Large language models help computer programs to evolve

Jean-Baptiste Mouret

Large language models help programs to evolve

Jean-Baptiste Mouret
Inria, CNRS, Université de Lorraine, France
jean-baptiste.mouret@inria.fr

Author-generated version / preprint.

Full reference: Mouret, J.-B. Large language models help computer programs to evolve. *Nature* **625,** 452–453 (2024)
https://www.nature.com/articles/d41586-023-03998-0

Although machines outperform humans at many tasks, from manufacturing to playing games[1,2], they are commonly considered incapable of creating or inventing. But this is changing: over the past three years, technology companies have been releasing artificial intelligence (AI) programs that can draw or write with impressive creativity. Scientific discovery might be the next target, as computers start to uncover knowledge that has eluded scientists. In a recent article, Romera-Paredes et al.[3] report an autonomous mathematical discovery for which AI was used – not to check a proof or to execute tedious computations – but instead to solve open problems. This proof of concept is likely to be followed by others like it, as software becomes a creative contributor to scientific discoveries.

Romera-Paredes and colleagues' work is the latest step in a long line of research that attempts to create programs automatically by taking inspiration from biological evolution, a field called genetic programming[4]. The process starts by taking many random programs and running each of them to find out how well they solve a target problem. The best programs are selected, copied and then randomly modified, in a manner that is akin to genetic variation. The process then begins again with these modified programs, which are selected and modified until the program solves the problem adequately.

The key question in genetic programming is how to represent programs so that they can be easily but meaningfully be modified by random variation. In other words, what is the 'DNA' of a computer program? For instance, adding random letters to a program written in the Python scripting language is unlikely to result in a program that follows Python syntax, which means that the vast majority of modified programs cannot be executed by the computer, and are therefore useless.

To approach this problem, genetic programming researchers have taken inspiration from compilers, which are the programs that transform the text written in a programming language into code that a computer can interpret. Compilers represent programs in the computer's memory with an abstract 'tree' structure. Using this representation, a genetic programming system 'mutates' a program by randomly changing one node in the tree to a different value (Fig. 1a). For example, a mutation might replace a plus sign with a minus sign or the number 2 with a 3. Similarly, the system mixes two programs by exchanging randomly chosen sub-trees from two separate programs.

This approach has given rise to many fascinating results. For the past 20 years, a competition has been held to showcase how evolution-inspired algorithms measure up to humans (see human-competitive.org/awards). One example is a program called Eureqa[5], which automatically finds the equations of motion for classic dynamical systems, such as a swinging pendulum, using data alone. Another example is the GenProg system[6], which provides a way of automatically repairing existing open-source programs – and even earned money doing so.

Nevertheless, genetic programming has so far been successful only for equations or short sections of programs, typically comprising a dozen lines. The reason is that programming is hard. It requires the right syntax. It requires intuition about what could work. And every change requires that the programmer take into account the context of the rest of the program. In general, random variations in syntax trees are rarely meaningful and the genetic-programming process yields results only through millions of repetitions.

This situation is changing with large language models (LLMs). These large neural networks are trained to predict the next words given a context (known as a prompt), after having been fed millions of lines of code. This large body of examples gives the model an 'intuition' about what a typical programmer would write in the same situation. LLMs that are specialized in code generation are used daily by many programmers in the same way that others use the autocomplete function available in many apps – the model guesses the most likely text to follow each piece of code, so there is no need to type it or to memorize the function names.

LLMs are game-changing tools for programmers, but they might also be the missing piece of genetic programming: they embed the knowledge of thousands of programmers to generate meaningful code for a given context. Instead of replacing random parts of a syntax tree, an LLM can generate a variation of a program written in a standard programming language, such as Python. To do so, a simple, but powerful, approach is to select two programs, concatenate them, and ask the LLM to complete the program – resulting in the generation of a third program (Fig. 1b). The result will probably have valid syntax and be meaningful in its context. However, it might not be exact nor optimal. This is why the process must be iterated, by selecting the highest-performing programs, generating variations using the LLM, and testing these variations.

Romera-Paredes et al. used this fresh approach to genetic programming to find ways of solving mathematical problems in optimization and geometry better than the best attempts of human programmers. The authors' system shows promise, but it still requires guidance in the form of an 'evaluate function', which nudges the model in the most productive direction. A more direct way would be to use a 'validate function', which determines when the problem has been solved. The authors' evaluate function is similar to a school test that has been designed to reward learning by striking a balance between being easy and difficult.

Using automatic programmers, such as the one developed by Romera-Paredes and colleagues, will require the same level of careful judgement in crafting the right tests. However, the authors' innovation demonstrates the power and potential of using LLMs to write creative programs for solving problems, and this advance is part of a developing success story[7,8]. As the problems intended for these models become ever more relevant, it's clear that LLMs will breathe new life into genetic programming.

1. Hsu, F.-H. *Behind Deep Blue: Building the computer that defeated the world chess champion* (Princeton University Press, 2002).

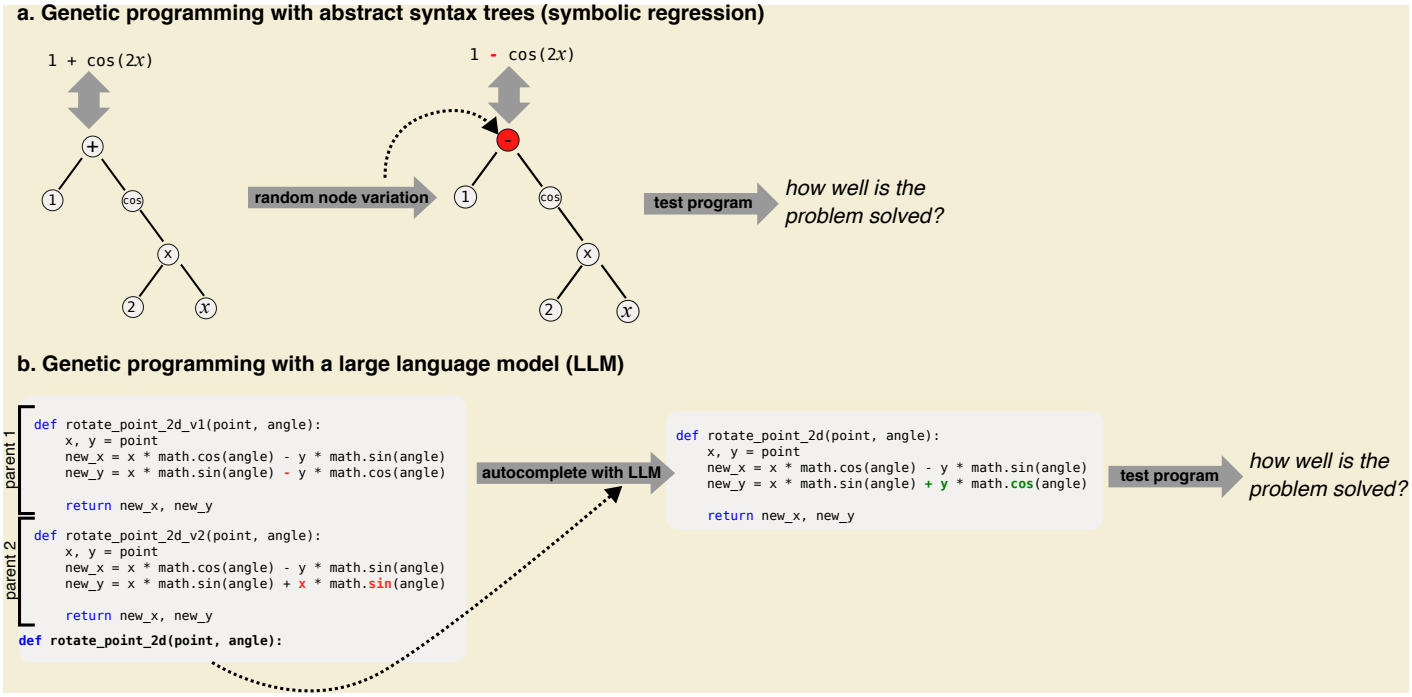

Fig. 1. Genetic programming with large language models. **a**, Genetic programming is a computer-science approach that 'mutates' code, one variation at a time. When a computer program can be represented with an abstract 'tree' structure, then genetic programming involves changing one node in the tree at random, testing to see if the change has improved the program, and then repeating the process on the highest-performing programs. **b**, Instead of making changes to an abstract tree, Romera-Paredes et al.[3] devised a genetic programming method that involves taking two programs, concatenating them, and asking a large language model (LLM) to generate a third program by autocompleting the concatenated pair. This approach enabled the authors to generate new variants of programs that are likely to be meaningful in their context.

2. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529,** 484–489 (2016).
3. Romera-Paredes, B. *et al.* Mathematical discoveries from program search with large language models. *Nature* **625,** 468–475 (2024).
4. Koza, J. R. *Genetic programming* (MIT press, 1992).
5. Schmidt, M. & Lipson, H. Distilling free-form natural laws from experimental data. *Science* **324,** 81–85 (2009).
6. Weimer, W., Nguyen, T., Le Goues, C. & Forrest, S. *Automatically finding patches using genetic programming* in *31st International Conference on Software Engineering* (2009), 364–374.
7. Lehman, J. *et al.* in *Handbook of Evolutionary Machine Learning* 331–366 (Springer, 2023).
8. Meyerson, E. *et al.* Language Model Crossover: Variation through Few-Shot Prompting. *arXiv preprint arXiv:2302.12170* (2023).