generation they give back a certain percent of resource to environment.

The number of individuals in population depends on the amount of resource in environment, however, the initial size of population is specified by the user. If the environment has more resource, then more individuals can exist. If there is a small amount of resource and many individuals receive it, then many individuals will have too few resources and will be killed at the reinsertion stage.



(a) (b)

Fig. 19. Allocation of resource at initialization stage (a), and in consecutive generations (b)

Each population lives in a certain environment (island). The environment possesses a specified amount of resource which circulates between the environment and agents. The resource of population can change only through migration (individuals which migrate take resource to the other population). The amount of resource in the system is constant because the sum of resource in all populations and in the environment does not change. In the first stage resource is allocated proportionally to the fitness: the greater fitness the greater amount of resource. The entire population receives resource unless there exist individuals whose fitness is not positive. Fig. 19a shows the manner of resource allocation at the initialization stage.

In the consecutive generations—in order to increase selection pressure—only the part of population receives resource. Resource is not allocated proportionally to the fitness value. The fitness value is raised to the power specified by the user and the resource is allocated on the basis of such modified fitness. The manner of resource allocation after initialization is shown in Fig. 19b.

4 The Experiments

In order to examine the generalization capabilities of the system and compare the proposed algorithms, strategies which earn the largest profit per year for random stocks were sought. An attempt was made to determine, which algorithm generalizes in a best way and what quantity of stocks should be used for strategies generation so that the system would not overfit. It is also interesting to assess which algorithm generates the best strategies and has the smallest convergence.

4.1 Plan of the Experiments

The presented results of the experiments comparing the quality of the generated strategies and convergence properties of the considered algorithms are average values from 30 runs of the algorithms. Each algorithm was run for 500 generations on the data of 10 randomly chosen stocks. The session stock data came from the WIG index ([9]) and the period of 5 years was chosen (from 2001-09-29 to 2006-09-29). The size of the population in all the algorithms was equal to about 40 individuals (CoEMAS approach uses variable-size populations).

All experiments were made with the use of optimal values of the parameters. These values were found during consecutive experiments. The algorithms were run 10 times for each parameter value coming from the established range and average results were computed—on this basis the set of best parameters' values were chosen.

While comparing all algorithms three approaches were used:

- on-line efficiency: $\frac{1}{T}\sum_{t=1}^{T} f(t)$, where T is the number of generations algorithm worked through, and $f(t)$ is fitness of the best individual in generation t.
- off-line efficiency: $\frac{1}{T}\sum_{t=1}^{T} f^*(t)$, where $f^*(t) = \max\{f(1), f(2), \ldots, f(t)\}$
- the best value in last generation: $\max_{i=1,\ldots,N} f_i(T)$, N is the number of individuals in the last generation, and f_i is fitness of i-th individual.

While examining the generalization capabilities, each algorithm generated the solution for n random stocks (stage 1). Next, different n stocks were chosen ten times at random and the profit was calculated using the best strategy obtained in the stage 1. Then, the average of these profits was counted. These calculations were carried out four times for $n = 3, 5, 7, 10$.

Like in the first type of experiments (when the quality of the solutions and the convergence properties were compared), populations had similar sizes in the case of all three algorithms. All experiments were carried out on the machine with one AMD Sempron 2600+ processor.

4.2 Parameters' Values Selection

Table 3 and Table 4 show optimal parameters' values, which were determined using back to back experiments. Table 3 concerns all three algorithms. All algorithms have the same values of these parameters. Table 4 concerns only CoEMAS algorithm. Parameters in this table refer to migration and resources which occur only in CoEMAS.

Table 3. Optimal values of parameters for all three algorithms

Parameter	Parameter value
Fitness function	
Price for entering the market	Open
Price for exiting the market	Close
Entry commissions	0.0
Exit commissions	1.0
Kind of got commissions	Points ($)
Transaction length weight	0.1
Profit weight	1.0
Formula complexity weight	0.2
Initialization	
Initial formula depth	4
Mutation	
Entire function mutation probability	0.3
Entire function mutation—if probability depends on depth	true
Entire function mutation factor	2.0
Partial function mutation probability	0.1
Partial function mutation probability—if probability depends on depth	true
Partial function mutation factor	1.5
Function mutation probability	0.03
Function arguments mutation probability	0.03
Formula depth change probability	0.01
Recombination	
Arguments recombination—usage probability	0.4
Arguments recombination—argument change probability	0.3
Function recombination—usage probability	0.4
Function recombination—function change probability	0.3
Return value recombination—usage probability	0.7
Return value recombination—function change probability	0.2
Population	
Population size	40
Reproduction	
Tournament size during reproduction	5
Reproduction factor	0.8
Reinsertion	
Tournament size during reinsertion	3

Table 4. Optimal values of additional parameters for CoEMAS

Parameter	Parameter value
Migration	
Number of generation from which mutation is started	5
Mutation probability	0.05
Resources	
Amount of resource required to reproduction	20.0
Amount of resource given back at migration	14.0
Minimal amount of agent's resource to survive	10.0
Initial amount of environment resource	980.0
Percent of resource received from parents	20.0
Percent of resource which environment loses at reinsertion	100.0
Percent of resource taken back at ageing	3.0
Percent of population which receives energy	20.0
Exponent of *pow* function at reinsertion	4.0

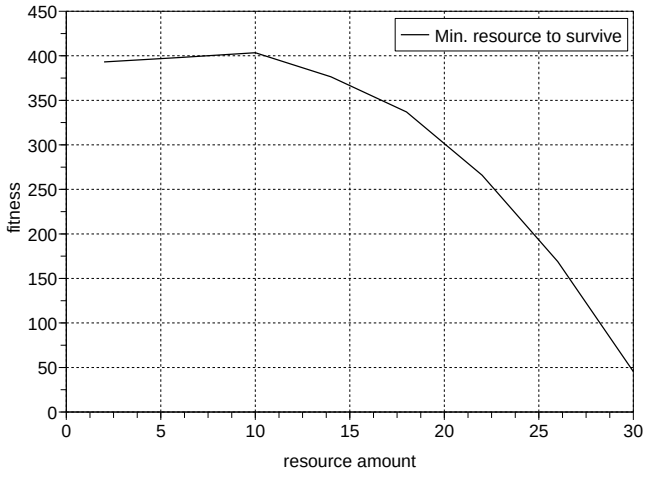


Fig. 20. The influence of minimal amount of resource for individuals to survive on results obtained by CoEMAS (each point is the average of 10 values)

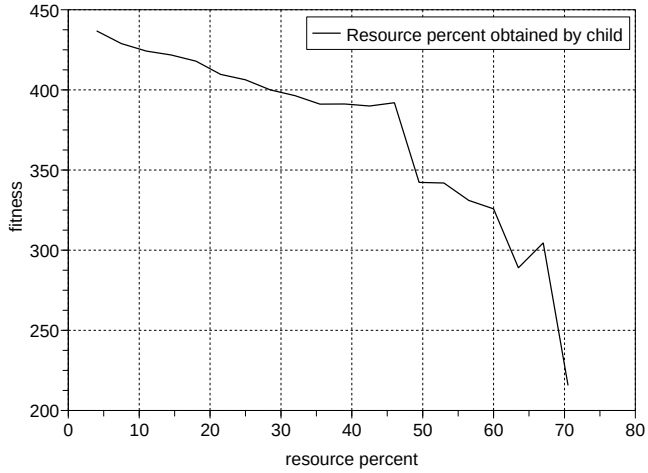


Fig. 21. The influence of the percent of resource obtained by a child after recombination on results obtained by CoEMAS (each point is the average of 10 values)

In some cases, when changing values of parameters, no regularity of influence on efficiency changes was found. But there were some exceptions.

In Fig. 20 there is presented the influence of agent's minimal resource level on CoEMAS efficiency. In the beginning, with the growth of this parameter the value of efficiency slightly increases. But when resource amount exceeds 10 units it systematically decreases.

Fig. 21 presents the influence of the percent of resource obtained by a child after recombination in CoEMAS on its efficiency. It indicates that giving a lot of resource causes a decrease in efficiency of the algorithm. Whereas giving a small amount of resource causes the increase of system's efficiency.

Fig. 22 presents dependency between the amount of resource which individuals give back to environment at migration stage and CoEMAS performance. At the beginning when the amount of resource which is given back grows, efficiency grows too (best individual achieved fitness about 430 units). Optimum is at about 14 units. Then, the increase in returned resource causes performance decrease.

Fig. 23 presents influence of migration probability on CoEMAS efficiency. Only probability smaller than 0.15 causes high performance (the best individual achieved fitness of about 400 units). Efficiency decreases when probability is greater than 0.15.

4.3 Comparison of Algorithms

In EA and CCEA the size of populations during evolution is constant and is adjusted by the parameter specified by the user. Admittedly, the population size increases at recombination step, but after reinsertion becomes again the same as before the recombination.

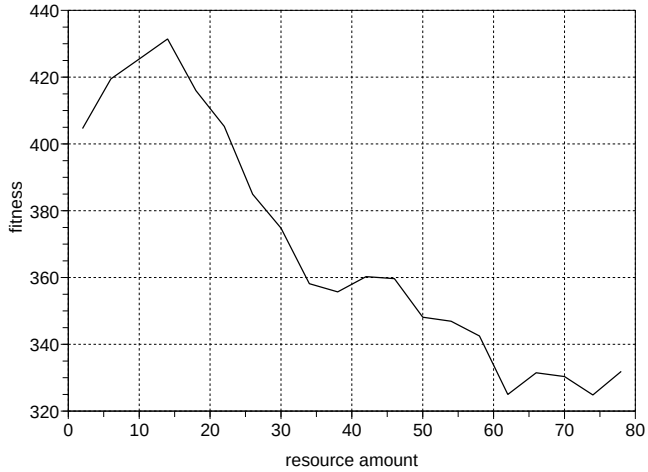


Fig. 22. The influence of the amount of resource given back at migration stage on results obtained by CoEMAS (each point is the average of 10 values)

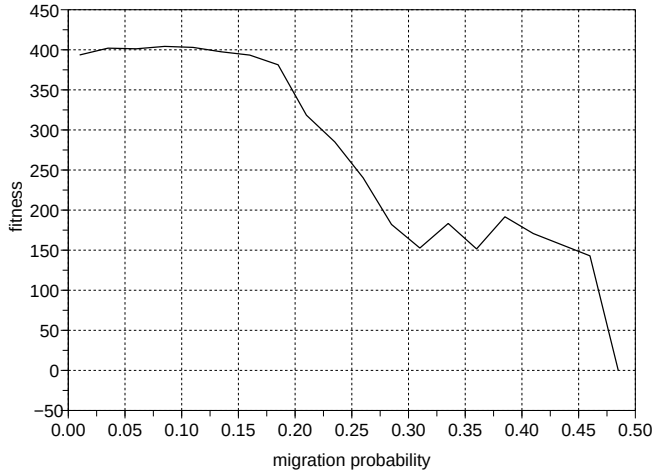


Fig. 23. The influence of the migration probability on results obtained by CoEMAS (each point is the average of 10 values)

Different behavior can be observed in CoEMAS—the size of the population vary during evolution.

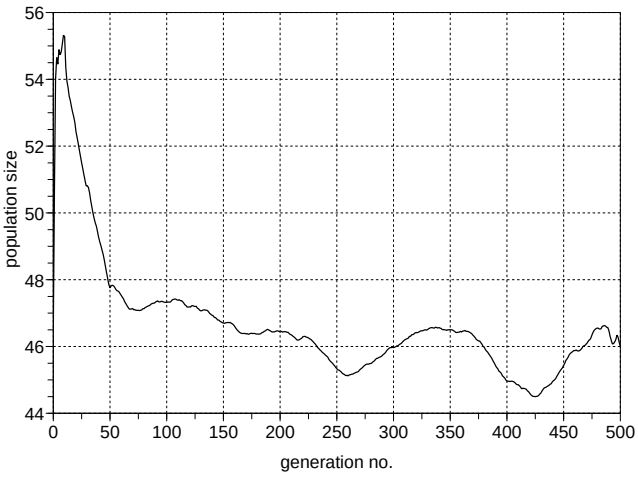


Fig. 24. Population size in CoEMAS during evolution

Fig. 24 presents population size in CoEMAS during evolution process. It shows that at the beginning initial population is small (about 47 individuals) but after three steps it becomes larger (up to 56 individuals). It is caused by the appearance of very good individuals in populations. They receive much resource from environment and create many equally good individuals in reproduction stage. Because the quantity of offsprings is greater than the quantity of dead individuals, the size of population increases. When the ability of an algorithm to find better and better individuals diminishes, the size of population also decreases. When the algorithm no longer finds new and much better solutions (after about 100 generations) then, the population size does not change significantly.

Fig. 25 shows the average fitness (from 30 experiments) of the best individuals for each generation. Presented results show that the evolutionary algorithm achieved the best results. The co-evolutionary algorithm achieved a slightly worse results. The quality of the solution generated by the CoEMAS was close to that of the CCEA.

Fig. 26 presents the plot of the convergence. The convergence is the phenomenon of losing by the evolutionary algorithm the ability to search the solution spaces before finding a solution which would be a global optimum. It is manifested by the occurrence of the pairs composed of identical individuals in the population. In the case of convergence the evolutionary algorithm had the worst results. Large convergence occurred already at the beginning and from 200 generation it was in the range from 50% to 60%. The co-evolutionary algorithm appeared to be much better. CoEMAS had the smallest

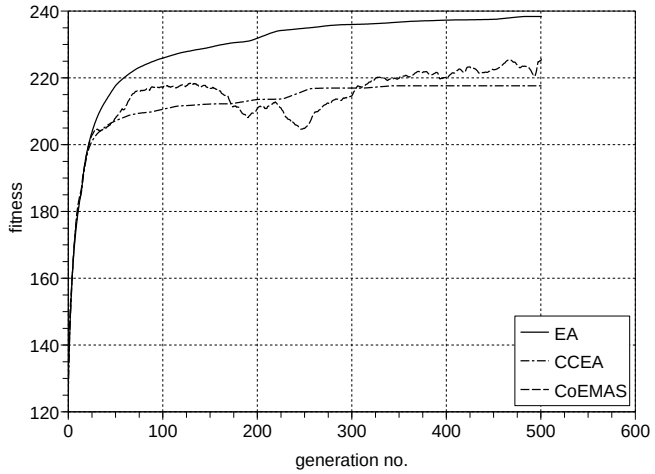


Fig. 25. Fitness of the best individuals (average values from 30 experiments)

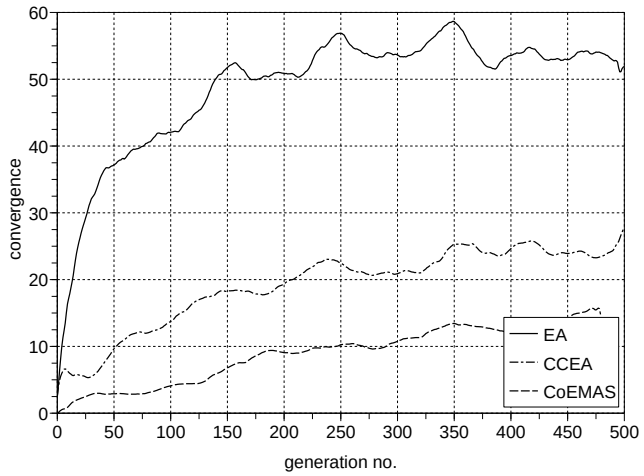


Fig. 26. Convergence values for three compared algorithms (average from 30 experiments)

convergence. For the last two algorithms convergence grew slowly from the beginning, but even in the end of the experiment it did not rise much.

Table 5. Comparison of efficiency, work time and convergence of all three algorithms

Algorithm	On-line efficiency	Off-line efficiency	The best fitness in last generation	On-line convergence	Convergence in last generation	Algorithm work time (m:s)
EA	229.888	230.229	238.369	48.807	51.910	1:47.574
CCEA	213.069	213.071	217.629	19.145	27.147	1:50.213
CoEMAS	214.744	238.741	226.170	8.851	14.785	9:29.872

Table 6. Generalization capabilities of the algorithms

Algorithm	No. of stocks in the group	Profit (%) per year (stocks from the group)	Profit (%) per year for buy & hold strategy (stocks from the group)	Profit (%) per year (random stocks)	Profit (%) per year for buy & hold strategy (random stocks)
EA	3	133.24	86.24	8.79	34.81
	5	89.15	28.85	1.8	33.62
	7	68.32	21.43	13.74	23.72
	10	87.57	24.29	20.81	26.07
CCEA	3	115.63	46.12	3.29	21.07
	5	64.82	8.83	7.39	31.36
	7	85.75	45.24	14.69	24.64
	10	69.06	26.37	20.75	25.14
CoEMAS	3	87.39	12.64	3.97	30.28
	5	66.97	7.64	-1.93	20.72
	7	86.93	39.48	19.01	34.47
	10	46.67	18.67	24.6	24.98

Table 5 confirms results presented in the figures. The evolutionary algorithm had the best efficiency and the worst results in the case of convergence, whereas the agent-based co-evolutionary algorithm was always better than the co-evolutionary algorithm with the exception of work time. The agent-based co-evolutionary algorithm had better off-line efficiency than the evolutionary algorithm because it found good solutions faster, however, those solutions were not good enough for the on-line efficiency to be better. Considering work times of all algorithms, both evolutionary and co-evolutionary algorithms came off quite well. The co-evolutionary algorithm is somewhat worse because it processes two populations. The agent-based co-evolutionary algorithm has work time six times longer than other algorithms. It is caused by the necessity of processing two co-evolutionary algorithms in two threads (computations were carried out on a machine with one processor). But the agent-based co-evolutionary algorithm can be easily distributed because of the decentralized nature of agent-based computations.

In Table 6 the results of experiments, which goal was to investigate the capability of generalization of all compared algorithms are presented. The results show that while generating a strategy, at least 7 stocks should be used, so that the strategy could be used on any stock and earn profits in any situation (see Table 6). If there are more stocks used during the strategy generation, the profit will be greater in the case of random stocks. For random stocks, when there was 3 or 5 of them in the group, the profits are varied and unstable. For this reason it is difficult to compare implemented algorithms with *buy and hold* strategy. It is not so, when the number of stocks in the group is 7 or 10. In the case of the random stocks, buy and hold strategy was always better (on average 2.67 times) than the strategies generated by all three evolutionary algorithms, but for the stocks from the learning set generated strategies were always better (on average 1.45 times) than buy and hold strategy.

5 Summary and Conclusions

Generating investment strategies is generally very hard problem because there exist many assumptions, parameters, conditions and objectives which should be taken into consideration. In the case of such problems finding the globally optimal solution is impossible in most cases and sub-optimal solution is usually quite sufficient for the decision maker. In such cases some (meta-)heuristic algorithms like biologically inspired techniques and methods can be used. In this chapter the system for generating investment strategies, which uses three types of evolutionary algorithms was presented. The system can generate strategies with the use of "classical" evolutionary algorithm, co-evolutionary algorithm, and agent-based co-evolutionary algorithm. These algorithms were verified and compared with the use of real-life data coming from the WIG index.

The presented results show that evolutionary algorithm generated the individual (strategy) with the best fitness, the second was agent-based co-evolutionary algorithm, and the third co-evolutionary algorithm. When the population diversity (convergence) is taken into consideration, the results are quite opposite: the best was CoEMAS, the second CCEA, and the worst results were reported in the case of EA. Such observations generally confirm that co-evolutionary and agent-based co-evolutionary algorithms maintain population diversity much better than "classical" evolutionary algorithms. This can lead to stronger abilities of the population to "escape" from the local minima in the case of highly multi-modal problems. High population diversity is also very desirable in the case of dynamic environments.

When we consider the generalization capabilities (profit gained from 7 and 10 random stocks during one year) of the strategies generated with the use of each evolutionary algorithm, the best results were obtained by CoEMAS (21.8% profit on the average), the second was CCEA (on the average 17.7%), and the worst results were obtained in the case of EA (17.3% on the average). Implemented algorithms provide better results than buy and hold strategy for stocks from the learning set and worse results in the case of the random stocks.

The future research could concentrate on additional verification of the proposed algorithms, and on the implementation and testing of other co-evolutionary mechanisms—especially in the case of the most promising technique: CoEMAS. Also, the implemen-

tation of the distributed version of the agent-based algorithm is included in the future plans.

References

1. F. Allen and R. Karjalainen. Using genetic algorithms to find technical trading rules. *Journal of Financial Economics*, 51(2):245–271, 1999.
2. T. Bäck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing and Oxford University Press, 1997.
3. A. Brabazon and M. O'Neill. *Biologically Inspired Algorithms for Financial Modelling*. Springer-Verlag, 2006.
4. A. Brabazon and M. O'Neill, editors. *Natural Computation in Computational Finance*. Springer-Verlag, Berlin, Heidelberg, 2008.
5. R. Dreżewski. A model of co-evolution in multi-agent system. In V. Mařík, J. Müller, and M. Pěchouček, editors, *Multi-Agent Systems and Applications III*, volume 2691 of *LNCS*, pages 314–323, Berlin, Heidelberg, 2003. Springer-Verlag.
6. R. Dreżewski and L. Siwik. Techniques for maintaining population diversity in classical and agent-based multi-objective evolutionary algorithms. In Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *Computational Science – ICCS 2007*, volume 4488 of *LNCS*, pages 904–911, Berlin, Heidelberg, 2007. Springer-Verlag.
7. R. Dreżewski and L. Siwik. Co-evolutionary multi-agent system for portfolio optimization. In Brabazon and O'Neill [4], pages 271–299.
8. S. G. Ficici. *Solution concepts in coevolutionary algorithms*. PhD thesis, Brandeis University, Waltham, MA, USA, 2004.
9. Historical stock data. http://www.parkiet.com/dane/dane_atxt.jsp.
10. S. K. Kassicieh, T. L. Paez, and G. Vora. Investment decisions using genetic algorithms. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 5. IEEE Computer Society, 1997.
11. O. V. Pictet, et al. Genetic algorithms with collective sharing for robust optimization in financial applications. Technical Report OVP.1995-02-06, Olsen & Associates, 1995.
12. J. Paredis. Coevolutionary algorithms. In T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation, 1st supplement*. IOP Publishing and Oxford University Press, 1998.
13. M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature — PPSN III*, volume 866 of *LNCS*, pages 249–257, Berlin, 1994. Springer-Verlag.
14. M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.
15. L. M. Tertitski and A. G. Goder. Method and system for visual analysis of investment strategies, December 2002. US Patent 6493681.
16. X. Yin. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufman, 1993.