APRIL, 2026

GI-Agent

Search-Based LLM Agent for Code Optimization with Genetic Improvement

DONGHYUN LEE

MENG @ UCL

WILLIAM B. LANGDON

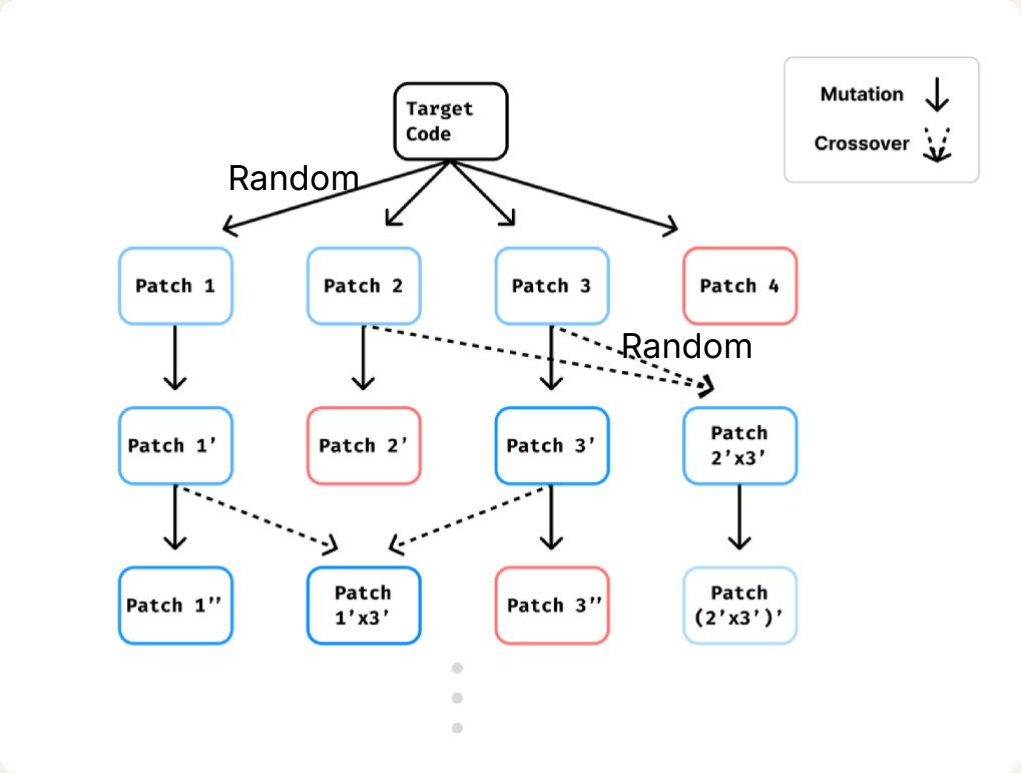
PROFESSOR @ UCL

JUSTYNA PETKE

PROFESSOR @ UCL

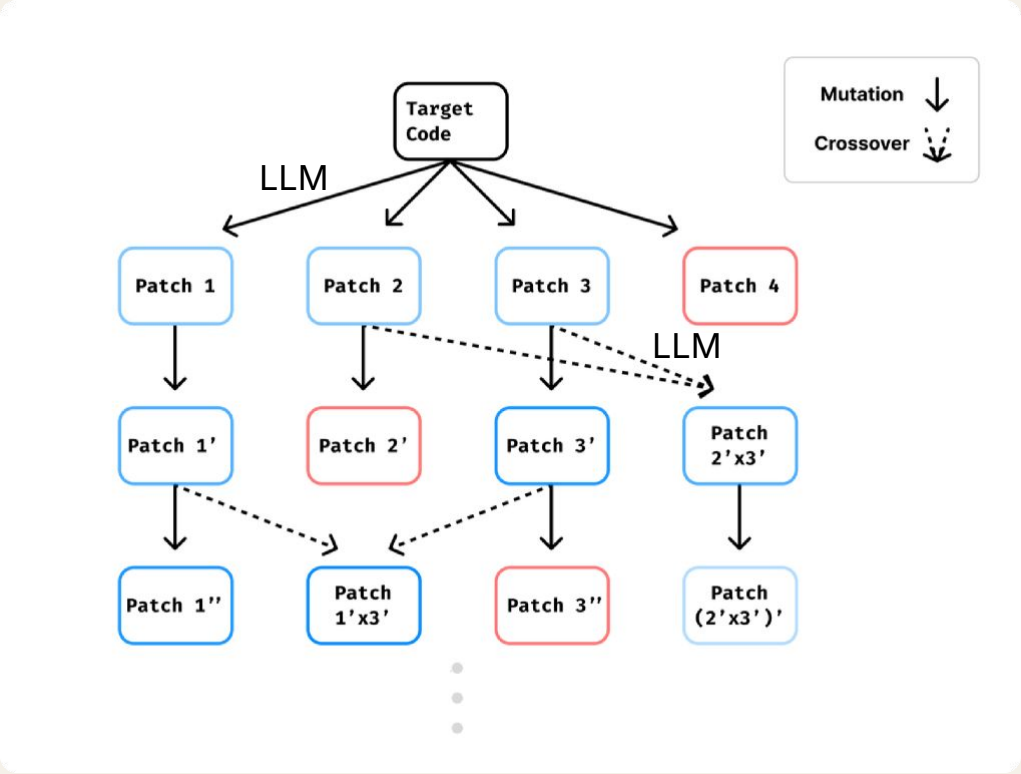
Genetic Improvement

GI has shown success for software engineering applications.
But they rely on **random operations** without contextual awareness.



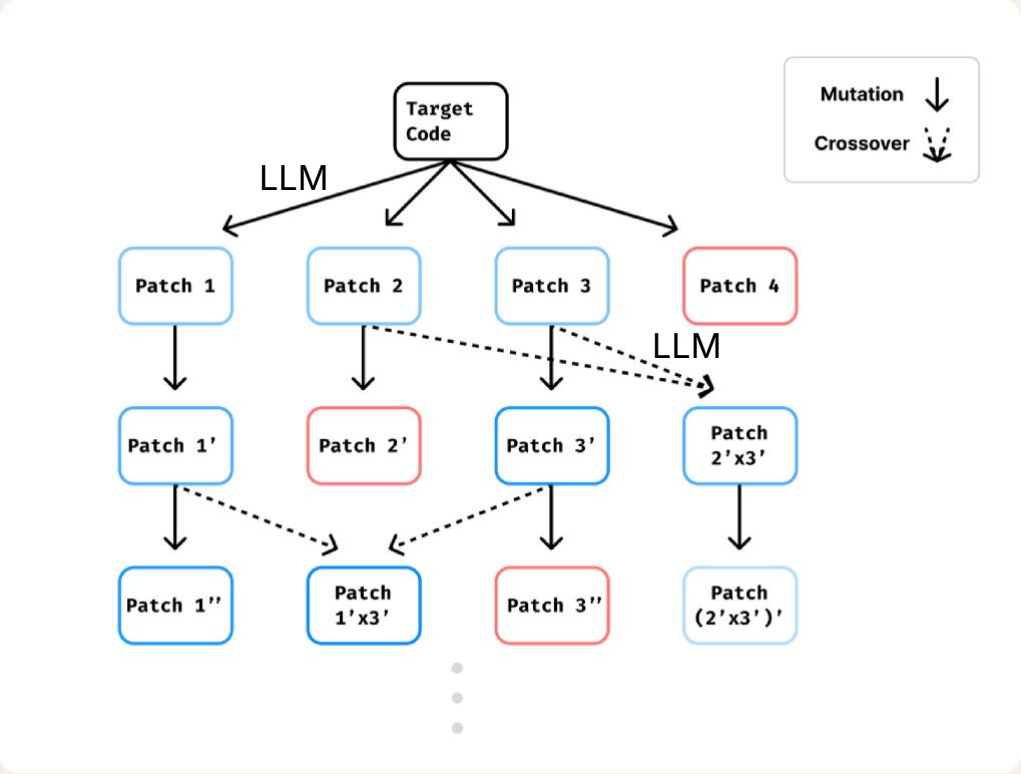
Genetic Improvement x LLMs

Brownlee (2023) and Bouras et al (2025) each showed that **LLM-assisted** mutations and crossovers improve GI performance. LLMs can perform **semantic-aware** operations.



Genetic Improvement x LLMs

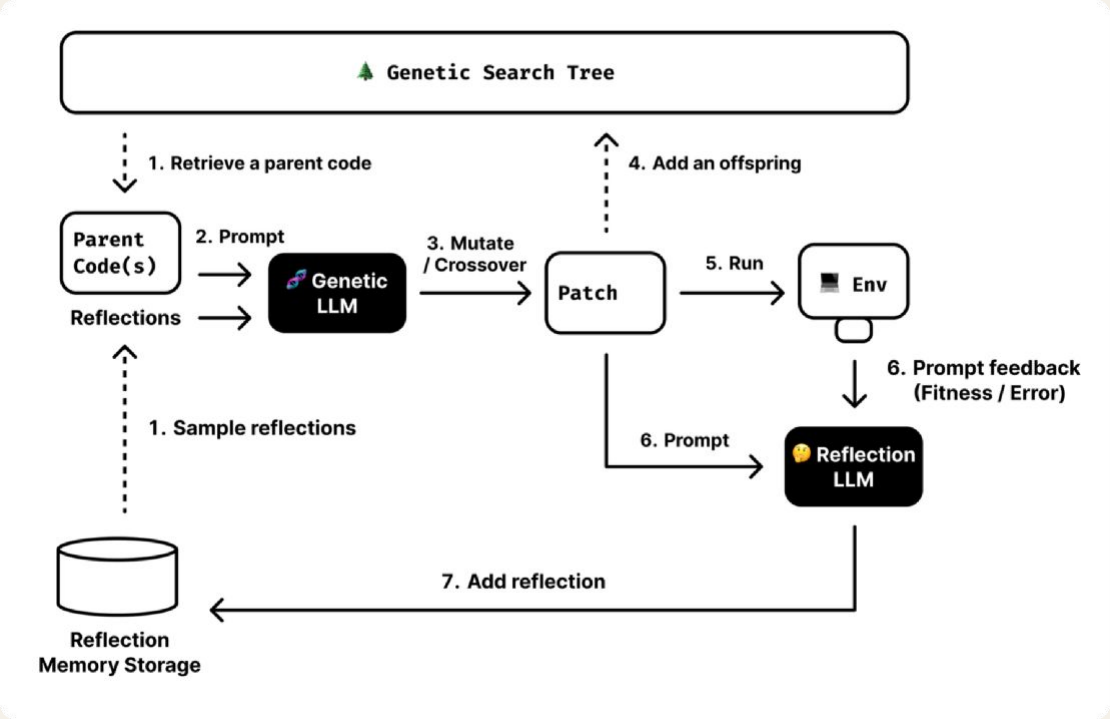
Brownlee (2023) and Bouras et al (2025) each showed that **LLM-assisted** mutations and crossovers improve GI performance.
LLMs can perform **semantic-aware** operations.
But LLMs are static **without learning**.



In this agentic era...

Can LLMs *learn* throughout the interaction with the environment to perform better search operations?

Key ideas of GI-Agent



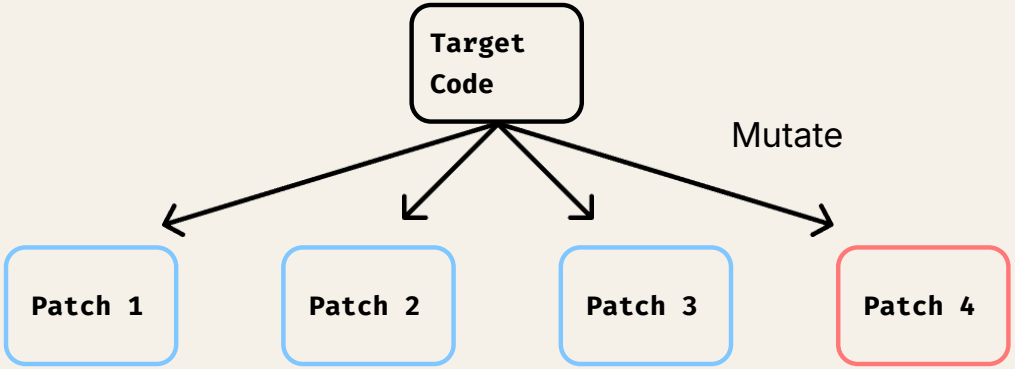
An LLM agent that acts (mutate/crossover), evaluates, learns, remembers, and repeats to maximize an objective score

GI-Agent

1. Act: LLM are in charge of all mutations and crossovers
2. Evaluate: LLM-generated programs are evaluated in the environment
3. Learn: LLM reflects on its own operation and the results
4. Remember: The reflections are fed to LLMs in the next operations
5. Repeat

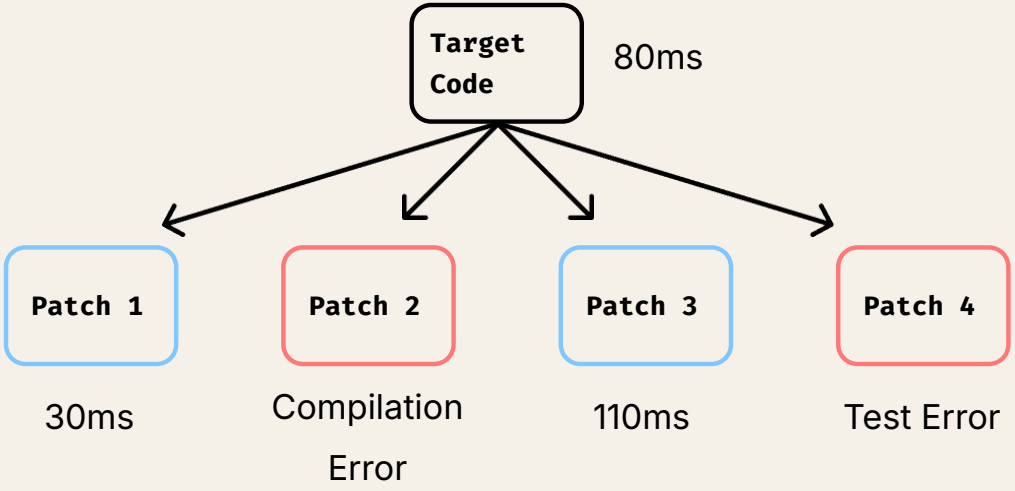
GI-Agent

1. **Act:** **LLM are in charge of all mutations and crossovers**
2. Evaluate: LLM-generated programs are evaluated in the environment
3. Reflect: LLM reflects on its own operation and the results
4. Remember: The reflections are fed to LLMs in the next operations
5. Repeat



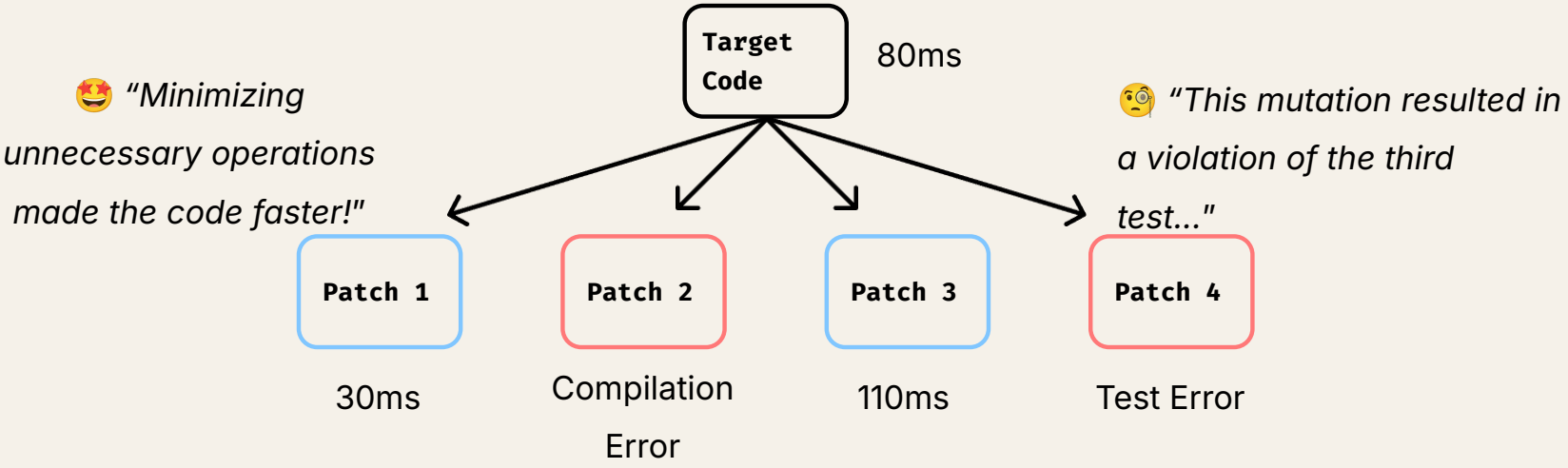
GI-Agent

1. Act: LLM are in charge of all mutations and crossovers
2. **Evaluate: LLM-generated programs are evaluated in the environment**
3. Reflect: LLM reflects on its own operation and the results
4. Remember: The reflections are fed to LLMs in the next operations
5. Repeat



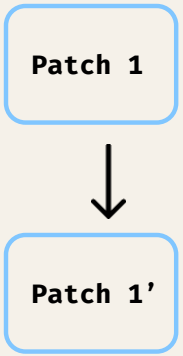
GI-Agent

1. Act: LLM are in charge of all mutations and crossovers
2. Evaluate: LLM-generated programs are evaluated in the environment
3. **Reflect: LLM reflects on its own operation and the results**
4. Remember: The reflections are fed to LLMs in the next operations
5. Repeat



GI-Agent

1. Act: LLM are in charge of all mutations and crossovers
2. Evaluate: LLM-generated programs are evaluated in the environment
3. Learn: LLM reflects on its own operation and the results
4. **Remember: The reflections are fed to LLMs in the next operations**
5. Repeat

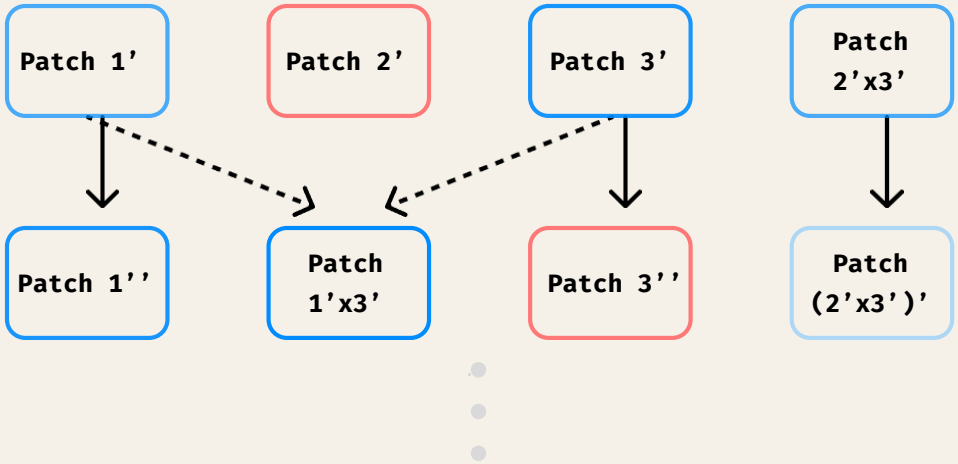


+ reflections from the past

20ms

GI-Agent

1. Act: LLM are in charge of all mutations and crossovers
2. Evaluate: LLM-generated programs are evaluated in the environment
3. Learn: LLM reflects on its own operation and the results
4. Remember: The reflections are fed to LLMs in the next operations
5. **Repeat**

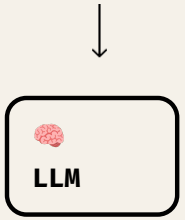


Reflection

Parent code(s) Parent score(s)

Child code(s) Child score(s)

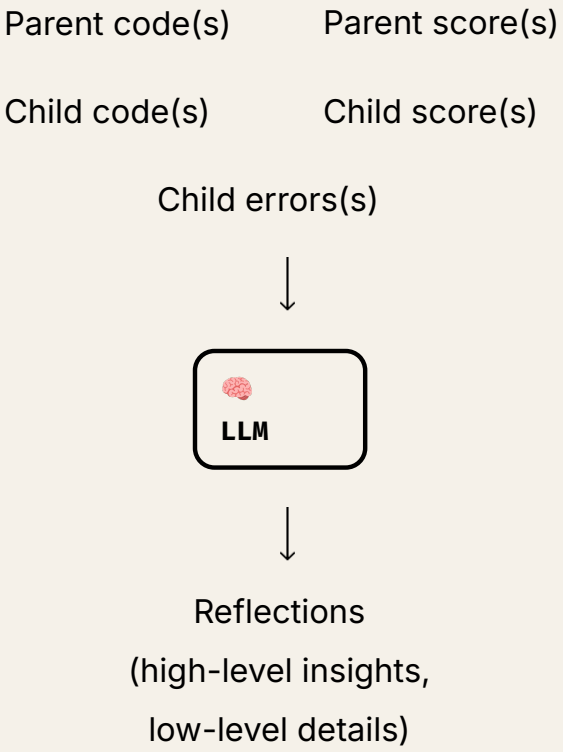
Child errors(s)



Reflections
(high-level insights,
low-level details)

```
You are an expert developer. Previously, you were given a code and you wrote a new code
to make the code faster by improving the fitness score.

This was the parent code you were given:
{parent_codes}

This was the fitness score of the parent code:
{parent_fitnesses}

This was the new code you wrote:
{child_code}

Here is the run result:
{run_result}

Here is the run stderr (if the code failed):
{run_stderr}

New code fitness score:
{run_fitness}
(-1 means the code failed to run. The positive, smaller fitness score means the new code is
    faster.)
Now, you need to reflect on this.
The purpose is to learn what strategy worked, what didn't, so that you can later use this
    compressed knowledge to write better codes.
For example, if the new code is faster, you should reflect on what strategies made it faster.
If the new code is slower, you should reflect on what strategies made it slower.
If the test or compile failed, you should reflect on what went wrong by looking at the stderr.
```

Reflection



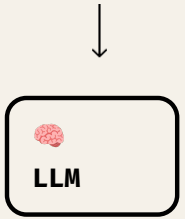
Outcome	Reflection
Speedup	Removing the call to sort literals in the clause addition routine significantly reduced overhead during clause insertion, yielding a roughly 20% speed improvement. Other minor refactoring and formatting had minimal impact. This suggests that minimizing unnecessary array operations in hot paths, like eliminating `sort(ps)` in `addClause`, can substantially enhance SAT solver runtime.
Increased Latency	Disabling the clause database reduction routine to save processing time backfired. Skipping reduction increased clauses processed during propagation, raising the fitness score from 0.673 (parent) to 1.040 (slower execution). Periodic clause reduction is vital for managing memory and propagation overhead. Future attempts should adjust reduction frequency rather than eliminate it.
Compilation Error	Replacing index-based iteration with pointer arithmetic using `ws.begin()` and `ws.end()` failed because `Minisat::vec` lacks an `end()` method, causing a compile-time error. Assuming STL conventions for non-STL containers proved risky, breaking the code instead of improving watcher iteration speed.
Test Error	The modifications made to the propagation loop—replacing pointer iteration with an index-based `writeIdx` approach and using `std::swap` to rearrange literals—caused an invariant violation. Specifically, the expected condition that the clause's second literal equals the false literal was not maintained, triggering an assertion failure. This indicates that our optimization in the propagate routine, intended to enhance performance by reducing unnecessary copying, instead disrupted the clause structure, leading to runtime failure. Overall, the intended performance benefit was outweighed by breaking critical invariants.

Mutation

Parent code

Reflections

{number of children to generate}



<Mutation N>

strategy: I will simplify the lines 34 ~ 54 ...

- array.sort()

+ # remove this

```
Here's the code to mutate:
{code}

Here are reflections from previous trials and errors that you may use to generate mutations:
{reflections}

[OUTPUT FORMAT]
<Mutation N>
strategy: Focus on the method / line / etc. that do X, which could be optimized by Y.
- target code block line 1
- ...
- target code block line N
+ new code line 1
+ ...
+ new code line N

Now, generate {num_offsprings} mutations.
```

Crossover

Parent codes

Parent scores

Reflections

{number of children to generate}

<Crossover N>

strategy: I will combine A and B

- array.sort()

+ # remove this

```
Here is the original source code to be optimized:
{original_source_code}

Now, here are the parent codes with their fitness scores.
Each parent code is provided as a code diff:
{codes_and_fitnesses}

Here are reflections from previous trials and errors that you may use to generate mutations:
{reflections}

[OUTPUT FORMAT]
<Crossover N>
strategy: Combine the method, line, etc. from Parent X that does A with the method, line, etc.
    from Parent Y that does B, optimizing for C.
- target code block line 1
- ...
- target code block line 3
+ new code line 1
+ ...
+ new code line 3

Now, generate {num_offsprings} crossovers.
```

Evaluation

Benchmarks

- SAT4J : large Java SAT solver (~2.5k lines), modular
- MiniSAT : compact C++ SAT solver (~1k lines), well-studied in GI
- Objective: Find a faster, correct code

Configs

- population size=20, max_steps=220 (10 gen)
- 60% mutation / 20% crossover / 20% elitism
- 20 runs per method, Mann-Whitney U ($p<0.05$)

Model

- GPT-o3 Mini

Evaluation

Methods

	LLM Mutations & Crossovers	Reflect on all codes	Reflect on only faster & correct codes
Traditional GI			
GI-Agent (NoReflect)	✅		
GI-Agent (ReflectAll)	✅	✅	
GI-Agent (ReflectSuccess)	✅		✅

GI-Agent can find faster programs

	SAT4J			MiniSAT		
Method	Min	Median	Max	Min	Median	Max
GI	0.0%	1.8%	2.9%	0.0%	15.2%	81.0%
NoReflect	0.0%	2.9%	8.3%	**10.3%**	**34.9%**	**82.9%**
ReflectAll	**1.2%**	2.6%	**9.9%**	4.8%	30.8%	80.9%
ReflectSuccess	**1.2%**	**3.5%**	5.7%	9.8%	31.5%	80.8%

Speedup percentages over the baseline

In SAT4J, reflections helped GI-Agent find faster programs.

In MiniSAT, reflections didn't help, but LLM-driven GI found faster solutions.

GI-Agent discovers high-quality programs efficiently

	SAT4J					MiniSAT				
Method	20%	40%	60%	80%	100%	20%	40%	60%	80%	100%
Traditional GI	0.6%	1.5%	1.8%	1.8%	1.8%	0.3%	7.8%	8.1%	14.9%	15.2%
NoReflect	1.8%	2.3%	2.6%	2.9%	2.9%	**16.2%**	21.3%	27.4%	28.0%	**34.9%**
ReflectAll	**2.6%**	2.6%	2.6%	2.6%	2.6%	6.0%	14.0%	29.2%	**30.7%**	30.8%
ReflectSuccess	2.3%	**3.5%**	**3.5%**	**3.5%**	**3.5%**	7.5%	**28.6%**	**30.7%**	**30.7%**	31.5%

Median speedup percentages

at 20%, 40%, 60%, 80%, and 100% of the maximum steps (200)

In both benchmarks, reflections helped leverage previous insights to find faster programs in fewer steps

LLM operations & reflections hurt the diversity of candidates

	SAT4J			MiniSAT		
Method	Min	Median	Max	Min	Median	Max
Traditional GI	28.9%	**37.3%**	**44.3%**	**23.9%**	**36.6%**	**48.8%**
NoReflect	44.8%	52.5%	60.2%	32.8%	45.3%	53.2%
ReflectAll	**0.0%**	50.7%	61.2%	39.8%	44.8%	55.2%
ReflectSuccess	46.8%	51.2%	58.2%	35.5%	44.3%	50.7%

Percentages of repeated variants

GI, thanks to its randomness, finds less repeated variants.

Reflections reduces the diversity even more.

After our paper, LLM-based search field itself has evolved

- **2025 April: GI-Agent**
- **2025 May: Google's AlphaEvolve** optimizes Google data centers and chip designs

  Shortly after, Algorithmic SuperIntelligence Labs releases **OpenEvolve**
- **2025 July: UC Berkeley's GEPA** uses LLM-based evolution and becomes a SOTA prompt optimizer
- **2025 September: Sakana AI's ShinkaEvolve** discovers a SOTA circle packing solution
- **2026 Feburary: UC Berkeley** releases **optimize_anything** and **SkyDiscover,** which are flexible LLM-driven optimization frameworks for any optimization tasks (system, prompt, code, agent optimization, etc.)

Conclusion

- GI-Agents can find faster programs better and faster when reflections from prior trials and errors are provided
- The study of combining a GI framework and LLMs has rapidly grown recently
- Encouraging LLMs to explore a search space beyond their bias still remains as a key challenge
- Future work includes exploring the use of modern, powerful Claude Code agents that can perform better GI operations