Genetic Algorithms in a Visual Declarative Programming

EMILIA GOLEMANOVA[1], TZANKO GOLEMANOV[1]
[1]Department of Computer Systems and Technologies
University of Ruse
8 Studentska Str., Ruse
BULGARIA

Abstract: - Imperative languages like Java, C++, and Python are mostly used for the implementation of Genetic algorithms (GA). Other programming paradigms are far from being an object of study. The paper explores the advantages of a new non-mainstream programming paradigm, with declarative and nondeterministic features, in the implementation of GA. Control Network Programming (CNP) is a visual declarative style of programming in which the program is a set of recursive graphs, that are graphically visualized and developed. The paper demonstrates how the GA can be implemented in an automatic, i.e. non-procedural (declarative) way, using the built-in CNP inference mechanism and tools for its control. The CNP programs are easy to develop and comprehend, thus, CNP can be considered a convenient programming paradigm for efficient teaching and learning of nondeterministic, heuristic, and stochastic algorithms, and in particular GA. The outcomes of using CNP in delivering a course on Advanced Algorithm Design are shown and analyzed, and they strongly support the positive results in teaching when CNP is applied.

Key-Words: - genetic algorithms, declarative programming, visual programming, Control Network Programming

Received: April 14, 2021. Revised: April 17, 2022. Accepted: May 13, 2022. Published: June 21, 2022.

1 Introduction

While a lot of attention is usually being paid to a study or improvement in the various heuristic/stochastic operators of Genetic Algorithms (GA), a little effort is focused on how these algorithms can be implemented. Even though some authors demonstrate that implementation matters [1][2] and explore several popular imperative languages [3][4][5] and concurrent-functional languages [6] for implementing GA, the declarative approaches in new languages/paradigms are not investigated sufficiently.

This paper continues and extends the research on the implementation of genetic algorithms in Control Network Programming started in [7].

Control Network Programming, or **CNP,** is a multi-paradigm programming style, which combines features of imperative (procedural) programming, declarative (non-procedural) programming, and visual programming. The program in CNP, formally defined in [8], is a set of recursive graphs, called **Control Network (CN)**. In the CNP programming language **SPIDER,** and more precisely in **SpiderCNP** IDE [9][10], the program is graphically visualized and developed. The basic building blocks of the program, named **primitives**, are functions in some imperative programming language, and they form the arrows of the CN. As the CNP program is initially a graph, the imperative style programmer, i.e. the traditional professional programmer is not obliged to translate the intrinsic graph-like description of:

- *algorithms*, represented in their graphical form, similar to UML-activity diagrams or flow-charts, in their corresponding sequential textual form,
- and, more importantly, some *problems*, that possess nondeterminism or randomness, into their sophisticated sequential algorithmic solution. Instead, the CNP has a built-in inference engine (interpreter) that searches for a solution to the problem itself. It traverses the CN, corresponding to the nonlinear graphical description of the problem, and in this way, it resolves the nondeterminism. In addition, SPIDER has powerful built-in tools for control of the interpreter, allowing easily to implement heuristics and randomness.

In the first case, CNP enables the programmer with explicit program control, while in the second – with an "automatic", i.e. a declarative solution to the problem. The resulting programs are easy to develop and understand, which is important in Artificial Intelligence, where algorithms are usually nondeterministic, heuristic, or stochastic.

Genetic algorithms (GA) are typical examples of such kinds of algorithms. The purpose of the paper is

twofolded. On one hand, it expands the application area of CNP. SPIDER has already proven to be a very efficient programming language for the implementation of many heuristic and stochastic search algorithms in Problem Solving [11][12][13] and Constraint Satisfaction Problems [14]. Now we present the usage of SPIDER for programming genetic algorithms. Additionally, the intent of this paper is to demonstrate the SPIDER and more specifically, the SpiderCNP IDE, as convenient teaching and learning software for presenting the core concept layed behind the GA, both as a whole, and as its operators. The emphasis is on how to implement declaratively various selection methods, the crossover and mutation Bernoulli trials [15]. The SPIDER program is visually identical to the aforementioned operators' "natural" graphical descriptions, i.e. to the manner the developer thinks and specifies nondeterministic and randomized algorithms. Due to the fact that the program is intuitive, CNP implementations and SpiderCNP IDE respectively can be successfully applied in teaching GA.

There are two main approaches for using software in teaching an introduction to genetic algorithms [16] - using frameworks or libraries, and the second one - programming a simple genetic algorithm starting from scratch. For didactical purposes allowing students to make their own programs is the preferable approach, since the students have to assimilate the basic concepts in detail. Moreover, once a sense of control over the process is acquired, students can run the algorithms step-by-step and review the traces of the algorithm, analyze the effect of the operators, while using the capabilities of current IDEs in their own code. The best way to apply this approach is to use more expressive programming languages.

This paper presents the expressiveness of SPIDER and the suitability of SpiderCNP IDE in programming GA. This IDE has been used for several years in the Advanced Algorithm Design master course at Ruse University and has already proven to be very useful in teaching nondeterministic, heuristic, and randomized algorithms, including GA.

2 The basics of Control Network Programming

This part is a concise description of CNP and the methodology of programming in SPIDER. A more in-depth description of CNP - the theoretical model, CNP solutions to some exemplary problems, and SPIDER IDEs, can be found in [8][17][18].

What is known in traditional programming languages as a "program", in CNP is named a **Control Network (CN)** because it is a set of visually represented graphs (**subnets**). Each subnet can invoke another subnet, even invoking itself is viable. The starting point of the program is the so-called main subnet. The nodes of the subnet are named **states**, and the arrows are sequences of **primitives**. The states present the moments of computation and the primitives are elementary functions created by the CNP developer in some procedural, even object-oriented language. The computation in CNP is not deterministic, as it is in the imperative programming paradigm, but a graph traversal, executing the primitives along the way. The built-in interpreter (computation/search engine) uses an extended backtracking algorithm for searching a path from the start state of the main subnet to the system state FINISH. The primitives can be successfully or unsuccessfully executed - the conditions for a failure are defined by the programmer. When a primitive is unsuccessfully executed the control backtracks, causing the interpreter to change the direction of traversing and executing the primitives from the current arrow "backward" (performing "undo" of their actions made in a forward direction). Detailed and formal descriptions of the syntax and semantics of SPIDER can be found in [19][9].

We can approach solving problems in CNP in two ways. In the first approach, the CN is similar to a UML-activity diagram or a flowchart of the problem solution. In this case, the CN actually emulates an algorithm and, consequently, does not involve any nondeterminism. Having a pure imperative nature of the CN, we refer to these kinds of CNP implementations as **imperative** or **procedural implementations**, and CNP manifests itself as a universal programming paradigm. CNP offers a much different and more interesting approach when it is used as a declarative programming paradigm. In this case, the CN simply describes the problem (possibly involving nondeterminism or randomness) and does not require the development of an explicit procedure that implements the search process for finding the solution. Instead, the built-in search engine, along with a rich palette of system tools that control it, performs the desired strategy. Such a strategy could be even a heuristic or a stochastic one. These "automatic" implementations of search algorithms are called **declarative** or **non-procedural**. However, not all strategies are suitable for such an elegant non-procedural solution. Algorithms that can be directly implemented should be based on the backtracking strategy. Some more complex strategies, such as metaheuristic and genetic

algorithms, require a combination of both procedural and declarative techniques, and their implementations are called **hybrid**.

An example of a CNP program will be demonstrated using the second, more interesting

> *"There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around (if it is already at the box) and grasp the banana if standing on the box directly under the banana. Can the monkey get the banana?"*

approach for development – the declarative one, on a well-known AI toy problem - the **Monkey and Banana Problem** (MBP) as is stated in [20]:

The natural graph-like representation of this problem reflects its physical environment, i.e. the map of the room, and is depicted in Fig. 1. The nodes of this graph are the positions in the room (*Door*, *Middle*, and *Window*), and the arrows are the possible actions of the monkey from a given position (*Walk*, *Push*, *Climb*, or *Grasp*). This problem definition is nondeterministic, due to the presence of more than one arrow coming out of a node, that can be tried in searching for a solution.

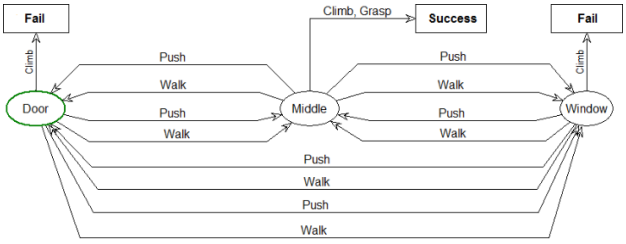


Fig. 1: Monkey and Banana Problem

The declarative solution of MBP (with an emphasis on CN, rather than on primitives' implementation), corresponding to its non-procedural specification from Fig. 1, is presented in Fig. 2. Both screenshots are from SpiderCNP IDE – the current CNP programming environment, equipped with an embedded graphic editor and debugger.

The CN consists of a main subnet **MonkeyAndBanana** and the subnet **Room**. The task of the main subnet is to initialize the variable *StartPlace* (the starting position of the monkey, e.g. the door in our case of problem definition), to call the subnet **Room** with the *StartPlace* as an initial point of traversal, and at the end – to print the solution. The second subnet **Room** "copies" the definition of the problem in its graphical form presented in Fig. 1. In other words, the subnet **Room** has a descriptive nature rather than a procedural one and the task of the built-in interpreter is to "compute" this subnet, searching for a path in a backtracking manner between the initial state and the system state RETURN.

Assigning the responsibility for finding the solution of the problem to the built-in inference engine, the CNP programmer, eventually, may want to use some of its **static control** tools (system options) in order to switch off/on backtracking, to prevent infinite loop and recursion, or to specify some parameters of the found solution paths, e.g. their number, length, or costs. For example, if an acyclic path of monkey's positions is required the option [LOOPS=0] should be used. Determining another system option [SOLUTIONS=ALL] forces the interpreter to find all the solution paths.

In addition to the static tools for controlling the basic parameters of the built-in search mechanism, the CNP programmer has the possibility to upgrade it in a more advanced strategy like its heuristic or stochastic variant. This can be achieved by rearranging, selecting, or reducing the outgoing arrows from a given state, randomly or according to their heuristic evaluations or selection probabilities. The programmer is provided with system tools for performing such action even when computation is in process. That's why these types of tools are referred to as **dynamic control** tools. In fact, to a large extent, it is the dynamic control tools that make CNP especially suitable for easy programming of heuristic and stochastic strategies. Description of the SPIDER toolkit for static and dynamic control of the main built-in search mechanism is made in [21][22], whilst its usage and methodology for modeling various advanced strategies are discussed in [23][24][25].

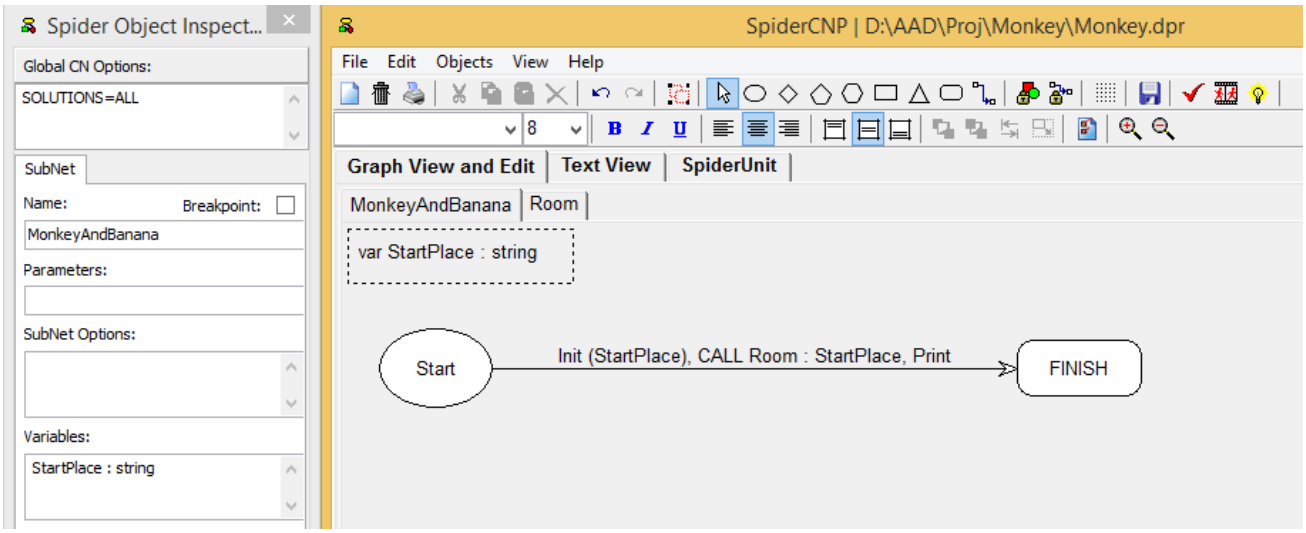


Fig. 2a: CNP program – Monkey and Banana Problem – the main subnet

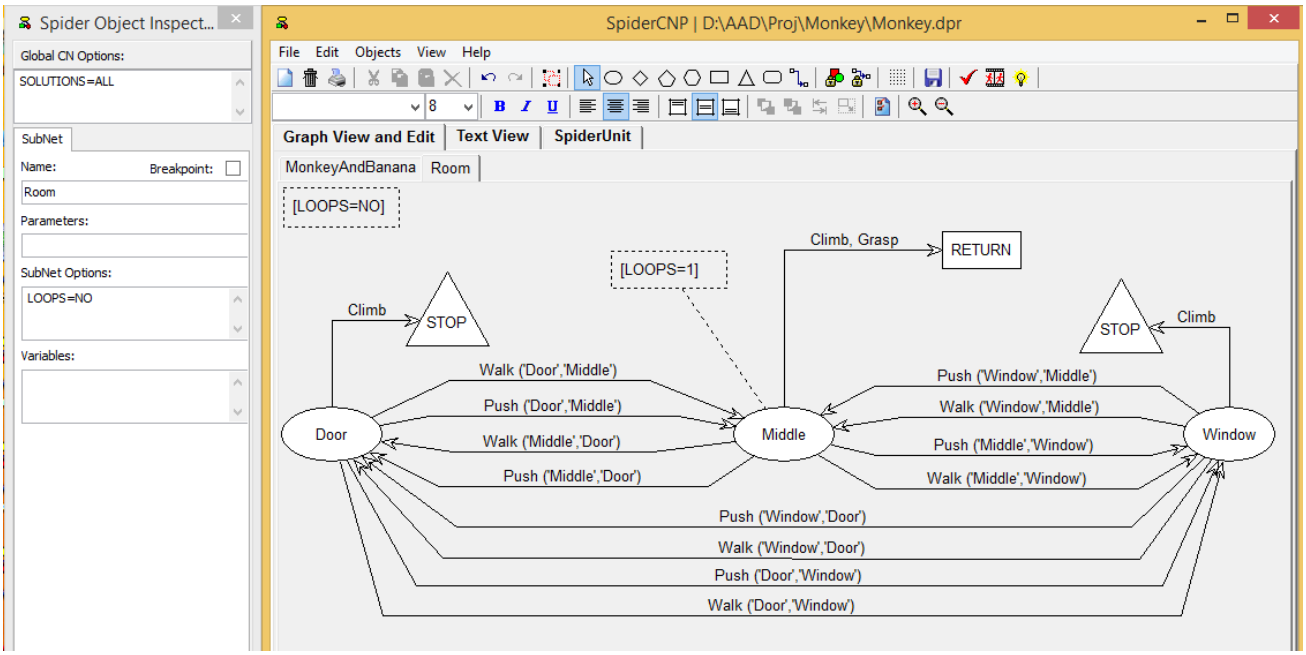


Fig. 2b: CNP program – Monkey and Banana Problem – the "Room" subnet

3. Simple Genetic Algorithm

Following the idea to demonstrate how the genetic algorithms, in their general form, are approached in SPIDER, the prime genetic algorithm named a Simple Genetic Algorithm (SGA) [20], is implemented.

The genetic algorithms theory was developed and published by John Holland in 1975 [26] and since then there have been many applications in science and economics. **Genetic algorithms (GA)** are stochastic, heuristic search algorithms that mimic the model of natural evolution to solve optimization problems. Some candidate solutions, i.e. feasible solutions [27] to the problem, form the so-called population of individuals (chromosomes) and compete for a survival based on their fitness (objective function). A generic structure of GA includes three basic operators - selection, crossover, and mutation. They are performed on the population, generating a new population, in the hope that it will be better, and ideally containing the global optimum. SGA simplifies offspring generation and parent replacement by using two non-overlapping populations. This "evolution" process is repeated until some termination criterion is met. Algorithm 1 shows the pseudocode of SGA (a version of [28][29]):

Algorithm 1: SGA

Result: best solution in the current population

[Start] Generate randomly, or heuristically (through a greedy algorithm), a population of n chromosomes (represented as strings) and evaluate their fitness.

while *the end condition is not satisfied* **do**

  [Generation] repeat

    [Selection] Select two parents from the population, with selection probability depending on a fitness function.

    [Crossover] Cross over the parents with a probability *Pc* and generate two new individuals.

      if *no crossover* **then**

        Copy the parents into children.

      end

    [Mutation] Change the two new chromosomes at each locus with a probability *Pm* and add them to the new population.

  until *the new population is complete;*

  if *n is odd* **then**

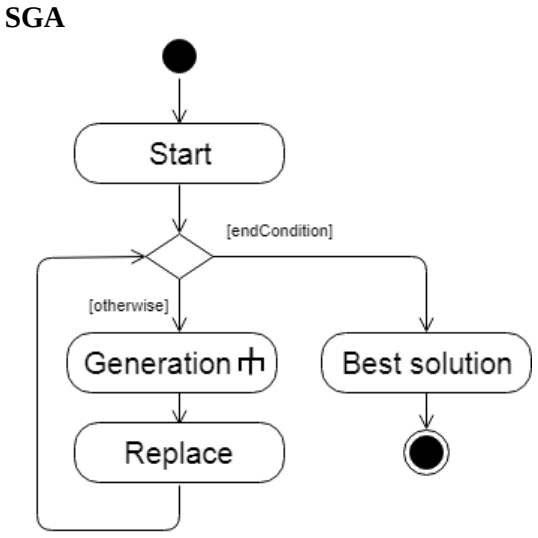
    Delete randomly a new population member.

  end

  [Replace] The new population becomes a current population. Keep the best individual from the old population (elitism).

end

This generic idea of GA is very often presented in graphical forms [30] [31], similar for example to the UML-diagram depicted in Fig. 3:



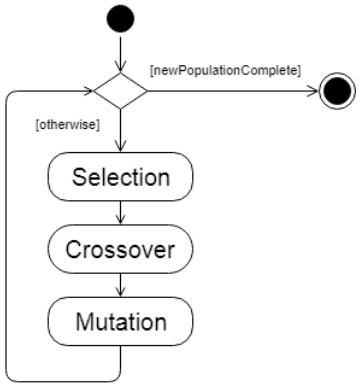


Fig. 3: The Simple Genetic Algorithm

The mathematical modeling of GA is presented in [32], and the computational complexity of SGA was analyzed by Oliveto and Witt and has been proven to have exponential runtime with overwhelming probability for population sizing up to $\mu \leq n^{1/8-\varepsilon}$ for some arbitrarily small constant ε and problem size n [33].

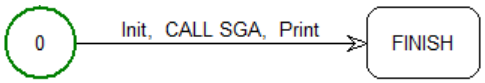
4 Programming SGA in SPIDER

Intending to demonstrate the CNP programming methodology applied to genetic algorithms we have implemented the SGA on an example toy-problem, namely the **8-queens**. Specifically, we use the problem instance, as it is presented in figure 4.6 from [34], which is defined onto a population of four chromosomes. As the algorithm's termination criterion the number of generations was chosen.

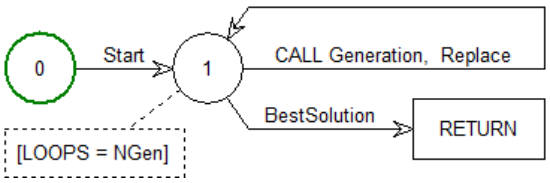
The population is represented as an array of four 8-digit strings along with their fitness values. Each digit, which is a number from 1 to 8, is the column of the chess board where the queen is placed (the row is determined by the index of the array). The objective function calculates the number of pairs of queens that do not attack each other and is to be maximized, with a maximum value of 28 for a solution.

In the previous section, the SGA was presented in the form of a UML diagram which obviously has a graph-like structure. Hence, it is easy to translate it into a CN. The conversion is in an almost trivial manner – the initial, final, and conditional UML blocks "become" states of the CN, while the operational UML blocks form the arrows of the CN. As a result, the CN depicted in Fig. 4, mirrors the algorithm's specification from Fig. 3:

main MainNet;



sub SGA;



sub Generation;

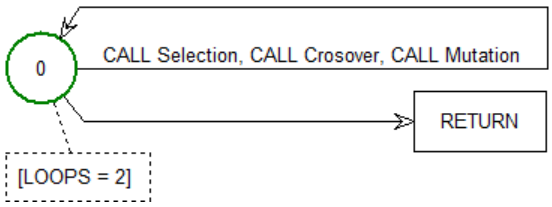


Fig. 4: SPIDER implementation of SGA

The main subnet **MainNet** has an initialization purpose mainly. The primitive **Init** initializes SGA parameters (declared as global variables) - the number of generations (*NGen*), crossover probability *Pc*, mutation probability *Pm,* and their inverse values (*InvPc* and *InvPm*). Note that the concrete programming implementations of the primitives in the presented CNP solution will not be discussed as they have simple logic and could be easily implemented in any imperative programming language. **MainNet** calls the subnet **SGA** and prints the solution found. The subnet **SGA** implements the main steps from the Algorithm 1 – creation of the initial population (primitive **Start**), generation of a new population (subnet **Generation**), and replacing the current population with the new one (primitive **Replace**). The population renewal is performed *NGen* number of times. It is easily achieved by the system option [LOOPS=NGen] for the state 0. In the general case, the option [LOOPS=n] is designed to limit the repeated visits to a given state, although its most common use is to prevent loops. The solution to the problem is found by the primitive **BestSolution** as the best-fit individual in the final population. The subnet **Generation** implements the inner loop of the SGA - iterations over the offspring generating until the new population is filled in. Therefore, it has similar to the subnet **SGA** architecture – it binds the basic SGA operators – selection, crossover, and mutation – and repeats them two times, through the system option [LOOPS=2], in order to generate four children.

In the described part of the CNP implementation (Fig. 4) we have used the first approach for CNP programming – the procedural. What follows is the demonstration of the second approach – the declarative one, in implementing the randomness and heuristics in GA. As it is well known, the GA can be viewed as a random heuristic search in the search space of the problem, guided by the "intelligent ideas" of the nature - selection, crossover, and mutation. One of the main features of the randomized, heuristic, and nondeterministic computation, is the existence of so-called "choice points" [27][35], i.e. the points where a choice of the way to proceed is to be done. These choice points have a natural graph-like representation. The stochastic heuristic nature of the three GA operators is determined by the incorporation of randomness and heuristics in their choice points – in the selection of the mating parents and in the decisions to perform (or not) a crossover and a mutation. In SPIDER there is no need for the programmers to implement the randomness and heuristics themselves. Instead, programmers simply specify the choice points with all the alternatives as states in CN with outgoing arrows, corresponding to these alternatives. Then they (the programmers) can use built-in tools (control states and system options), which model a random or a heuristic choice of the emanating arrow to be traced. The resulting SPIDER implementations are declarative and correspond to the natural understanding of nondeterministic, heuristic, and random choice because they keep the graphical representation of the choice points.

The declarative implementations of the three operators are discussed in detail in the following subsections.

4.1 Selection

The parent selection operator determines how to choose the individuals of a population for crossover. In Darwin's theory of evolution, the individuals with high fitness have a higher probability to be chosen to reproduce. A wide variety of selection strategies have been proposed, like **Roulette Wheel Selection**, **Rank Selection, Tournament Selection**, **Truncation Selection**, **Threshold Selection, Boltzman Selection**, and **Stochastic Universal Sampling** [36]. Most of them are very suitable to be declaratively programmed in SPIDER and how to achieve these implementations will be described below.

4.1.1 Roulette Wheel Selection

Roulette Wheel Selection (RWS) is the most common approach for applying fitness-proportionate selection [28], i.e. the scheme where the probability of an individual being chosen is proportional to its fitness. It simulates the roulette wheel operation - the individuals "occupy" areas of the wheel proportional to their fitness values and then the wheel is spun. The wheel pointer determines the individual which is selected for mating. In the example under consideration, where the population size *n* is 4 individuals, it is necessary to spin the wheel four times – two times for each of the two couples of parents.

The presented idea can be easily implemented in SPIDER through a control (not ordinary) state of a type, named SELECT. In SPIDER, in the usual case the outgoing arrows from an ordinary state are traversed in the order they are defined in the CN. But there are three types of **control states** (SELECT, ORDER, and RANGE) that allow the predefined order of arrows to be dynamically changed and controlled according to some "heuristic" evaluations of those arrows. The SELECT state has a property named a **selector** and only the arrows with an evaluation identical to the selector are selected to be examined. In addition, the system options SELECTMODE and PROXIMITY allow refining, "tuning" this choice. The graphical sign of SELECT state is a rhombus.

The selection of the parent couple is modeled through the SELECT control states **PARENT_1** and **PARENT_2**, presented in Fig. 5. The primitive **GetFitnesses** initializes the variables F_i, $i \in \{1, .., 4\}$ with the fitness values, which play the role of arrow evaluations of **PARENT_1** and **PARENT_2**. Both states use the variable **WheelPoint** as a selector. The system option [SELECTMODE=ROULETTE] determines the selection of the active arrow to be in accordance with the principle of roulette with a wheel pointer, presented by the state selector **WheelPoint** (random number).

The selected parents are stored in the global variables *Parent1* and *Parent2* by the primitives **SetParent1** and **SetParent2**. The parameter of these primitives is the index of the selected chromosome from the population.

sub Selection;

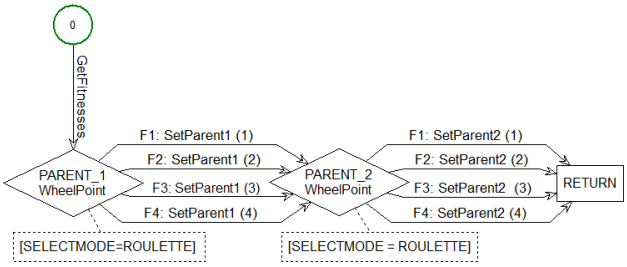


Fig. 5: SPIDER implementation of RWS

4.1.2 Rank Selection

Rank Selection is the other classical selection operator, built-in in SPIDER. It selects parents according to their ranks, i.e. the worst chromosome has a fitness 1, the next one 2, etc., and the best will have a fitness *n*. Therefore, it utilizes the relative instead of the absolute fitness value. It doesn't matter what the fitness ratio is between the fittest individual and the next one - their selection probabilities would be the same in all cases. The distinction between RWS and Rank Selection is illustrated (Fig. 6) in the following figure from [28]:

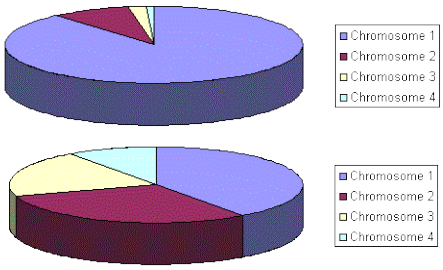


Fig. 6: RWS vs Rank Selection

The implementation of Rank Selection in SPIDER could be achieved straightforwardly by a slight modification of the program depicted in Fig. 5 – simply by using the option [SELECTMODE=RANK] instead of [SELECTMODE=ROULETTE]. The fragment of the SPIDER solution for one of the parents is depicted in Fig. 7:

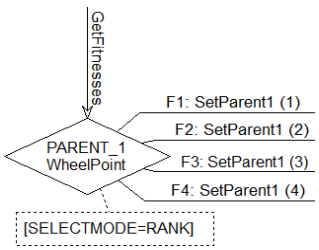


Fig. 7: Fragment of the SPIDER implementation of Rank Selection – choosing the first parent

4.1.3 Tournament Selection

The RWS and Rank Selection methods described above are time-consuming procedures as they require a computation of the fitness value of each individual in the population, and/or sorting of the entire population. Tournament Selection is similar to RWS as an idea, but it is computationally more efficient. It involves choosing *k* individuals from the population randomly who are participants in the tournament and selecting deterministically or stochastically the winner from that group.

The implementation of the Tournament Selection in CNP will be presented just for the first parent from the mating pair (Fig. 8).

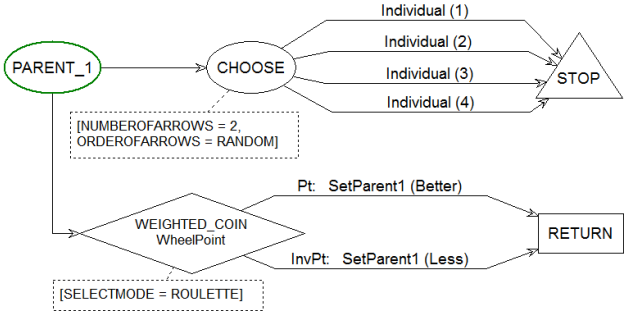


Fig. 8: Fragment of the SPIDER implementation of Stochastic Binary Tournament Selection – choosing the first parent

The random selection of the participants in a binary tournament (when $k=2$) is achieved using system options [ORDEROFARROWS=RANDOM] and [NUMBEROFARROWS=2] determining that the outgoing arrows from the state **CHOOSE** will be rearranged randomly and two of them will "survive". The primitive **Individual** records the individual, specified by the primitive's parameter, in the global structure *Tournament_Participants* and calculates its fitness.

Tournaments can be either deterministic, in which the best solution is always selected, or stochastic, where less fit solutions may be probabilistically chosen. In the *stochastic* binary tournament, after two individuals are picked out of the population, a weighted (biased) coin is then tossed. The idea of the weighted coin toss (Bernoulli trial), coming up heads with some probability P_t, is usually implemented as follows: a randomly generated number $0 \le r \le 1$ is compared with P_t. If $r < P_t$, the better individual is chosen, otherwise - the other one. The probability P_t is a parameter of the algorithm, which could be fixed, for example, 0.75, or depending on the run, like in the Boltzman tournament, which has clear similarities to simulated annealing. In fact, the Bernoulli trial is equivalent to RWS with only two sectors – with selection probabilities P_t, and respectively $1\text{-}P_t$. The random number r is the wheel pointer. This is implemented in SPIDER (Fig. 8) by the SELECT-type control state, named **WEIGHTED_COIN,** and the system option [SELECTMODE=ROULETTE]. The two outgoing arrows are labeled with evaluations P_t and *InvPt*. The primitive **SetParent1** records the winner of the tournament as the first parent. The primitive's parameter identifies which one of the two tournament participants is selected.

The CNP-implementation of the *deterministic* tournament will be presented in a more general case – *k*-tournament. After the *k* individuals are chosen from the population, the best one is selected through a control SELECT-state whose selector is equal to the maximum fitness value. The system option [PROXIMITY=NEAREST] will cause the arrow labeled with a value F_i, $i \in \{1, .., k\}$, closest to the value of the selector to be chosen. Fig. 9 is a fragment of the SPIDER implementation illustrating this idea in the case of $k=3$ (see the control state **TOURNAMENT**).

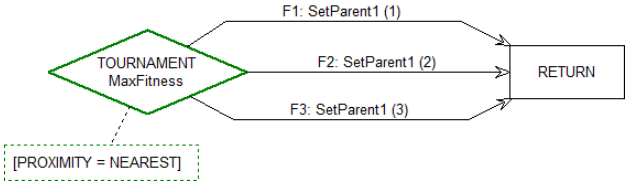


Fig. 9: Fragment of the SPIDER implementation of Deterministic Tournament Selection

The correspondence between the graphical representation of the Tournament Selection from [37] (Fig. 10), usually used in the explanation of the idea, and the CNP implementation is evident. This results in easy programming and a better understanding of the algorithm.

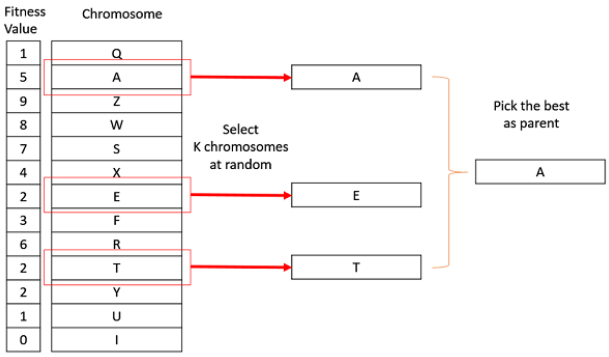


Fig. 10: Tournament Selection

4.1.4 Truncation Selection

In the previously described selection methods the mating pool is gradually filled, while in the Truncation Selection and Threshold Selection it is generated at once.

In the Truncation Selection, only a percentage *p* of the population participates in the crossover, i.e. the individuals are sorted according to their fitness values, and then some proportion *p* of the best ones is chosen to reproduce.

This idea is easily implemented in SPIDER by a control state of another type – ORDER, graphically represented as a pentagon. Its outgoing arrows are traversed according to the proximity of their estimates to the state selector. Stating this selector with the biggest possible fitness value will cause ordering the individuals, i.e. the arrows, in decreasing order of their fitness – the fittest one will be attempted first. The truncation of the non-perspective individuals is accomplished by the system option [NUMBEROFARROWS=*n***p*], where *n* is the size of the population. This idea is applied to the 8-queens problem under discussion assuming *p=1/2*. The resulting fragment of CN is depicted in Fig. 11. The best half of the population, respectively the best half of arrows emanating from the state **TRUNCATION**, will be taken into account.

sub Selection;

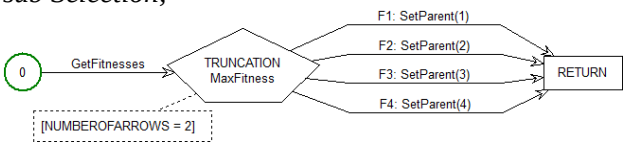


Fig. 11: SPIDER implementation of Truncation Selection

4.1.5 Threshold Selection

In Threshold Selection individuals that are below the threshold fitness value are not examined. This variant of the Truncation method determines the fraction of the population to be chosen for reproduction, not according to a number of individuals, but to a fitness bound.

In SPIDER there is another, very powerful control state, named RANGE, and graphically represented as a hexagon, which cuts off the arrows, according to their evaluation values. It has two parameters – lower bound *L* and upper bound *H*, and only the arrows whose evaluations are in the range *[L, H]* "survive". Therefore, this type of control state with a specification of just one of the selectors is an appropriate tool for implementing the idea of a Threshold Selection. The corresponding SPIDER implementation for the 8-queens problem is presented in Fig. 12. As the fitness function is to be maximized, the threshold of the selection method is set as a lower bound, respectively the selector **LowBound** of the control state **THRESHOLD.**

sub Selection;

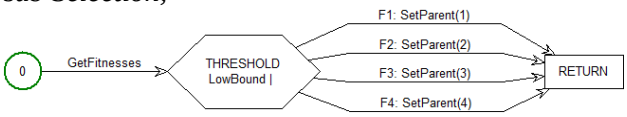


Fig. 12: SPIDER implementation of the Threshold Selection

Furthermore, in SPIDER we can determine the order of entering the individuals in a mating pool, specifying the value of the system option RANGEORDER. This allows the mating pool to be created in increasing or decreasing order of the fitness values, or randomly.

4.2 Crossover

Once a couple of parents is selected, they cross over with a probability *Pc* to get two children. If no crossover occurs, then the children copy their parents. This concept could be modeled by a weighted coin toss, i.e. a Bernoulli trial with probability *Pc*. As it has already been demonstrated in the previous section, the weighted coin toss is easily simulated by an RWS method with two roulette segments, which may be chosen with probabilities *Pc,* and respectively *1-Pc*. Realizing that, the SPIDER implementation of the crossover operator is straightforward.

sub Crossover;

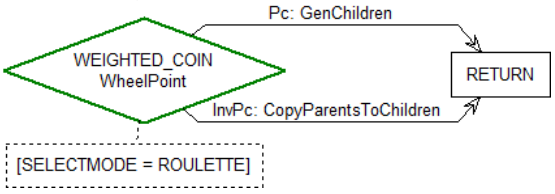


Fig. 13: SPIDER implementation of Crossover

The depicted in Fig. 13 subnet **Crossover** works on two parent's chromosomes and produces as an outcome two offspring chromosomes. The method of a weighted coin toss, determining whether the crossover on the parent pair will be performed, is implemented by the SELECT-type control state **WEIGHTED_COIN** with two emanating arrows - with selection probabilities *Pc,* and correspondingly *1-Pc.* Which of these two alternatives will be chosen is determined according to the Roulette wheel method by setting the system option

[SELECTMODE=ROULETTE]. The primitive **GenChildren** executes the crossover itself. If no crossover occurs, i.e. the second arrow is active, the primitive **CopyParentsToChildren** copies parents (variables *Parent1* and *Parent2*) into children *Child1* and *Child2*.

4.3 Mutation

The two offspring chromosomes then undergo a mutation process (Fig. 14).

sub Mutation;

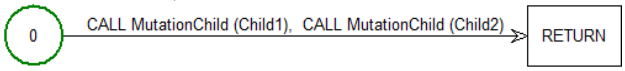


Fig. 14: The "Mutation" subnet

In SPIDER, similarly to primitives (and to functions in programming languages), the subnets could have parameters. This feature serves very well in use cases where we need to "execute" subnets with different data. The formal parameter *Child* of the subnet **MutationChild** (Fig. 15) determines on which chromosome the mutation is performed.

sub MutationChild;

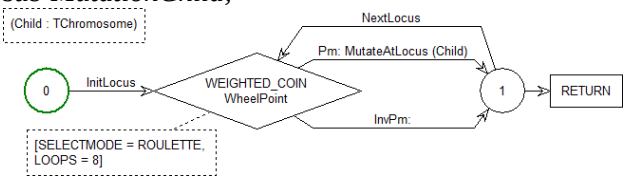


Fig. 15: SPIDER implementation of Mutation

The mutation is defined as a modification with some usually very small probability *Pm* at each locus in the chromosome. The global integer variable *Locus* ranged from 1 to 8 is processed by the primitives **InitLocus** and **NextLocus**. The probabilistic mutation at a given locus is performed again by a Bernoulli trial and implemented in SPIDER as a SELECT-control state **WEIGHTED_COIN** with an option [SELECTMODE=ROULETTE]. In addition, the system option [LOOPS=8] determines how many entries are allowed to this state, corresponding to the number of genes in the chromosome.

5 Students' Opinion

SPIDER programming and SpiderCNP IDE have been used for five years in a graduate course on Advanced Algorithm Design which deals with the design techniques that cope with hard problems (non-polynomial time problems). The CNP approach has been introduced and implemented as an instrument for the realization of more advanced design techniques such as intelligent exponential search, randomized algorithms, heuristic algorithms, and local search.

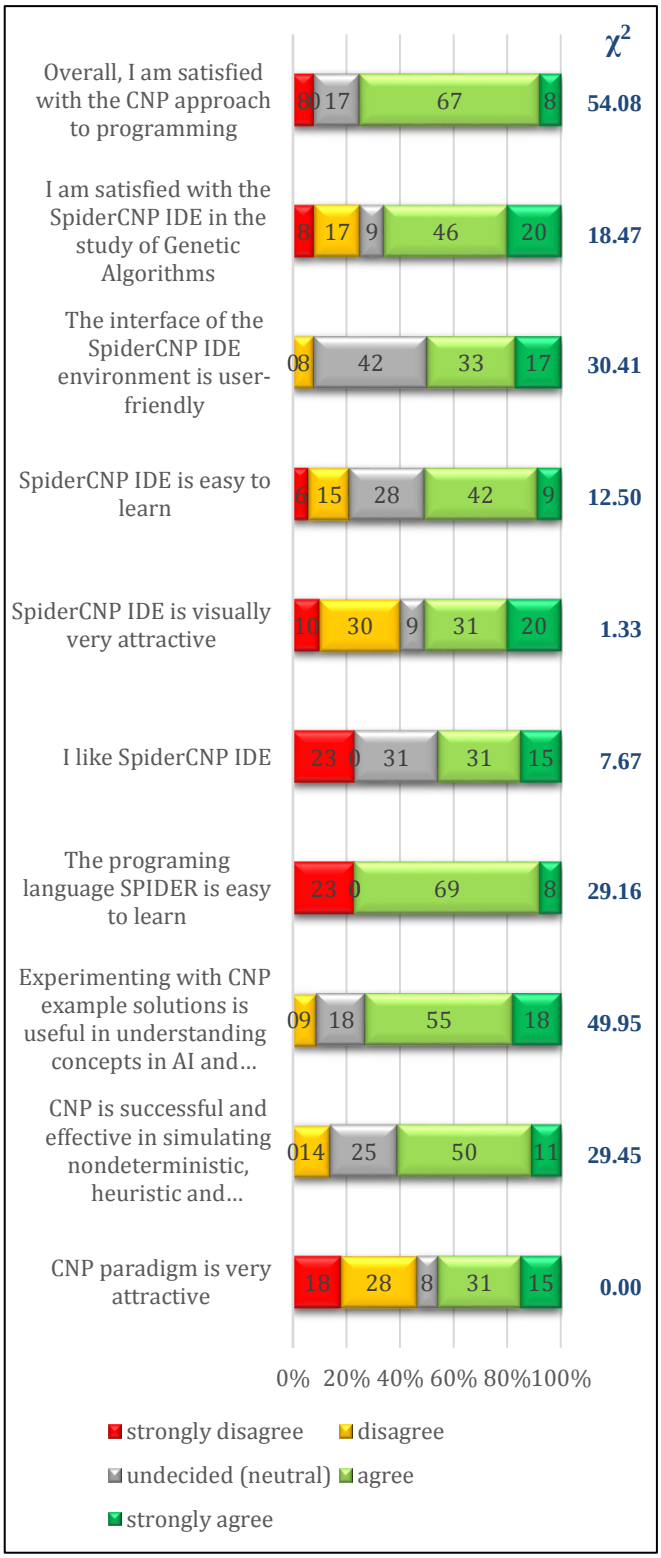


Fig. 16: Likert Scale multiple choice positive questions' results

To get a student's feedback on the effectiveness of SpiderCNP as an educational environment and to

improve it, in the last four years, we conducted qualitative questionnaires together with the students. The survey was carried out on a sample of 103 Master's degree students. It includes two types of multiple-choice questions.

The summary quantitative data of the 5-Likert Scale multiple-choice questions are presented in percentages in Fig. 16 and Fig. 17, for positive and negative questions, respectively:

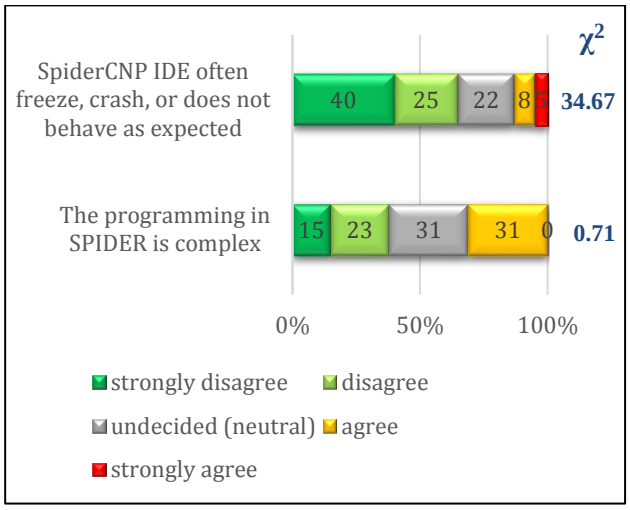
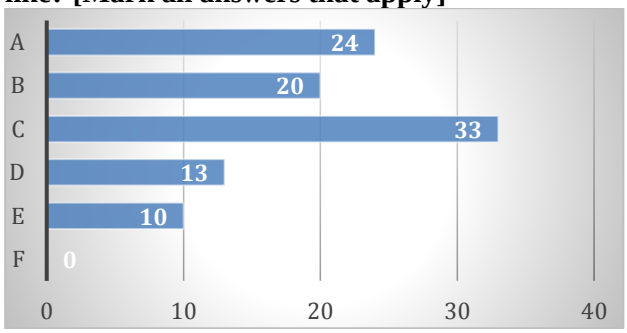


Fig. 17: Likert Scale multiple choice negative questions' results

The quantitative results from the second type of closed questions are presented graphically in Fig. 18 and Fig. 19:

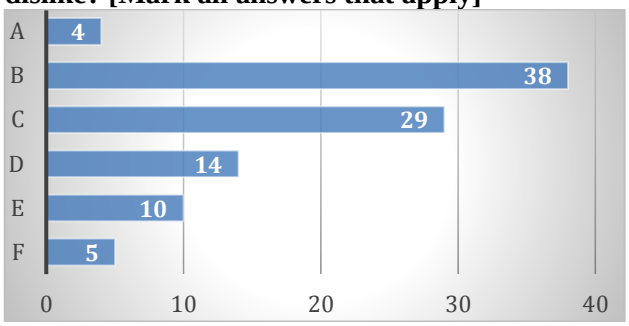
Which of the following features of CNP do you like? [Mark all answers that apply]



A: *Programming by drawing graphs*
B: *Flow of control is clear and understandable*
C: *The programmer need not bother about the existence of multiple possible solution paths*
D: *Powerful means for computation (inference) control are available*
E: *The programming environment helps me develop the solution easily*
F: *Other*

Fig. 18: Question chart: "Which of the following features of CNP do you **like**?"

Which of the following features of CNP do you dislike? [Mark all answers that apply]



A: *Installation is difficult*
B: *Understanding is difficult*
C: *Writing and testing primitives is difficult*
D: *Drawing and editing the Control Network using a graphical editor is annoying*
E: *The separation of the control (the Control Network) and the primitives used is counter-intuitive and confusing*
F: *Other:*

- ✓ *"We come from a traditional programming background, so it is slightly easier for us to do things using imperative and object-oriented programming styles. I accept that the CNP is easy to use. C, C++, Java, etc. might be more complicated than CNP, but they are much more flexible and fast."*
- ✓ *"In CNP, we draw graphs, and then the GUI generates the code behind it. It might seem nice, but it is not."*
- ✓ *"We were given just some simple problems. Real-life problems would be really complicated, and someone cannot draw all of them by hand."*
- ✓ *"In addition, there is no updated version of CNP in MacOS and Linux. It works only on Windows. IMHO, you should have written a plug-in to Eclipse or Netbeans, then it can run on any platform. Or, just make it open source maybe."*
- ✓ *"The habits can be the reason but I don't prefer CNP unless my work is related to AI. It is good to visualize the working progress of the program basically with arrows and maps. But because of the insufficient map-drawing tool, it takes much time to draw the map. It is not reasonable for me to make an effort and spend much time only to see the progress of the program."*

Fig. 19: Question chart: "Which of the following features of CNP do you **dislike**?"

The survey response data from the Likert scale is analyzed through Chi-square statistical test, which is used in order to examine the differences between actual responses and expected responses. Chi-square evaluates the statistical significance of a particular hypothesis. We set the null hypothesis as follows: "There is no difference in the proportion of 'agree' and 'disagree' answers.". Thus, we combine the answers 'strongly agree' and 'agree' into one category, 'strongly disagree' and 'disagree' into another, and skip the neutrals. The χ^2 statistic displayed in Fig. 16 and Fig.17 is calculated for each of the questions: $\chi^2 = \sum(O_i - E_i)^2/E_i$, where O_i is the observed (actual) value and E_i is the expected value. The critical χ^2 value is 3.84 in the case of the significance level 0.05 and the degrees of freedom 1. If the calculated χ^2 value is greater than the critical χ^2 value, there is strong evidence to reject the null hypothesis of 'no difference'. Therefore, it is concluded that in most of the questions there is a significant difference in the proportion of 'agree' and 'disagree' responses.

The survey conducted with our students shows that 75% of them are satisfied with the CNP approach to programming. Other results reveal that 61% think that CNP is successful and effective in simulating nondeterministic, heuristic, and randomized algorithms while 66% find SpiderCNP IDE helpful in studying genetic algorithms. According to 51% of answers, SpiderCNP IDE is visually very attractive and easy to learn, while the most significant difficulty is coding the primitives and encountering a new very different programming paradigm. The most liked features of CNP (77% of students) are programming through drawing graphs, the understandable control flow, the lack of necessity to work with nondeterminism, and the existence of multiple possible solution paths.

Based on the presented results, it can be assumed that the overall impression is positive. According to the majority, the CNP, SPIDER, and SpiderCNP IDE are useful and supportive for students in their efforts to apprehend and master the uneasy concept of genetic algorithms.

6 Discussion

In this section we summarize and comment on the results of the presented study and its contributions and limitations in comparison with the corresponding studies.

While imperative programming has been extensively studied and used to implement GA [3], exploration of the feasibility of the declarative paradigm is not a focus of attention for software researchers. The efforts to implement declaratively GA are aimed at logic programming in Prolog and programming in languages with functional features. Prolog was used many years ago [38] [39] in the implementation of GA. Despite the advantages presented over imperative languages such as compact code and built-in 'don't care' operator, the approach is of limited use due to the limited control means of the logic inference mechanism. More recent research [6] explores the feasibilities of some concurrent-functional languages like Erlang, Scala, Clojure, etc. to develop GA, but in its parallel versions.

SPIDER is distinguished from the conventional imperative and declarative approaches by the following features:

- SPIDER applications are multi-paradigm projects in which the imperative object-oriented paradigm is enhanced by adding means for programming nondeterministic, heuristic, and genetic algorithms.
- SPIDER program is visually presented as a set of recursive graphs. GA is implemented in the same way. The graphical declarative specification of SGA reflects clearly the logic of the algorithm.
- SPIDER provides built-in 'RWS' and 'Rank' operators. Other heuristic and stochastic system tools allow experimentation with different genetic operators to obtain a wide variety of GA.
- Imperative and functional languages may be trickier to debug GA programs. Unlike them, SPIDER supports a visual graphical tracing facility, i.e. step-by-step tracking of the computation flow on the graph program.
- The graphical presentation of the program and the possibility of graphical debugging allow SPIDER, and more specifically SpiderCNP IDE, to be used as appropriate software for teaching and learning GA.

As a result of the study of the feasibility of SPIDER for GA programming in solving complex problems and in GA teaching, some limitations of the CNP approach have been noticed:

- The SPIDER modeling of choice points with a lot of alternatives leads to difficulties in their graphical representation.
- The traditional programming background of students is imperative and the development of declarative programs is slightly unusual for them.
- Some students find difficulties in the visual development of a program by drawing graphs instead of program textual specification.

7 Conclusion

Many problems, especially those involving nondeterminism, are usually presented in the form of a graph, because of their innate nonlinear structure. SPIDER and its supporting visual IDE - SpiderCNP were developed in response to the desire to eliminate the "representational gap" and allow the program to be a graph, moreover, the program could be a recursive network. Unlike another declarative programming language – PROLOG, where the program also has the form of a graph, but it is implicit, in SpiderCNP the program is represented graphically which is more comprehensive and easily verifiable. It is unnecessary for programmers to cope with the nondeterminism, instead, the built-in interpreter does it for themselves. Furthermore, the programmers are enabled with a powerful visual toolkit for interpreter control which allows for automatic, i.e. declarative implementation, not only of nondeterministic but also heuristic and randomized strategies [11][40][13].

GA are examples of such kinds of algorithms. The fundamental operators of GA – selection, crossover, and mutation have choice points where a choice (often random or heuristic) of the way to proceed is to be done. These choice points have an inherent graph-like representation, which helps for a better understanding of the idea. Keeping this graphical description in the SPIDER implementation and automating the stochastic and heuristic choice through built-in tools, makes this implementation easily programmed and intuitive.

In our teaching experience, the notion of "genetic algorithms" causes significant difficulties for students due to the complexity of this type of metaheuristics. SpiderCNP IDE not only allows for easy programming of the basic GA but also enables students to experiment with the different methods for selection, crossover, and mutation (even ones developed by themselves) using the rich palette of heuristic and stochastic SPIDER tools. The resulting SPIDER programs are visually identical to their "natural" graphical descriptions, i.e. to the manner the developer thinks and specifies nondeterministic and randomized algorithms. This makes the concept of GA more comprehensive. Due to the fact that the resulting CNP programs are intuitive, SPIDER implementations and SpiderCNP IDE respectively can be successfully used in teaching GA. Our current experience in delivering a course on Advanced Algorithm Design strongly supports this statement.

We are currently investigating two main enhancements to SPIDER. We are adding system tools to provide other variants of GA operators, e.g. Boltzman Selection and Stochastic Universal Sampling. Thus the language will cover a wider class of GA and will increase the flexibility of the SpiderCNP IDE as a teaching software. Secondly, CNP can be extended to other programming languages. Currently, the host languages of SPIDER are Delphi Object Pascal and Lazarus Free Pascal. C++-based SPIDER is under development and the other candidates could be C#, Java, and Python.

Another direction for future efforts could also be the improvement of SpiderCNP IDE.

Acknowledgments

The paper is supported by project No. 2022-EEA-01, funded by the Research Fund of the University of Ruse.

References:

[1] J. L. J. Merelo, J.J., Romero, G., Arenas, M.G., Castillo, P.A., Mora, A.M., Laredo, "Implementation Matters: Programming Best Practices for Evolutionary Algorithms," in *Advances in Computational Intelligence. IWANN 2011. Lecture Notes in Computer Science, vol 6692*, G. Cabestany, J., Rojas, I., Joya, Ed. Springer, 2011, pp. 333–340.

[2] G. J.-J. Merelo, M. García-Valdez, and S. Rojas-Galeano, "Implementation matters, also in concurrent evolutionary algorithms," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 1591–1598.

[3] J. et al. Merelo-Guervós, "Ranking Programming Languages for Evolutionary Algorithm Operations," in *Applications of Evolutionary Computation. EvoApplications 2017. Lecture Notes in Computer Science, vol 10199*, G. Squillero and K. Sim, Eds. Springer, 2017, pp. 689–704.

[4] E. Wirsansky, *Hands-On Genetic Algorithms with Python*, 1st editio. Packt Publishing, 2020.

[5] I. Gridin, *Learning Genetic Algorithms with Python*. BPB Publications, 2021.

[6] J. A. Cruz, J. J. Merelo, L. Acevedo-Martínez, and P. Cuevas, "Implementing Parallel Genetic Algorithm Using Concurrent-functional Languages," in *Proceedings of the International Joint Conference on Computational Intelligence - Volume 1 (IJCCI 2014)*, 2014, pp. 169–175.

[7] E. Golemanova and T. Golemanov, "Declarative Implementations of Genetic Algorithms in Control Network Programming," in *Computer Systems and Technologies*, 2019, pp. 91–97.

[8] K. Kratchanov, "CINNAMONS: A Computation Model Underlayng Control Network Programming," in *7th International Conference on Computer Science, Engineering & Applications (ICCSEA 2017)*, 2017, pp. 1–20.

[9] T. Golemanov, "Development and Study of an Integrated Development Environment for Control Network Programming, Ph. D Thesis," Ruse University, 2014.

[10] T. Golemanov, "SpiderCNP: An Integrated Environment for Visual Control Network Programming," in *Annals of Ruse University, 51, ser. 3.2*, 2012, p. 123–127 (in Bulgarian).

[11] K. Kratchanov, E. Golemanova, T. Golemanov, and Y. Gökçen, "Implementing Search Strategies in Winspider II: Declarative, Procedural, and Hybrid Approaches," in *Knowledge-Based Automated Software Engineering*, I. Stanev and K. Grigorova, Eds. Cambridge Scholars Publishing, 2012, pp. 115–135.

[12] K. Kratchanov, E. Golemanova, T. Golemanov, and T. Ercan, "Non-procedural Implementation of Local Heuristic Search in Control Network Programming," in *14th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2010)*, 2010, pp. 263–272.

[13] K. Kratchanov, E. Golemanova, T. Golemanov, T. Ercan, and B. Ekici, "Procedural and Nonprocedural Implementation of Search Strategies in Control Network Programming," in *Intern. Symposium on Innovations in Intelligence Systems and Applications (INISTA 2010)*, 2010, pp. 386–390.

[14] E. Golemanova, "Declarative Implementations of Search Strategies for Solving CSPs in Control Network Programming," *WSEAS Trans. Comput.*, vol. 12, no. 4, pp. 176–182, 2013.

[15] "Encyclopedia of Mathematics: Bernoulli trials." [Online]. Available: url: http://www.encyclopediaofmath.org/index.php?title=Bernoulli_trials&oldid=26363.

[16] S. Valverde, "What is the best software for teaching an introduction to genetic algorithms?," 2012. [Online]. Available: https://www.researchgate.net/post/What-is-the-best-software-for-teaching-an-introduction-to-genetic-algorithms.

[17] K. Kratchanov, E. Golemanova, and T. Golemanov, "Control Network Programming Illustrated: Solving Problems with Inherent Graph-Like Representation," in *Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008)*, 2008, pp. 453–459.

[18] K. Kratchanov, B. Yüksel, T. Golemanov, and E. Golemanova, "Control Network Programming Development Environments," *WSEAS Trans. Comput.*, vol. 13, pp. 645–659, 2014.

[19] K. Kratchanov, "Syntax and Semantics for Cinnamon Programming," *Int. J. Comput. Sci. Inf. Technol.*, vol. 9, no. 5, pp. 127–150, 2017.

[20] I. Bratko, *Prolog Programming for Artificial Intelligence*, 4th ed. Pearson Education, 2011.

[21] K. Kratchanov, T. Golemanov, and E. Golemanova, "Control Network Programming: Static Search Control With System Options," in *8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)*, 2009, pp. 423–428.

[22] K. Kratchanov, T. Golemanov, E. Golemanova, and T. Ercan, "Control Network Programming with SPIDER: Dynamic Search Control," in *14th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2010)*, 2010, pp. 253–262.

[23] E. Golemanova, "Study of the Paradigm of Search and Development of a Methodology for Implementation of Search Algorithms in Control Network Programming, Ph. D Thesis," Ruse University, 2014.

[24] K. Kratchanov, E. Golemanova, T. Golemanov, and B. Külahçıoğlu, "Using control network programming in teaching nondeterminism," in *International Conference on Computer Systems and Technologies (CompSysTech'12)*, 2012, pp. 391–398.

[25] K. Kratchanov, E. Golemanova, T. Golemanov, and B. Külahçıoğlu, "USING CONTROL NETWORK PROGRAMMING IN TEACHING RANDOMIZATION," in *International Conference on Electronics, Information and Communication Engineering*, 2012, pp. 67–72.

[26] J. H. Holland, "Adaptation in Natural and Artificial Systems," *Ann Arbor Univ. Michigan Press*, vol. 1, no. 97, p. 5, 1975.

[27] J. Hromkovic, *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer,

2010.
[28] "Introduction to Genetic Algorithms, www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php." [Online]. Available: www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php.
[29] M. Mitchell, *An Introduction to Genetic Algorithms*, Fifth prin. The MIT Press, 1999.
[30] "LEARN GENETIC ALGORITHMS: Genetic Algorithms - Fundamentals." [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals.htm.
[31] S. L. Yadav and A. Sohal, "Comparative Study of Different Selection Techniques in Genetic Algorithm," *Int. J. Eng. Sci. Math.*, vol. 6, no. 3, pp. 174–180, 2017.
[32] D. Liu, "Mathematical modeling analysis of genetic algorithms under schema theorem," *J. Comput. Methods Sci. Eng.*, vol. 19, no. S1, pp. 131–137, 2019.
[33] P. S. Oliveto and C. Witt, "On the runtime analysis of the Simple Genetic Algorithm," *Theor. Comput. Sci.*, vol. 545, no. C, pp. 2–19, 2014.
[34] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.
[35] M. Sipser, *Introduction to the Theory of Computation, Third Edition*. Cengage Learning, 2013.
[36] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimed. Tools Appl.*, vol. 80, pp. 8091–8126, 2021.
[37] "LEARN GENETIC ALGORITHM: Genetic Algorithms - Parent Selection." [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm.
[38] C. Medsker and I. Y. Song, "ProloGA: a Prolog implementation of a genetic algorithm," in *Proceedings IEEE International Conference on Developing and Managing Intelligent System Projects*, 1993, pp. 77–84.
[39] W. Erben and J. Keppler, "A genetic algorithm solving a weekly course-timetabling problem," in *Practice and Theory of Automated Timetabling. PATAT 1995. Lecture Notes in Computer Science, vol 1153*, 1995, pp. 198–211.
[40] K. Kratchanov, E. Golemanova, T. Golemanov, and Y. Gokcen, "Declarative and Procedural Search Strategy Implementations in WinSpider," in *Fundamental Sciences and Applications, Plovdiv, Bulgaria, J. of Technical Univ. at Plovdiv*, 2011, p. v.16, book1, 217-22.

Contribution of individual authors to the creation of a scientific article (ghostwriting policy)

Both authors have contributed equally to the creation of this article.

Sources of funding for research presented in a scientific article or scientific article itself

Creative Commons Attribution License 4.0 (Attribution 4.0 International , CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US