Automated Software Performance Improvement with Magpie

Aymeric Blot
Université de Rennes, France
April 13, 2026 — GI@ICSE 2026, Rio de Janeiro

Personal Background

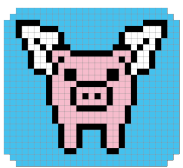
Aymeric Blot

- Lecturer @ Université de Rennes
- Research Fellow @ IRISA
- Member @ DiverSE (Inria/IRISA)
- Developing Magpie
- Contributed to PyGGI 2.0
- Developed MO-ParamILS

Université de Rennes

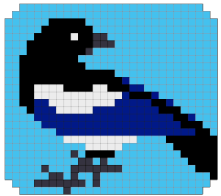
UMR IRISA Inria

DiverSE
Diversity-Centric Software Engineering



What is... Magpie?

Metaheuristic Automation for General Performance Improvement of SoftwarE

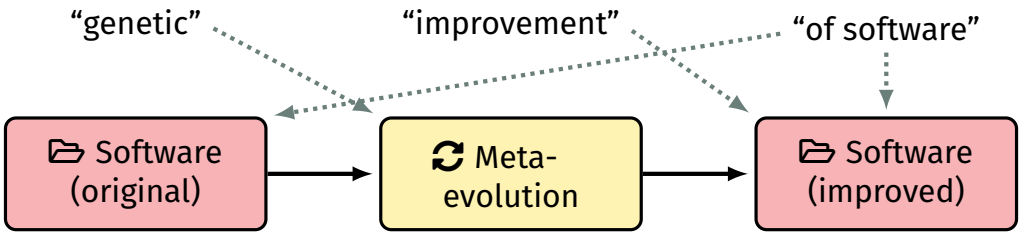


Magpie is...
a Python framework...
to model software...
and automate searching...
for improved software variants

Key points

- Free, libre, and open-source
- Small sized but very versatile
- Multi-purpose and trustworthy

What is... Genetic Improvement of Software?

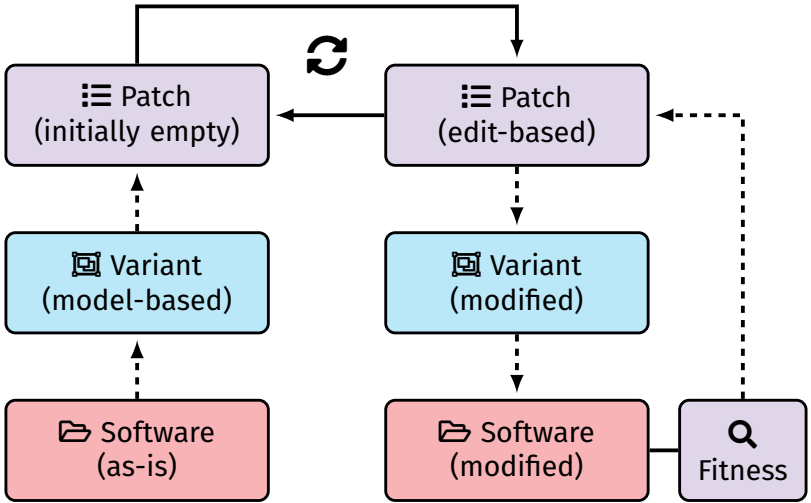


Core targets

- **Functional properties:** e.g., automated program repair
- **Non-functional properties:** performance improvement
  → e.g., execution time, energy/memory usage, solution quality...

Magpie: A Unified Transformation-Centred Approach

For Genetic Improvement, Parameter Tuning, Algorithm Configuration, etc.



Terminology

Variant

A *set* of **models** on which to apply **patches**

Model

An *internal representation* of *one part* of the target software
→ *array* of lines of code, syntax *tree*, parameter *mapping*, ...

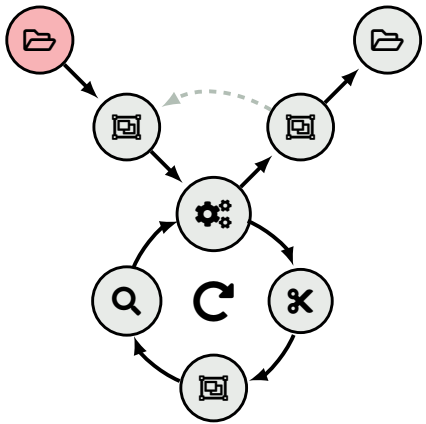
Patch

A *sequence* of **edits**

Edit

A single **model** *transformation*
→ `LineReplacement(('foo.rb', 'line', 11), (foo.rb', 'line', 5))`

Metaheuristic Search for Improved Variants



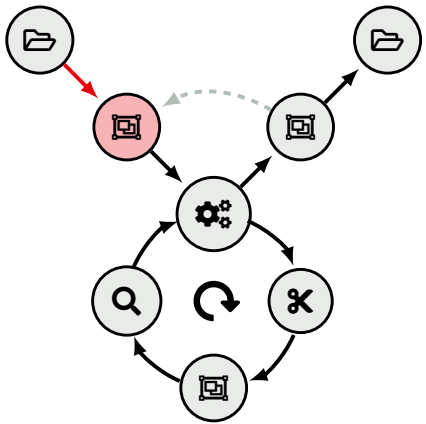
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



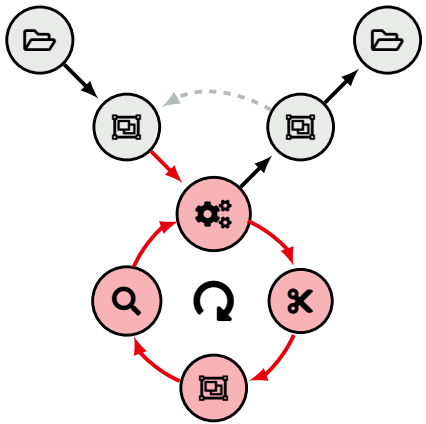
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



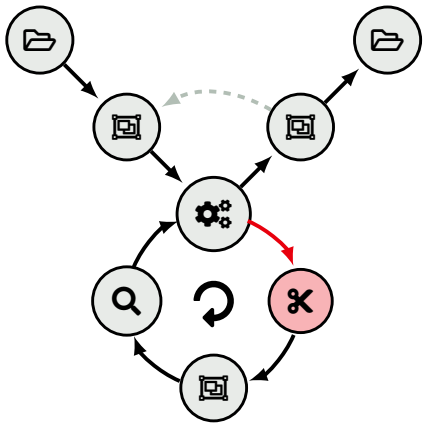
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



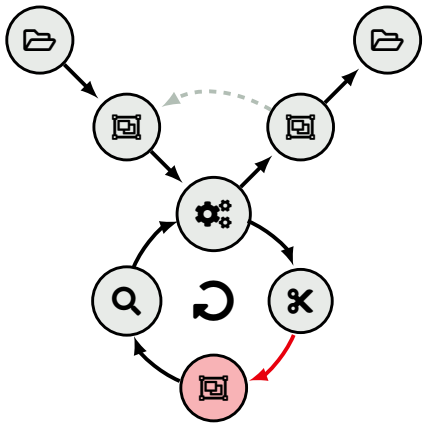
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



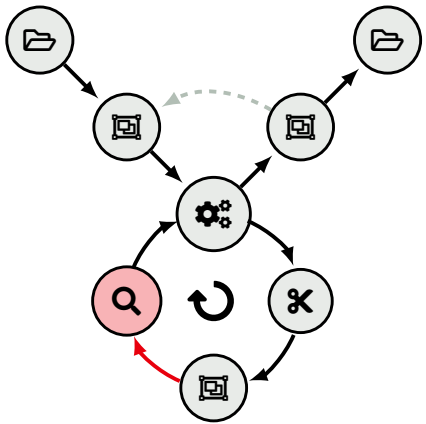
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



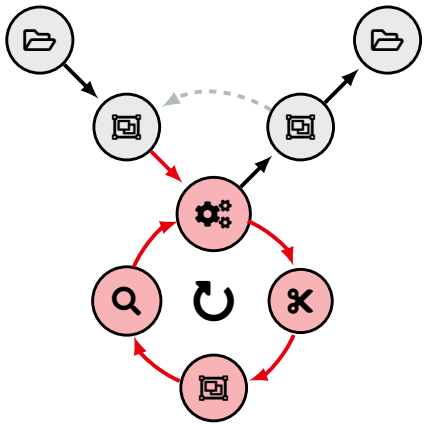
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



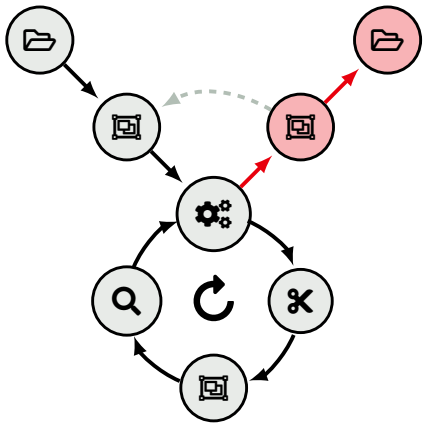
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



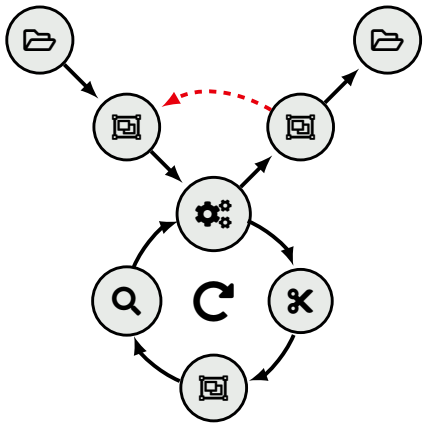
Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Metaheuristic Search for Improved Variants



Standard search procedure

1. Start with existing software
2. Abstract parts on which to focus
3. Evolve a patch (GP, LS)
   - 3.1 Add/remove edit
   - 3.2 Apply to create new variant
   - 3.3 Evaluate fitness function
4. Enjoy your improved software

Optional additional steps

- Test for generalisation
- Minimise patch size
- Manually assess semantics

Why Does it Work?

The "*competent programmer*" hypothesis (from MT)

Competent programmers write programs that are *almost* correct

The "*plastic surgery*" hypothesis (from APR)

1. Program source code changes occurring during development contain snippets that already exist in the codebase
2. These snippets can be efficiently found and exploited

The "*your software is not unique*" observation (from *me*)

There are *infinitely many* equivalent software, in a *fractal* fashion

Hot take: GenAI may remove the need of both hypotheses!

Magpie Key Components

Software

- compilation
- execution
- test suite

Models + Edits

- lines of code
- syntax trees
- CLI parameters

Search

- local search
- genetic programming
- patch minimisation
- ablation analysis

Fitness

- execution time
- resource usage
- program repair
- output-based

Strong focus on modularity and reproducible experiments

Magpie 1.3(?) in a Nutshell

Added

- support for arbitrary expressions in fitness functions
  - (e.g., `(perf<cpu-cycles>-perf<instructions>)/1000`)
- new syntax for configurations models
  - (e.g., `show warmup_steps if scheduler in ["linear", "cosine"]`)
  - (e.g., `set c = 100 - a - b`)
  - (e.g., `assert (a + b <= 100) and (a > 10 or b > 10)`)
- misc (Gradle support, read-only models, more ENV data, ...)

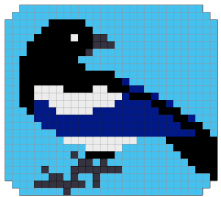
Changed

- added human-readable dates to log filenames (YYYYMMDD)
- changed multi-objective syntax (“ ” to “;”)

Fixed

- minor issues with warmup values aggregation

Try Magpie Now!



Magpie

- Modular Python framework for GI/MT/AC/CO
- Hack-friendly, for researchers!
- User-friendly, for developers!

Plug-and-play!

```
> git clone https://github.com/bloa/magpie.git
> cd magpie
> python3 magpie local_search \
      --scenario examples/triangle-c/_magpie/scenario_slow.txt
```

Magpie for Developers

```
magpie/ (← the Git repository)
├── docs/ (← Diátaxis-based - 🔗 https://diataxis.fr/)
├── examples/
│   └── triangle-c/ (← target software)
│       └── _magpie/ (← scenario files)
├── magpie/
│   ├── core/ (← base classes + scenario config ≈ 1600 loc)
│   ├── models/ (← line, xml, config, astor ≈ 1300 loc)
│   ├── algos/ (← LS, GP, validation, ablation ≈ 800 loc)
│   ├── fitness/ (← time, repair, bloat, output ≈ 200 loc)
│   ├── bin/ (← entry points ≈ 200 loc)
│   └── __main__.py (← main entry point)
└── tests/
```

Note: no dependencies, Magpie is entirely self-contained!

Magpie in Practice

```
/
├── magpie/ (← the one containing "__main__.py")
├── your_software/
├── scenario.txt
├── _magpie_work/ (← automatically generated)
│   └── your_software_167874800/
└── _magpie_logs/ (← automatically generated)
    ├── your_software_20261412_1776026828.log
    ├── your_software_20261412_1776026828.patch
    └── your_software_20261412_1776026828.diff
```

Magpie in a nutshell

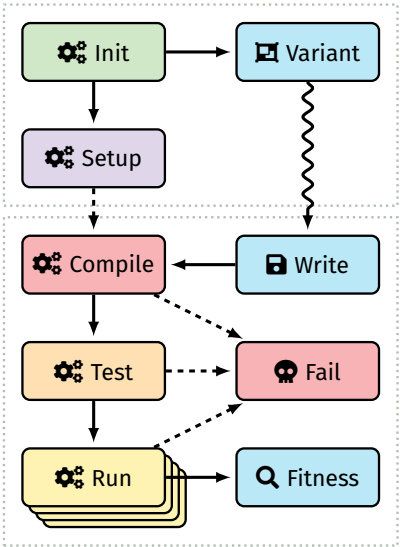
```
› python3 magpie local_search --scenario scenario.txt
› python3 magpie minify_patch --scenario s.txt --patch p.txt
```

Scenario File: INI Configuration File

```
1  [software]
2  path = examples/triangle-rb
3  target_files =
4      triangle.rb
5  fitness = repair
6
7  init_cmd = bash init_bug.sh
8  test_cmd = ruby test_triangle.rb
9
10 [search]
11 target_fitness = 0
12 max_steps = 100
13 possible_edits =
14     LineReplacement
15     LineInsertion
16     LineDeletion
```

Note: by default, model association is done through file extension

Evaluation Pipeline



At the start of the search

- **init:** environment preparation
  → **always executed**
- **setup:** pre-processing
  → **only when required**

For every variant

- **compile:** test/run preparation
- **test:** semantic check
- **run:** fitness computation
  → **can be repeated**

Magpie Hands-on!

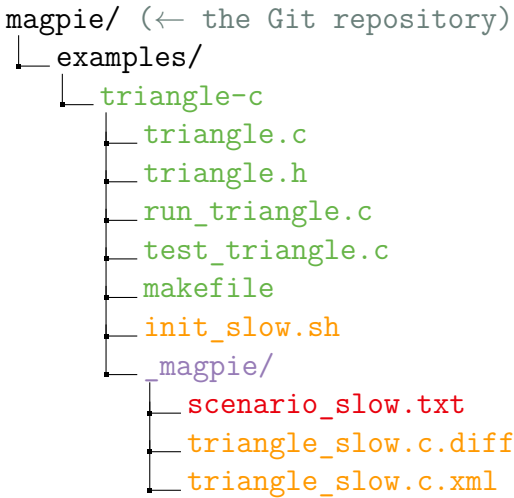
Simple use case – Triangle

- evaluation pipeline
- Magpie's output
- basic endpoints

Advanced use cases – MiniSAT

- line-level evolution
- AST-level evolution
- parameter tuning
- multi-granularity
- instance batch processing

First Steps: triangle-c



The Scenario

examples/triangle-c/_magpie/scenario_slow.txt

```
[software]
path = examples/triangle-c
target_files = triangle.c.xml
fitness = time

init_cmd = bash init_slow.sh
compile_cmd = make test_triangle run_triangle
test_cmd = ./test_triangle
run_cmd = ./run_triangle
run_timeout = 1

[search]
max_steps = 100
max_time = 60
possible_edits =
    XmlNodeDeletion<stmt>
    XmlNodeReplacement<stmt>
    XmlNodeInsertion<stmt,block>
```

The “Bug”

examples/triangle-c/_magpie/triangle_slow.c.diff

```
--- triangle.c
+++ triangle_slow.c
@@ -1,9 +1,16 @@
 #include "triangle.h"

+void delay() {
+   const struct timespec ms = {0, 0.001*1e9}; //(0.001s)
+   nanosleep(&ms,NULL); /*ignores possible errors*/
+}
+
 int classify_triangle(double a, double b, double c) {
    double tmp;

+   delay();
+
    // sort the sides so that a <= b <= c
    if(a > b) {
```

Running Magpie!

```
› python3 magpie local_search \
      --scenario examples/triangle-c/_magpie/scenario_slow.txt
```

```
1  ==== SEARCH: FirstImprovement ====
2  ~~~~ WARMUP ~~~~
3  WARM   SUCCESS              0.07 (--) [0 edit(s)]
4  WARM   SUCCESS              0.08 (--) [0 edit(s)]
5  WARM   SUCCESS              0.08 (--) [0 edit(s)]
6  REF    SUCCESS              0.08 (--) [0 edit(s)]
7
8  ~~~~ START ~~~~
9  1      TEST_CODE_ERROR      None (--) [1 edit(s)]
10 2      SUCCESS             *0.08 (96.18%) [1 edit(s)]
11 3      SUCCESS             *0.07 (92.74%) [2 edit(s)]
12 4      SUCCESS              0.08 (102.29%) [1 edit(s)]
13 5      SUCCESS              0.08 (96.18%) [1 edit(s)] [cached]
14 6      TEST_CODE_ERROR      None (--) [3 edit(s)]
15 7      TEST_CODE_ERROR      None (--) [3 edit(s)]
16 8      SUCCESS              0.15 (188.92%) [3 edit(s)]
```

Multi-Step Evaluation and Logging

Configured in scenario file

- `{step}_cmd`
- `{step}_timeout`
- `{step}_lengthout`
- `batch_timeout`
- `batch_lengthout`

Step-specific logging

- `{step}_CLI_ERROR`
- `{step}_CODE_ERROR`
- `{step}_TIMEOUT`
- `{step}_LENGTHOUT`
- `{step}_PARSE_ERROR`
- `SUCCESS` (i.e., fitness value)

Note: Fails during warmup trigger automated self-diagnostic

Finding Variants

```
24      TEST_CODE_ERROR        None (--) [4 edit(s)]
25      COMPILE_CODE_ERROR     None (--) [4 edit(s)]
26      SUCCESS               *0.00 (4.08%) [2 edit(s)]
^C~~~~ END ~~~~

==== REPORT ====
Termination: keyboard interrupt
Log file: /home/aymeric/git/magpie/_magpie_logs/triangle-c_17
Patch file: _magpie_logs/triangle-c_1712601481.patch
Diff file: _magpie_logs/triangle-c_1712601481.diff
Reference fitness: 0.08
Best fitness: 0.00

==== BEST PATCH ====
XmlNodeInsertion<stmt,block>(('triangle.c.xml', '_inter_block

==== DIFF ====
--- before: triangle.c
+++ after: triangle.c
```

Final Diff

```
--- before: triangle.c
+++ after: triangle.c
@@ -2,7 +2,7 @@

 void delay() {
    const struct timespec ms = {0, 0.001*1e9}; //tv_sec=0, tv_
-   nanosleep(&ms,NULL); /*ignores possible errors*/
+    /*ignores possible errors*/
 }

 int classify_triangle(double a, double b, double c) {
@@ -40,6 +40,7 @@
    if(a == b || b == c)/*auto*/{

      return ISOSCELES;
+     return EQUILATERAL;
    }/*auto*/
    return SCALENE;
 }
```

What to do with a Patch?

Look at it (and possibly `--keep` it)

```
› python3 magpie show_patch \
      --scenario examples/triangle-c/_magpie/scenario_slow.txt \
      --patch _magpie_logs/triangle-c_1712601481.patch \
      --keep
```

Minify it (possibly on a different scenario)

```
› python3 magpie minify_patch --scenario ...  --patch ...
```

Revalidate it (possibly on a different scenario)

```
› python3 magpie revalidate_patch --scenario ...  --patch ...
```

Analyse it piece-wise

```
› python3 magpie ablation_analysis --scenario ...  --patch ...
```

Advanced Use Case 1: Line-level Evolution

examples/minisat/_magpie/scenario_runtime_line.txt

```
[software]
path = examples/minisat
target_files = core/*.cc
fitness = time

init_cmd = bash init.sh
setup_cmd = bash compile.sh
compile_cmd = bash compile.sh
test_cmd = bash test.sh
run_cmd = bash run_fixed.sh

[search]
max_steps = 100
possible_edits =
    LineReplacement
    LineInsertion
    LineDeletion
```

Please note: wildcard in target files; prophylactic compilation in setup

Advanced Use Case 2: AST-level Evolution

examples/minisat/_magpie/scenario_runtime_xml1.txt

```
[software]
path = examples/minisat
target_files = core/Solver.cc.xml
fitness = time

init_cmd = bash init.sh
setup_cmd = bash compile.sh
compile_cmd = bash compile.sh
test_cmd = bash test.sh
run_cmd = bash run_fixed.sh

[search]
max_steps = 100
possible_edits =
    XmlNodeDeletion<stmt>
    XmlNodeReplacement<stmt>
    XmlNodeInsertion<stmt,block>
```

XML AST Representation

examples/triangle-c/_magpie/triangle.c (excerpt)

```
1  // sort the sides so that a <= b <= c
2  if(a > b) {
3    tmp = a;
4    a = b;
5    b = tmp;
6  }
```

examples/triangle-c/_magpie/triangle_slow.c.xml (excerpt)

```
1  <comment type="line">// sort the sides so that a &lt;= b &l
2  <if_stmt><if>if<condition>(<expr><name>a</name> <operator>&
3    <expr_stmt><expr><name>tmp</name> <operator>=</operator>
4    <expr_stmt><expr><name>a</name> <operator>=</operator> <n
5    <expr_stmt><expr><name>b</name> <operator>=</operator> <n
6  </block_content>}</block></if></if_stmt>
```

https://www.srcml.org/ (supports C, C++, C#, Java, Python)

XML Processing

Default rules

```
1 [srcml]
2 rename =
3     stmt: break continue decl_stmt do expr_stmt for goto if r
4     number: literal_number
5 focus = block stmt operator_comp operator_arith number
6 internodes = block
7 process_pseudo_blocks = True
8 process_literals = True
9 process_operators = True
```

- `rename`: dictionary of renaming rules
- `focus`: list of tags to keep
- `internodes`: list of tags to process for future note insertion
- `process_...`: bespoke SrcML rules

Advanced Use Case 3: Parameter Tuning

examples/minisat/_magpie/scenario_runtime_config1.txt

```
1  [software]
2  path = examples/minisat
3  target_files = minisat_simplified.params
4  fitness = time
5
6  init_cmd = bash init.sh
7  setup_cmd = bash compile.sh
8  compile_cmd =
9  test_cmd = bash test.sh
10 run_cmd = bash run_fixed.sh
11
12 [search]
13 max_time = 60
14 possible_edits = ParamSetting
```

Parameter Configuration File

examples/minisat/minisat_simplified.params (excerpt)

```
1  CLI_PREFIX = "-"
2  CLI_GLUE = "="
3  CLI_BOOLEAN = "prefix"
4
5  # core
6  luby       {True, False}[True]
7  rnd-init   {True, False}[False]
8  gc-frac    e(0, 65535)[0.2]
9  rinc       e(1, 65535)[2]
10 var-decay  (0, 1)[0.95]
11
12 phase-saving  [0, 2][2]
13 ccmin-mode    [0, 2][2]
14 rfirst        g[1, 65535][100]
```

Examples: "`-luby`", "`-no-luby`", "`-gc-frac=0.2`", **etc**

More Syntax

```
# magic constants
CLI_...
TIMING="setup compile"

# categorical parameters
name1 {value1, value2, value3}[value1]
name2 {True, False, None}[None]

# numerical parameters
name3 (0, 1.0)[0] # uniform continuous
name4 e(0, 1.0)[0] # exponential
name5 [0, 100][0] # uniform discrete
name6 g[-999999999, 999999999][0] # geometric

# and more!
show name1 if name2 != False # conditional
assert name3 > 0.5 or name4 > 0.5 # assertions
set name7 = name5 + name6 # dynamic
@foo$bar {True, False}[True] # invisible stuff!
```

Advanced Use Case 4: Evolving at Multiple Granularities

```
1  [software]
2  target_files =
3      gcc.params
4      minisat.params
5      **/*.cc.xml
6
7  [search]
8  possible_edits =
9      ParamSetting
10     SrcmlNumericSetting
11     XmlNodeDeletion<stmt>
12     XmlNodeReplacement<stmt>
13     XmlNodeInsertion<stmt,block>
```

How are edits generated?

1. Select an edit type — from `possible_edits`
2. Identify target models and locations — potentially multiple times
3. Instantiate the edit — with additional data as needed

Advanced Use Case 5: Batch Processing

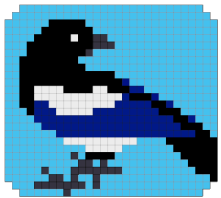
examples/minisat/_magpie/scenario_runtime_batch1.txt (excerpt)

```
1 [software]
2 run_cmd = bash run_single.sh {INST}
3
4 [search]
5 batch_instances =
6     file:data/inst_sat.txt
7     ___
8     file:data/inst_unsat.txt
9 batch_sample_size = 4
```

examples/minisat/run_single.sh

```
1 #!/bin/sh
2
3 ./simp/minisat $@
4 exit 0
```

Final Words



Magpie, One Framework to Rule Them All!

- Modular Python framework for GI/MT/AC/CO
- Hack-friendly, for researchers!
- User-friendly, for developers!

Use it now!

```
> git clone https://github.com/bloa/magpie.git
> cd magpie
> python3 magpie local_search \
      --scenario examples/triangle-c/_magpie/scenario_slow.txt
```

Please do not hesitate to contribute or reach out for support!