Spatial Genetic Programming

Iliya Miralavy[1,2(✉)] and Wolfgang Banzhaf[1,2,3]

[1] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA
miralavy@msu.edu
[2] BEACON Center of Evolution in Action, Michigan State University, East Lansing, MI, USA
[3] Ecology, Evolution and Behavior Program, Michigan State University, East Lansing, MI, USA

Abstract. An essential characteristic of brains in intelligent organisms is their spatial organization, in which different parts of the brain are responsible for solving different classes of problems. Inspired by this concept, we introduce Spatial Genetic Programming (SGP) - a new GP paradigm in which Linear Genetic Programming (LGP) programs, represented as graph nodes, are spread in a 2D space. Each individual model is represented as a graph and the execution order of these programs is determined by the network of interactions between them. SGP considers space as a first-order effect to optimize which aids with determining the suitable order of execution of LGP programs to solve given problems and causes spatial dynamics to appear in the system. RetCons are internal SGP operators which enhance the evolution of conditional pathways in SGP model structures. To demonstrate the effectiveness of SGP, we have compared its performance and internal dynamics with LGP and TreeGP for a diverse range of problems, most of which require decision making. Our results indicate that SGP, due to its unique spatial organization, outperforms the other methods and solves a wide range of problems. We also carry out an analysis of the spatial properties of SGP individuals.

Keywords: Genetic Programming · Spatial Computing · Evolutionary Computation

1 Introduction

Even though evolutionary algorithms have proven to be applicable for solving a wide range of computationally represented problems, they often are abstractions of their natural counterparts and do not account for the impactful dimensions of *time* and *space* in nature. Spatial Computing [7] is a relatively new field in computer science that states the distribution of computational elements in *space* can enhance the performance and the feasibility of computation. It further argues that it is more important to include *space* in our computational models as our understanding of natural computing systems and coupling of computational

G. Pappa et al. (Eds.): EuroGP 2023, LNCS 13986, pp. 260–275, 2023.
https://doi.org/10.1007/978-3-031-29573-7_17

models and physical elements increases. Additionally, Spatial Computing offers a more natural approach to parallel computation. Parallelism is an essential part of natural systems where elements, be it particles in physics, molecules in chemistry, or individual agents in biology, are all bound by *space* and perform their functions in time simultaneously. The spatial properties of these elements play a critical role in the performance of these systems.

The primary contribution of this paper is introducing Spatial Genetic Programming (SGP), a Genetic Programming (GP) system controlled by a 2D space that evolves Linear Genetic Programming (LGP) [4] programs. An SGP model consists of one or more LGP programs which are represented as graph nodes located in a 2D space. In SGP, *space* plays an integral role in determining the order of execution of LGP programs. In each individual, these nodes form a network of interactions, responsible for regulating the order of execution of the LGP programs based on their spatial properties and the internal dynamics of the system. If necessary, the flexible representation of SGP allows for controlling the evolution of iterative behavior to develop more compact models. To show the effectiveness of the proposed system, we utilize SGP to solve different classes of problems and compare it to two common GP paradigms.

2 Related Literature

The evolution of the SGP models consists of two main parts: evolving the structural properties of the system (i.e., the graphs representing the network of interactions between LGP program nodes) and the instructions of the LGP programs.

Various works in the literature focus on evolving graphs capable of representing solutions to computational problems. Tree GP (TGP) [15] uses graph representation of tree data structures. As the most common type of GP, TGP has been previously used for various types of applications such as transportation [24], symbolic regression [1,3], image processing [22], classification [2] and others. Although the tree data structure is simple for understanding and evolving solutions, traversing these structures is not a computationally trivial task and often causes bloat problems.

Cartesian Genetic Programming (CGP) [16] is another mainstream graph-evolving GP that has shown good performance for solving computational problems. CGP uses integer values as genes representing nodes in a graph, their functions, links between the nodes, and how inputs and outputs are connected to these nodes. Compared to TGP, CGP is computationally less expensive, and therefore its evaluation time is faster and is less prone to bloat [17]. An interesting feature of CGP is its ability to encode and control computational systems similar to Artificial Neural Networks [14,23]. CGP also has various applications in agent control [10], image processing [9] and circuit design [12]. Similar to CGP, in SGP, the computational cost of creating network graphs are reduced by a mechanism that controls the system with a 2D grid.

It is possible to evolve GP models that do not rely directly on graph representations. LGP is among these types of GP. This paradigm is represented

as a series of instructions, usually in the form of imperative programming language or machine language that execute sequentially. LGP supports branching operators, which allow the execution pointer to jump between instructions. One particular weakness of LGP is correctly determining the number of internal registers, which, if chosen wrong, will drastically undermine the performance of the solutions [19]. On the other hand, LGP programs are quite fast because they can be designed to run on the processor directly. This strength was the reason for choosing LGP programs to be a part of the SGP system. Another common GP variant that does not use graphs as their representation is Stack-based Genetic Programming. In such GP, fundamentally similar stack-based programming languages are responsible for obtaining operands for the program operators from a data stack and pushing the results of the operations to these stacks. Depending on the designed rules, multiple data stacks for different data types might exist. Generally, Stack-based Genetic Programming models are faster than tree structures, and it is possible to create bloat-free mutations and crossover mechanisms. Push GP [21] is one of the most famous stack-based systems and has been previously used for various applications such as automatic code simplification [11] and Python code synthesis [20].

Tangled Programming Graphs (TPG)s [13] are among the systems that evolve both computer programs and the relationship between them in the form of a graph. This system has been previously used for solving Visual Reinforcement Learning problems such as Atari games and has produced comparable results to deep learning algorithms. TPGs are one of the closest works in the literature to the idea of SGP since it is constructed based on mechanisms that control the execution flow of programs until a terminal state is reached; however, there are some key differences between the two systems. In SGP, the execution order is determined by minimizing a traverse cost value between the source program and every other program in the system. Furthermore, SGP supports iterative behaviors by allowing programs to execute more than once. In contrast, TPGs use a bidding system among teams of programs to determine the pathways taken to execute programs. SGP is controlled by a 2D space which makes the spatial properties of the nodes important for selecting the following programs to execute. Finally, unlike TPGs, and much like more conventional GP systems, in SGP a population of mutually exclusive individuals is used. In the next section, we'll be exploring the implementation of the SGP system in more detail.

3 Spatial Genetic Programming

SGP models are program nodes spread in a 2D coordinate system. The aim of the SGP interpreter is to choose the order of program executions until a termination condition is met while minimizing the traversing cost between programs. A cost function is used to calculate the cost of trajecting from the source coordinate (starting from $(0, 0)$ as null program) to every other program node. In other words, in each step, a weighted network of interactions between all the program nodes is made in which the weights are the cost of traversing from a source

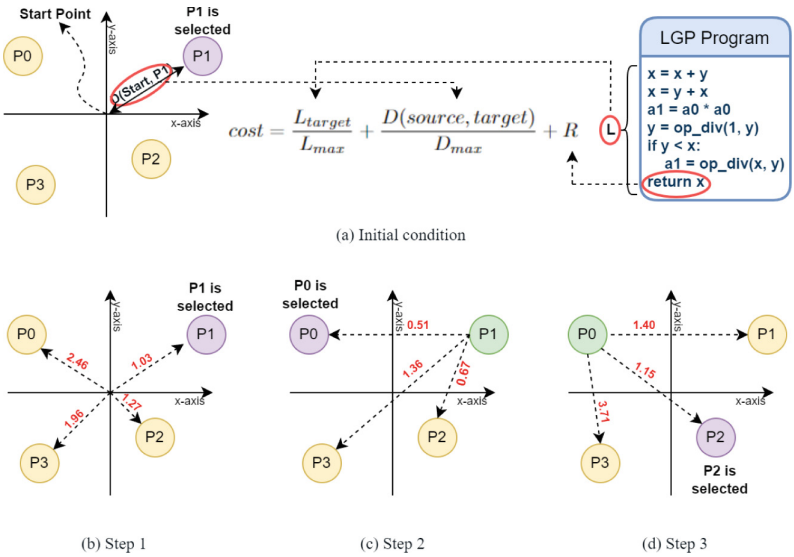


Fig. 1. Model representation and interpretation steps for an SGP model with 4 programs. a) Different contributions to the cost function. b) Step 1: P1 is selected since it has the lowest traverse cost from the starting point. Red values indicate cost c) Step 2: P1 is the source point and P0 with the lowest traverse cost is selected. d) Step 3: P0 is the starting point and P2 is selected.

coordinate to a destination program node. These weights alter as the source coordinate and the internal state of the system change.

The program with the lowest traverse cost is then selected to execute prior to the others. If termination conditions are not met, the same process repeats. The position of the most recently executed program is then set to be the source coordinate to determine the next program to be executed. In Fig. 1a, an overview of an SGP model is illustrated in its initial conditions. Each node represents a program and is labeled with the program name. Each program contains instructions that manipulate internal memory registers shared between all programs and outputs a single value corresponding to an internal register, an input, or a constant value. In step 1 (Fig. 1b), the cost of traversing from $(0, 0)$ to every other node is calculated (details of which can be found in the next section). Since P1 has the lowest cost, it is selected for execution. In the next step (Fig. 1c), P1 is the source point for calculating the costs to every other node, and therefore P0 is chosen for execution. The same principle continues until a termination condition is reached. Algorithm 1 (see supplementary material [18]) shares the details of how SGP selects the next program in line for execution. All of the individual programs are stored in a list. A loop on the program list is performed to calculate the cost of traveling to each program. Safeguards for protecting against infinite cost values and revisiting a node in case of loop-free configuration are in place

to prevent invalid selections. The program with the lowest traverse cost is stored in the *next_program* variable and is the final program's output.

3.1 The Cost Function

The cost function considers the spatial and internal states of the system to calculate the cost of traversing to a given program node based on a source coordinate. SGP operates in two main modes: *Spatial* and *Programmatical*, which will be described in turn.

In *Spatial* mode, the distance between the source coordinate and the target node and the target program's length are calculated and normalized. *Program length* is defined as the number of instructions in that program, meaning that programs with a higher number of instructions have a slightly lower chance of being selected. The cost in this mode is calculated using the following equation:

$$cost = \frac{L_{target}}{L_{max}} + \frac{distance(source, target)}{D_{max}}$$

In which L denotes program length, $D()$ is an internal function that returns the Euclidean distance between two coordinates, and D_{max} denotes the maximum distance between two nodes of SGP. For an evolved SGP model in *Spatial* mode, normalized length and distance between every two nodes are constant, meaning that the order of execution of the programs does not change, forming a static solution graph that is not affected by input values.

In the *Programmatical* mode, other than the metrics considered in the *Spatial* mode, the output variable of the target program is also taken into account. Therefore, calculating the cost in this mode follows the following equation:

$$cost = \frac{L_{target}}{L_{max}} + \frac{D(source, target)}{D_{max}} + R$$

In which R denotes the current value of the parameter set to be the output of the target LGP program. Each LGP program terminates with a return statement that outputs a numerical value for R. The rest of the variables are the same as the ones in the *Spatial* mode. The current value of R cannot be pre-computed and at every step highly depends on the previously executed programs and how the internal registers have been manipulated prior to the cost calculation step. The impact of program length and the distance between nodes are normalized; however, the value of R is not bounded to any range and depends on the problem inputs. This design decision might increase the impact of R on selecting the next program significantly; however, the cost function is configurable and can be modified to normalize the scale of R. It is prevalent for models evolved in this mode to take different execution routes with different sets of given inputs, forming dynamic solution graphs. This feature enables the opportunity to evolve localization in the system so that different sections of an SGP model respond to different sets of stimuli, a known characteristic of the brain in natural organisms.

3.2 Outputs, Termination Conditions and Model Execution

An imperative SGP system has four means of producing outputs. First is the numeric value returned by the last executed SGP program. Second, SGP also outputs the system's internal state, which is all the register values (initially set to 0) manipulated during the run-time of a model. Third, SGP operators are allowed to manipulate an external file, a computational object, or a third-party environment. Finally, it is possible to associate terminal programs with discrete actions or outputs. In other words, if a terminal program is reached, the action associated with that program is performed in the problem environment ending the individual execution. Depending on the model inputs, a different final program might get selected and thus produce a different action.

Multiple conditions can end the execution of a model. Each model program has a chance to become a terminal node and end the execution. Suppose a model does not have any terminal program. In that case, a limit equal to the total number of programs in that model is set, breaking the execution if the count of executed programs exceeds that limit. Execution also ends if there are no more candidate programs or if the execution time exceeds a time threshold. Algorithm 2 (see supplementary material [18]) shows the details of executing an individual model.

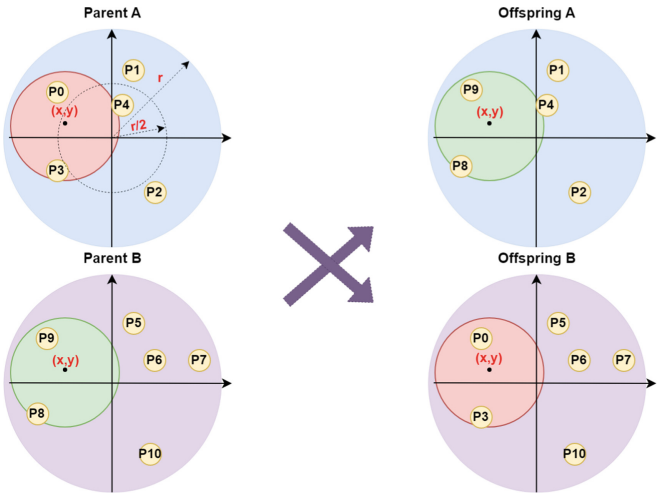


Fig. 2. Crossover between two SGP models. Suppose (x, y) is the randomly chosen point within $\frac{r}{2}$ distance of the center (0, 0). Programs within $\frac{r}{2}$ distance of (x, y) form S_i of parent A (red circle) and parent B (green circle), and the rest of the programs form S_o (blue circle for parent A and purple circle for parent B). Offspring A is a combination of the programs in S_i of parent A and S_o of parent B and Offspring B is a combination of the programs in S_i of parent B and S_o of parent A.

3.3 Evolution of Models and the Genetic Operators

Initially, a population of random SGP individual models is generated, in which the number of programs in each model, their length, and the initial coordination of the nodes within an allowed 2D space are randomly chosen. Next, an object pool of operators and operands is created from which the operator and operand(s) of each statement or instruction are randomly selected. If there is no suitable operand for an operator, it will be removed from the selection pool. Operator and operand objects are reusable and therefore do not add to the computational cost of the system.

After evaluating models in each generation of the evolution, tournament selection is applied to the population. The two best competitor models are selected and have a chance to crossover to produce two offsprings or directly make it to the next generation of models after mutation. If the crossover happens, a mutation with a chance is also applied to the two new offsprings.

The 2D space of the SGP models is bounded by a radius parameter r, meaning that the program coordinates must be within r distance from $(0, 0)$. A random coordinate point within $\frac{r}{2}$ distance from $(0, 0)$ is selected to be utilized while performing crossover between two individual models. Let us denote the set of programs within $\frac{r}{2}$ distance from the randomly chosen point of an individual with S_i and the set of programs outside that radius with S_o. Then, in the crossover between parent A and B, every program in S_i of parent A and S_o of parent B form one offspring while the rest of the programs form the other offspring (Fig. 2). There is no limit to the number of programs that are impacted by the crossover operator.

After crossover, there is a chance for every program of each individual to undergo mutation. SGP mutations can happen on a structural level, i.e., altering a program location or switching a program type (input program to output or vice versa), or on a statement level, i.e., altering the LGP programs. There are three types of structural mutations. 1) A program's coordination can change by performing a random walk with a fixed random step size. 2) The program type can alter from input to output or vice versa. 3) A program can be added or removed from/to the system. These modifications, along with an LGP mutation that targets the return value of the programs, are responsible for changing the behavior of how the programs will be selected for execution. There are three types of LGP mutations, which add statements to the program, delete a statement from the program or modify an existing statement if possible. By default, these mutations have an equal chance of occurring.

3.4 Conditional Return Statements

One of the abilities of SGP is to evolve rational pathways that change in response to the problem inputs. Conditional operators such as the basic *if* statements can help build a logic behind the return values of each program, forcing a different order of execution when different input values are given to the system. By default, however, SGP requires each program to have a final single return statement that

cannot be connected to any other operators, such as being tied to a conditional *if* statement. Since allowing evolution to use a combination of conditional operators and internal state values to evolve conditional pathways is not trivial, we come up with the idea of replacing the normal return statements of the program with a custom conditional operator called RetCon (stands for Return Conditions). This operator forces a condition on the return statement in a way that if the condition is true, an internal state value or a constant value will be returned. Otherwise, another return value will be selected. The two return values could be the same.

4 Experiments and Results

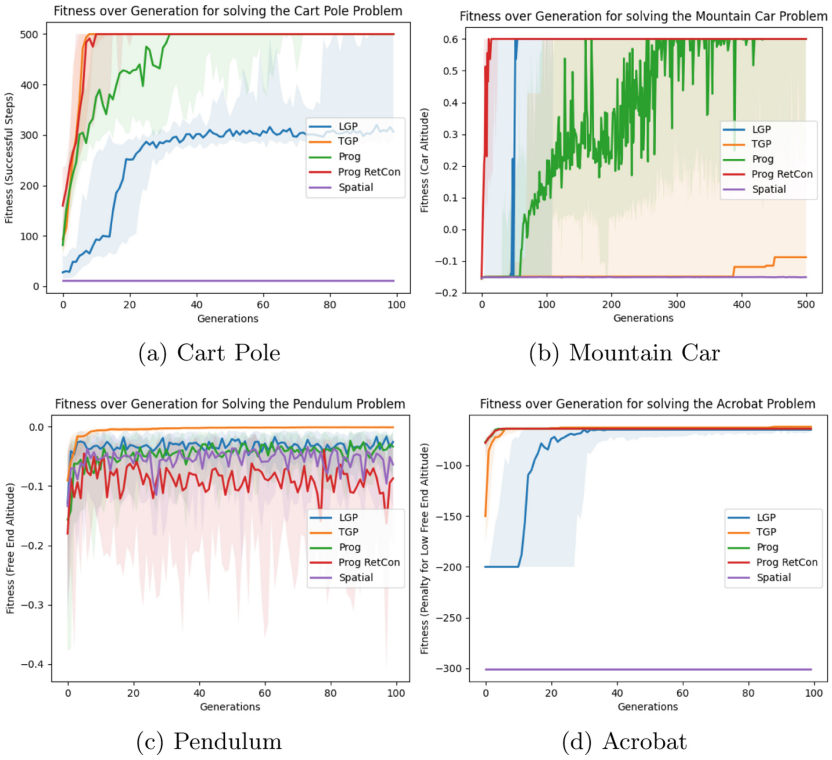
In this section, we apply SGP to a set of problems classified into two case studies to analyze the behavior of the system by comparing different modes of otherwise identical SGP setups with classical TGP and LGP. The TGP included in the DEAP framework [8] and the same LGP system used for the SGP programs were used to conduct the experiments. The use of RetCon in SGP facilitates the evolution of conditional pathways, making SGP models well-suited for addressing problems that require decision-making. The specific problem set for each case study was selected due to the presence of a decision-making component.

4.1 Case Study: Classic Control Problems

OpenAI Gym [5] is a library of Reinforcement Learning problems in Python which helps with the development and comparison of problem-solving algorithms by providing a straightforward environment-to-algorithm API. In particular, we tackled Cart Pole, Mountain Car, Pendulum and the Acrobat problems from the Gym library; details of which can be found in [5].

Figure 3 shows the results for tackling the OpenAI Gym classic control problems. 50 replicate experiments with different random seed values were conducted for each of the four problems. The median Fitness values over generations for the best-evolved models of each replicate are illustrated. The shaded areas represent the 25 and the 75 quantiles, while the solid lines represent the median. Three configurations of SGP are tested against these problems and are compared with classical TGP and LGP. *Prog* refers to the Programmatical mode of the system; *Prog RetCon* indicates the usage of conditional return statements in the Programmatical mode, and *Spatial* refers to the Spatial mode. In the spatial mode, the usage of RetCon operators does not make a difference since the return statements of the programs do not change the execution order. All of the experiments are run for 1000 generations; however, depending on the problem, after a certain number of generations, the fitness values cease to change, and therefore, a portion of the generations are selected to ease the analysis of the results. Finally, even though all the classic control problems are deterministic, the starting conditions are slightly randomized (e.g., the position of the car in

the mountain car problem) to help the problem solvers find a generalized solution. For all of the experiments, *if*, *assign*, and basic math operators are used as the function/operator set of LGP and SGP.



(a) Cart Pole (b) Mountain Car

(c) Pendulum (d) Acrobat

Fig. 3. The fitness over generations plot for solving the four classic control problems. a) Fitness equals to the number of steps in which the pole is held in an upright position. b) Fitness represents the car altitude at the end of each evaluation. c) Fitness equals to the altitude of the free end of the Pendulum at the end of each evaluation. d) Fitness indicates a -1 penalty for each step in which the free end has not passed the threshold line.

To solve the Cart Pole problem, SGP is configured to use discrete outputs in which each individual must consist of two terminal nodes, each associated with an action of either accelerating the cart towards left or right. 3a shows the results for solving the Cart Pole problem. SGP with RetCon and TGP solve this problem in less than 20 generations. Programmatical settings without RetCon also solve the problem. However, it takes more generations to solve, and the shaded green area shows that it takes more time for all the individuals in all the replicates to be able to solve this problem while all the individuals of the replicates for the RetCon settings solve the problem in less than 30 generations.

LGP also solves the problem but its performance is not as good as the rest of the approaches The Spatial setting fails to solve the task over all generations since, in this setting, the network inputs do not change the execution order of the graph. In other words, the same discrete action is always taken; therefore, constant fitness is achieved over generations.

Same as the Cart Pole problem, for the Mountain Car problem, SGP is configured to use discrete outputs. As illustrated in Fig. 3b the RetCon outperforms the other two configurations by solving the problem for all the replicates in less than 50 generations. LGP performs slightly worse, solving the problem in approximately 60 generations. The Programmatical setting without RetCon has a cold start, but the replicates mostly solve the problem at around 450 generations. However, the difference between the fitness of the best models among all the replicates varies greatly. The shaded orange area shows that there are individuals in the TGP approach that solve the problem but the median results are worse than the other approaches in 500 generations. Once again, the spatial mode fails to solve the problem while producing a constant fitness.

The nature of the Pendulum problem is slightly different from the other problems since it requires a continuous input indicating the amount of torque applied. Unlike the other three problems, the Spatial configuration performs comparably to the other approaches. TGP outperforms all other approaches; however as shown in Fig. 3c fitness values of approximately 10^{-4} were achieved by the best individuals of all the approaches showing almost an upright position of the pendulum. The high fluctuation of the median line is due to the high impact of the random starting position of the pendulum on the outcome of the evaluation.

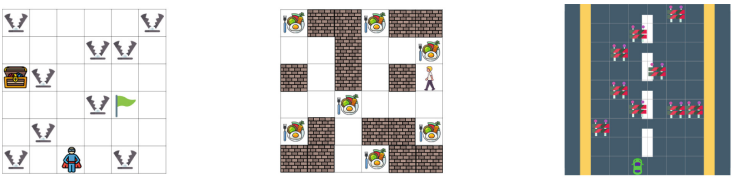
The final classic control problem tackled in this paper is the Acrobat problem. As illustrated in Fig. 3d, SGP manages to solve the problem in both Programmatical modes with or without RetCon in less than 5 generations and improves its performance until 10 generations managing to reach the specified line in all of the best models in about 60 steps. Like the other discrete output problems, the Spatial mode drastically fails by only producing a constant output. TGP has a slightly worse performance while LGP solves the problem in 40 generations.

4.2 Case Study: Custom Toy Problems

The custom Toy Problems is a custom library of three Reinforcement Learning problems included with the SGP source code that can be briefly described as the following (Fig. 4):

- **The Adventure Problem:** This problem is inspired by an Atari 2600 game called *Adventure* [6]
- **The Foraging Problem:** This is a famous classic Artificial Life problem in which an agent has to gather all the food spread in a 2D grid. The tiles are often blocked by obstacles or walls (Fig. 4b).

– **The Obstacle Avoidance Problem:** As illustrated in Fig. 4c a car agent is driving on a road that is occasionally blocked by randomly appearing roadblocks. The car agent has to avoid hitting the roadblocks for a specified number of time steps.



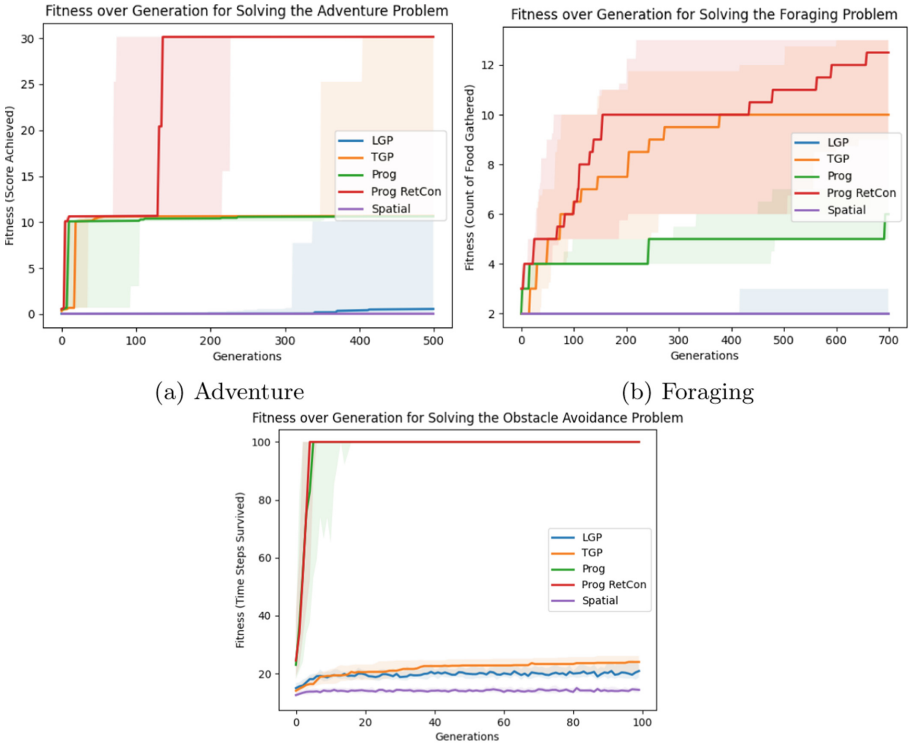
(a) Adventure (b) Foraging (c) Obstacle Avoidance

Fig. 4. Three different toy problems. *Icons used in the images are from:* https://www.flaticon.com/

Figure 5 depicts the results produced for solving the three Toy Problems for 50 replicates. The only non-deterministic problem is Obstacle Avoidance since the roadblocks spawn randomly.

In the adventure problem, the observation consists of 6 integer inputs corresponding to the agent's vision cone and a single bit corresponding to whether the agent has picked the treasure or not. The agent's vision cone shows two three-tile rows in front of the agent. The problem's action space consists of three discrete actions: moving one tile ahead, turning left, and turning right. All the entities in the problem grid and the empty tiles are coded with unique integer values and are visible to the agent. The agent can move to the treasure tile to automatically pick up the treasure. A small reward of 0.01 is given to the agents that move. The computational models are responsible for giving an agent instructions to solve the task through actions, and the individual's fitness equals the score the controlled agent achieves. A significant score of 10 is given to the agents that manage to grab the treasure, and a very significant score of 20 is given to the agents that reach the final destination while carrying the treasure. The simulation ends after 100 time steps or when the agent reaches the final destination or falls into a trap. The score is returned to the SGP evolver module as the controlling model's fitness value. Figure 5a depicts the results produced for solving the Adventure problem over 500 generations. As expected, the Spatial configuration fails to solve a task with discrete output. The programmatical settings manage to evolve agents capable of picking up the treasure; however, they fail to reach the final destination. The RetCon settings, however, solve the problem entirely in less than 250 generations. TGP slightly outperforms the Prog settings since in later generations, the best TGP individuals fully solve the problem. LGP on the other hand, only manages to find the treasure in the later generations but fails to completely solve the problem and the median line always stays low.

The Foraging problem has an observation space consisting of 6 inputs corresponding to the agent's vision cone. Like the adventure problem, the vision cone includes the two three-tile rows in front of the agent. The action space of the



(a) Adventure (b) Foraging

(c) Obstacle Avoidance

Fig. 5. The fitness over generations plot for solving the three toy problems. a) Fitness indicates the score of the agent at the end of evaluation. b) Fitness is equal to the number of food gathered by the agent c) Fitness is the total number of time steps that the agent has survived the environment

problem is the same as the Adventure problem consisting of three actions: moving, turning left, and right. A total of 20 food tiles are available for the agents to take, which are often placed at the end of a maze-like pattern in which the agent will have to return to the path taken to reach the food to get out (e.g., top left food in Fig. 4b). The simulation is run for 200 time steps while no reward is considered for moving. Compared to other problems, this is a more challenging task to solve since the maze-like patterns make it quite difficult to gather all the food in the allowed time steps. Figure 5b shows the result for solving this problem. Same as most cases, the Programmatical settings with RetCon outperforms the other two modes while being able to gather as much as 14 food at best among all the replicates. The changes in the median line seem to show evolution after 700 generations. Perhaps, running this task for a more extended period would help the system completely solve the problem. The programmatical SGP needs quite more time to evolve conditional logic to solve these types of problems only using

a basic *if* statement, and the spatial mode fails to solve the task. The performance of TGP on this scenario is slightly worse than the RetCon settings while LGP only manages to perform better than the spatial mode after approximately 400 generations.

The observation space in the obstacle avoidance problem consists of 12 integer values corresponding to the vision cone of the car agent. This vision cone includes four three-tile rows in front of the agent. The action space of the problem is three discrete actions: moving left and right and doing nothing. To achieve a perfect score, the car agent must avoid all the roadblocks for 100 time steps. The number of time steps before the car agent hits a roadblock is the fitness of the controlling SGP model. Figure 5c shows the results produced for solving this task. Both SGP Programmatical configurations with or without RetCon manage to avoid all the obstacles in less than 20 generations. At the same time, it takes a bit longer for all the replicates to completely solve the problem for the setting without RetCon. The fluctuations in the case of Spatial mode are due to the randomness of the roadblock patterns selected to appear in the far front of the car. TGP and LGP do not achieve good fitness levels on this problem but outperform the spatial mode.

4.3 Impact of a Spatial Crossover on the Evolution of Programs

To check whether the mechanisms in the system impact the spatial properties of the SGP models, an experiment was conducted with a different crossover algorithm called the Spatial Crossover. This crossover is quite similar to the normal crossover used in SGP however, instead of choosing a random circular area to form S_i, programs that are located in the top right quadrant of the 2D space (x-coord and y-coord greater than or equal to 0) are selected to form S_i of the parent individuals. In this approach, always the same spatial portion of the individuals swap to form offspring. We tracked the position of all the individuals' programs (not just the best) in all the 50 replicates. We summarized the results in Table 1. SC stands for Spatial Crossover, NC stands for Normal Crossover and P1 and P2 refer to two test arbitrary problems. The 2D space of each individual is divided into four quadrants starting from the top right (Q1) and going clockwise to the top left (Q4). Results show that in the case of using the Spatial Crossover, programs tend to move out of the Q1 area in which the crossover is happening. This behavior is reflected by the significantly lower percentage of appearance of programs in Q1 compared to the case where the normal crossover is being applied.

Table 1. Position of the final programs of all individuals in the latest generation using Normal Crossover (NC) and Spatial Crossover (SC) for two test problems.

Quadrant	RetCon P1 (NC)	RetCon P1 (SC)	RetCon P2 (NC)	RetCon P2 (SC)	Spatial P1 (NC)	Spatial P1 (SC)	Spatial P2 (NC)	Spatial P2 (SC)
Q1	25.69%	**12.63%**	26.23%	**10.98%**	27.51%	**8.98%**	34.17%	**6.8%**
Q2	22.09%	30.21%	22.84%	27.07%	21.68%	29.78%	26.48%	28.58%
Q3	27.87%	27.72%	23.79%	29.32%	21.66%	30.38%	17.68%	33.52%
Q4	24.34%	29.64%	27.14%	32.63%	29.14%	30.87%	21.67%	31.1%
Total	38310	40030	27.14%	38493	38566	41742	38059	41597

5 Conclusion

This paper introduced a new GP paradigm which accounts for the dimension of space as a first-order effect to optimize. SGP works in two modes of Spatial and Programmatical, which bring unique characteristics to the system, allowing it to evolve static and dynamic graphs, respectively. The impact of these two operation modes was tested against two classes of problems while introducing conditional return statements. SGP was tested against four classic Control Problems of OpenAI Gym library. RetCon's ability to quickly evolve conditional statements to choose the right pathway of the graph by manipulating the weights of the underlying regulatory network was shown during these experiments. For all the cases except the Pendulum problem, RetCon quickly solved the control tasks. The Pendulum problem required less decision-making and more accuracy on the produced continuous outputs (amount of torque). The Programmatical mode without RetCon was able to solve the control problems as well. However, it takes more time for evolution to evolve the factual conditional statements in the LGP programs to reflect the same decision-making structures. The Spatial mode fails in producing discrete outputs while showing promise in the Pendulum problem that requires continuous outputs. This is because of the ability of the Spatial mode to refrain from using too many conditional statements and rely more on the power of LGP to produce continuous outputs. SGP was compared to two other approaches of TGP and LGP. Except for the Pendulum task which had more of a continuous nature, SGP outperformed the other two approaches in all cases.

Three custom Toy Problems were introduced in this paper, on which SGP was tested. These problems had a larger observation space compared to the classic control problems. Comparing the three tested configurations, SGP produced a similar result to the Control Problems, with RetCon outperforming the two other configurations. The more complex observation space did not significantly impact the performance of SGP showing better performance than TGP and LGP. The Foraging problem was not completely solved; however, improvements in the fitness values showed the possibility of solving this problem if run for an extended period.

A shortcoming of SGP is not having enough control to create a balance in evolving structural elements and LGP programs simultaneously. Perhaps, the utilization of parallel island models that decouple focusing on the evolution of

the structural elements and the SGP programs from the main population while interacting with it now and then could be helpful in this scenario to achieve better results. Furthermore, a method for optimizing the system hyper-parameters during evolution could also be among the future directions of this work. As of now, the cost function used in the system adds the normalized distance and length to the return value of the SGP programs. This reduces the impact of distance and length compared to the possible return values. Perhaps a different cost function can result in more exciting results. Finally, to show the effectiveness of SGP, it is necessary to apply it to more realistic and complex problems and to compare it with state of the art problem solvers. Currently, we are working on including more evaluations to the proposed work to prepare it for a future journal submission. We aim to perform systematic analysis on the spatial properties of SGP and how the spatial properties are responsible for achieving the system's performance level and to compare it with TPGs and CGPs.

Code Availability and Supplemental Materials

SGP code in Python, description of main SGP algorithms, practical examples and tutorials for applying SGP and replicating the results of this article can be found in:

https://github.com/elemenohpi/EuroGP-SGP

References

1. Amir Haeri, M., Ebadzadeh, M.M., Folino, G.: Statistical genetic programming for symbolic regression. Appl. Soft Comput. **60**, 447–469 (2017)
2. Aoki, S., Nagao, T.: Automatic construction of tree-structural image transformations using genetic programming. In: Proceedings of the 10th International Conference on Image Analysis and Processing, pp. 136–141. IEEE (1999)
3. Augusto, D.A., Barbosa, H.J.: Symbolic regression via genetic programming. In: Proceedings, vol. 1. Sixth Brazilian Symposium on Neural Networks, pp. 173–178. IEEE (2000)
4. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007). https://doi.org/10.1007/978-0-387-31030-5
5. Brockman, G., et al.: OpenAI Gym (2016)
6. Chance, G.: Adventure - atari - atari 2600. https://atariage.com/manual_html_page.php?SoftwareLabelID=1. Accessed 07 Aug 2022
7. DeHon, A., Giavitto, J.L., Gruau, F.: 06361 Executive report - Computing media languages for space-oriented computation. In: Computing Media and Languages for Space-Oriented Computation. Dagstuhl Seminar Proceedings (DagSemProc), vol. 6361, pp. 1–5. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2007)
8. Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: DEAP: evolutionary algorithms made easy. J. Mach. Learn. Res. **13**(1), 2171–2175 (2012)

9. Harding, S., Leitner, J., Schmidhuber, J.: Cartesian genetic programming for image processing. In: Riolo, R., Vladislavleva, E., Ritchie, M., Moore, J. (eds.) Genetic Programming Theory and Practice X, pp. 31–44. Genetic and Evolutionary Computation. Springer, New York (2013). https://doi.org/10.1007/978-1-4614-6846-2_3
10. Harding, S., Miller, J.F.: Evolution of robot controller using cartesian genetic programming. In: Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 62–73. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31989-4_6
11. Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 937–944 (2017)
12. Hodan, D., Mrazek, V., Vasicek, Z.: Semantically-oriented mutation operator in cartesian genetic programming for evolutionary circuit design. Genet. Program. Evolvable Mach. **22**(4), 539–572 (2021). https://doi.org/10.1007/s10710-021-09416-6
13. Kelly, S., Heywood, M.I.: Emergent tangled graph representations for Atari game playing agents. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 64–79. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_5
14. Khan, M.M., Ahmad, A.M., Khan, G.M., Miller, J.F.: Fast learning neural networks using cartesian genetic programming. Neurocomputing **121**, 274–289 (2013)
15. Koza, J.R.: Genetic Programming: On the Programming of Computer by Means of Natural Selection. MIT Press, Cambridge (1992)
16. Miller, J.F.: An empirical study of the efficiency of learning Boolean functions using a cartesian genetic programming approach. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1135–1142 (1999)
17. Miller, J.F.: Cartesian genetic programming: its status and future. Genet. Program. Evolvable Mach. **21**(1), 129–168 (2020)
18. Miralavy, I., Banzhaf, W.: SGP supplementary materials and code (2023). https://github.com/elemenohpi/EuroGP-SGP
19. Oltean, M., Grosan, C.: A comparison of several linear genetic programming techniques. Complex Syst. **14**(4), 285–314 (2003)
20. Pantridge, E., Spector, L.: Code building genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2020), pp. 994–1002 (2020)
21. Spector, L.: Autoconstructive evolution: push, pushgp, and pushpop. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), vol. 137 (2001)
22. Tran, B., Zhang, M., Xue, B.: Multiple feature construction in classification on high-dimensional data using GP. In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1–8. IEEE (2016)
23. Turner, A.J., Miller, J.F.: Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, pp. 1005–1012 (2013)
24. Yao, M.J., Hsu, H.W.: A new spanning tree-based genetic algorithm for the design of multi-stage supply chain networks with nonlinear transportation costs. Optim. Eng. **10**(2), 219–237 (2009)