Evolutionary Computation

2025/26

Master Artificial Intelligence

Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

25/26

PSO: some more details

- The best global individual g can be included in the equation:
  - add $+U[0, \varphi_3] \circ (g - x_i)$
- The worst (local and global) positions can be *avoided*:
  - add $-U[0, \varphi_4] \circ (\overline{b}_i - x_i)$
  - and/or $-U[0, \varphi_5] \circ (\overline{h}_i - x_i)$
  - and/or $-U[0, \varphi_6] \circ (\overline{g} - x_i)$

PSO: different versions

binary version: the variables are interpreted as binary values according to a distribution or threshold

discret version: the variables are interpreted as integer values (for instance with simple rounding)

dynamic version: the search space is reinitialized and/or the local variables are reset (some type of outer Monte Carlo loop)

PSO: convergence

- the individuals should exhibit certain diversity (recall the similarity measures)
- diversity can be forced dynamically by adapting the parameters alongside simulation time
- or one might use the lack of diversity as a stopping condition

ant colony optimization (ACO)

The idea stems from stigmergy: exercise indirect communication and coordination through the environment (leave a trace and act on findings).

- The inspiration stems from ants, bees, termites, wasps, etc. (I use it at home :-)
- The individuals of a population leave information (pheromones) in the search space.
- The decisions are based on individual information or behavior and on the pheromones encountered.
- The information (pheromones) is volatile and evaporates.
- The pheromones or a statistical evaluation of the individuals define the solution.
- Initially invented to deal with combinatorial problems with construction of the solution (like TSP).

ACO: principal loop

An ant colony optimization can be summarized in the following principal loop:

```
InitializePheromoneValues()
while not Stopping():
  for individuals in range(n):
    ConstructSolution(individual)
    UpdatePheromoneValues()
    UpdateIndividuals()
```

ACO: how TSP can be approached

- The ant colony optimization takes place on the graph of the underlying problem
  (e.g., the complete graph among all cities, hence, huge memory requirements, naïvely $\mathcal{O}(n^2)$).
- The ants are placed at the cities.
- The initial pheromones are placed on the edges
  (either constant value or inversely proportional to the distance).
- The ants (in an appropriate iteration) run along a path in the graph (excluding already visited cities) and draw at each city a decision in which direction to continue.
- The decision is based on: pheromones on each possible edge, maybe on some own information stored at the individual, and on a random value.

ACO: how TSP can be approached (continued)

- Once the tour is completed for all ants, all of them deposit their pheromone on their tracks.
- The general evaporation process is applied to all/changed edges.
- The currently best tour is memorized.
- The iteration is repeated until a certain stopping condition is met.

ACO: when to use?

- ACO approaches are especially interesting when the underlying problem allows for a constructive solution (as seen, for instance, with the nearest-neighbor heuristic for the TSP).
- Simon gives the example that an ACO approach found a tour with 3% gap on the Berlin52 problem.
- Recent conference paper (2025) changes focus towards local search methods (see in the on-going) which improves over previous ACO methods
  (however: my current pair-center algorithm is 10 000 times faster *and* achieves significant lower gap with linear memory requirements).

AlphaEvolve: A coding agent for scientific and algorithmic discovery

Alexander Novikov*, Ngân Vũ*, Marvin Eisenberger*, Emilien Dupont*, Po-Sen Huang*, Adam Zsolt Wagner*, Sergey Shirobokov*, Borislav Kozlovskii*, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli and Matej Balog*
Google DeepMind[1]

In this white paper, we present *AlphaEvolve*, an evolutionary coding agent that substantially enhances capabilities of state-of-the-art LLMs on highly challenging tasks such as tackling open scientific problems or optimizing critical pieces of computational infrastructure. *AlphaEvolve* orchestrates an autonomous pipeline of LLMs, whose task is to improve an algorithm by making direct changes to the code. Using an evolutionary approach, continuously receiving feedback from one or more evaluators, *AlphaEvolve* iteratively improves the algorithm, potentially leading to new scientific and practical discoveries. We demonstrate the broad applicability of this approach by applying it to a number of important computational problems. When applied to optimizing critical components of large-scale computational stacks at Google, *AlphaEvolve* developed a more efficient scheduling algorithm for data centers, found a functionally equivalent simplification in the circuit design of hardware accelerators, and accelerated the training of the LLM underpinning *AlphaEvolve* itself. Furthermore, *AlphaEvolve* discovered novel, provably correct algorithms that surpass state-of-the-art solutions on a spectrum of problems in mathematics and computer science, significantly expanding the scope of prior automated discovery methods (Romera-Paredes et al., 2023). Notably, *AlphaEvolve* developed a search algorithm that found a procedure to multiply two 4×4 complex-valued matrices using 48 scalar multiplications; offering the first improvement, after 56 years, over Strassen's algorithm in this setting. We believe *AlphaEvolve* and

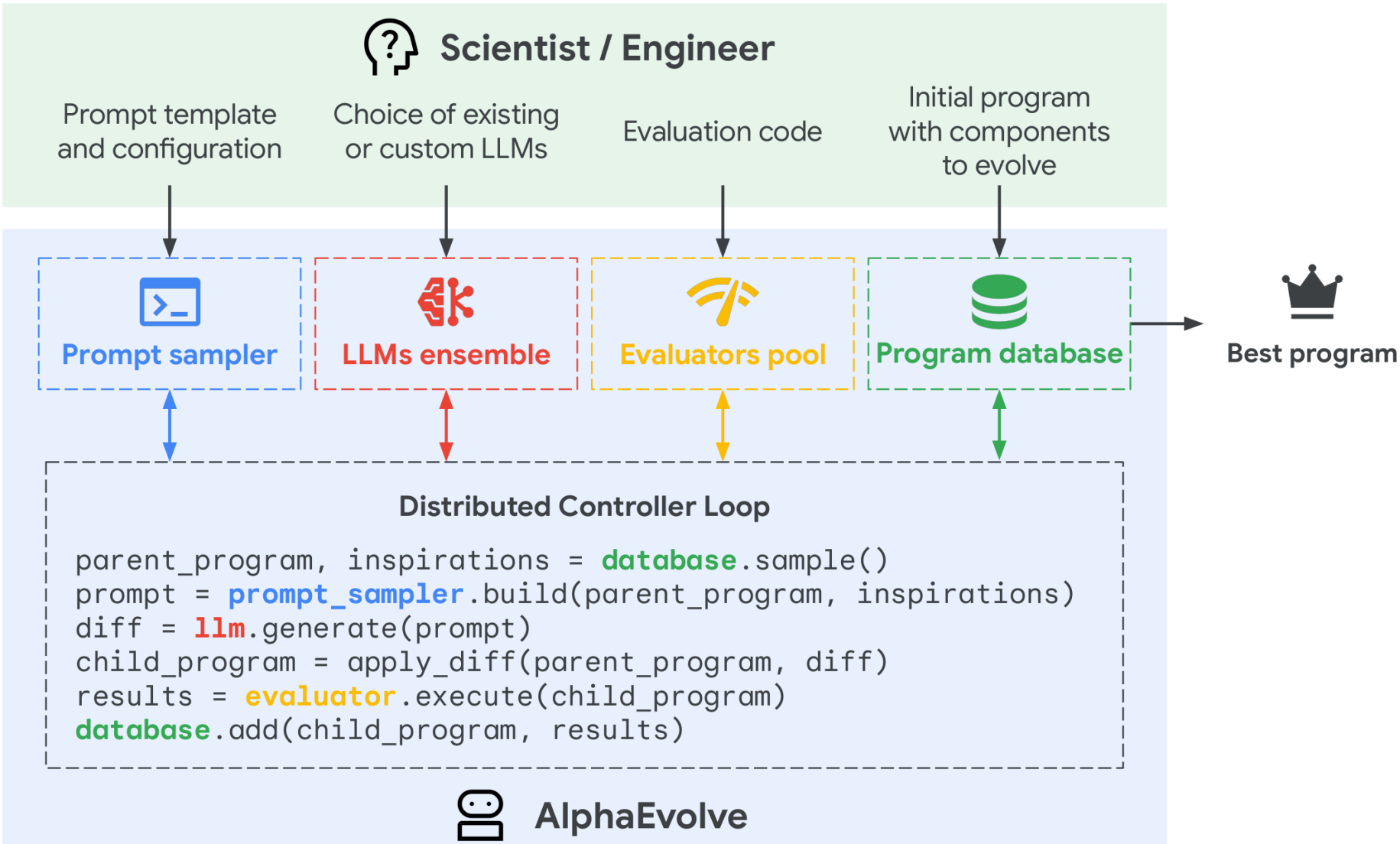
matrix multiplication: recent significant advances

multiply two matrices $10^6 \times 10^6$ (gigahertz item processing)

method	$*$ **per** 4×4	**order**	**runtime**
school method	64	$\mathcal{O}(n^3)$	31.7 years
Schönhage-Strassen	49	$\mathcal{O}(n^{2.807})$	2.2 years
AlphaEvolve	48	$\mathcal{O}(n^{2.792})$	1.8 years

- The figures are not exact, as *item processing* is not really equal for all methods; other factors (e.g., adds, data transport, cache behavior) must be considered.
- The methods have a different numerical robustness.
- For binary matrices (in $\mathbb{Z}_2$, no robustness problems) there is an even faster method, $\mathcal{O}(n^{2.772})$, detected in 2022 (AlphaTensor), so 1.5 years of runtime.
- The currently (2024) best known lower bound is $\mathcal{O}(n^{2.371})$, however with huge hidden constants, nevertheless—being naïve—we could come down to roughly 2 days.
  Hence, there is still a way to go!

samples of recent advances in matrix multiplication



The no-free-lunch theorem

The no-free-lunch theorem states that the performance of all optimization (search) algorithms, amortized over the set of all possible functions, is equivalent.

The implications of this theorem are far reaching, since it implies that no general algorithm can be designed, so that it will be superior to a linear enumeration of the search space (i.e. exhaustive search).

What are practical implications of the no-free-lunch theorem?

- Each problem (or each type/class of problem) might need its own and proper optimization method.
- Maybe for interesting problems we find good optimization algorithms (we are not interested in *all* problems).
- Benchmarking optimization algorithms is a challenge, as general benchmarks might just provide *average* data, but our algorithm might be special for a niche of problems.
- There is a need to categorize problems and algorithms to obtain some insight on which type of problem a certain type of algorithm performs well.

How to compare different approaches?

In order to compare different algorithms one might take into account:

- wall clock runtime on comparable systems
- (average) number of objective functions evaluations
  (but the rest of the run time must not be neglected in experimental comparisons)
  difficult to be used when comparing constructing algorithms
- distance to optimium or to some known lower bound
- mean best fitness
- properties of the solution histogram
  (fitness of all solutions found)
- scaling properties with problem size
  (applied to any measure above)

Practical aspects to be considered

One has to decide what is really needed:

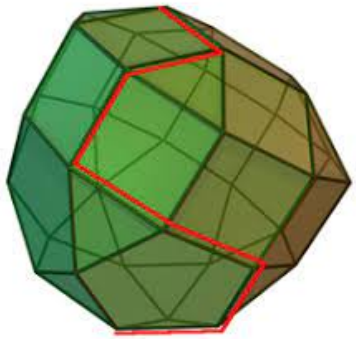
- need a good (or best) solution independent of runtime
  (e.g. controler for space telescope or the evolved antenna)
- need a moderate solution fast
  (e.g., daily TSP with time windows, where finding a feasible solution is already NP-hard)

Local search methods

- Local search methods explore the search space by inspecting (close or far) neighbor solutions.
- They stop at a local minimum, i.e., all neighbors are greater (remind: we are searching for a minimum).

Local search methods

- a classical example is the simplex method for linear programming



- or the Newton method (or Newton-Raphson method) applied to optimization (here formulated as maximization)

```
while GradientFobj(xi) > tolerance:
  xi=xi-GradientFobj(xi)/SecondDerivativeFobj(xi)
```

(Take care: should check if eventually really maximum, and not minimum.)

Local search methods

Further classic iterative optimization methods for the minimization of real-valued functions that need gradient information, are:

- Gauss-Newton method
  (variation of the Newton-method by using the Jacobean-matrix instead of the Hessian-matrix)
  second derivatives are not required here
- gradient descent (or, similarly, steepest decent)
- Levenberg-Marquardt methods
  (interpolation between Gauss-Newton methods and gradiant descent)
- Nesterov's method for convex optimization
  (with much faster convergence rate compared to gradient decent)