Execution-Guided Within-Prompt Search

Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, Sumit Gulwani
Microsoft, Redmond, USA

ICLR 2025

Code Generation from Tests

The Task: Generate code such that the given tests pass

```
assert f(a = '1 2 3', b = '2 3') == '13'

def f(a, b):
```

Challenge

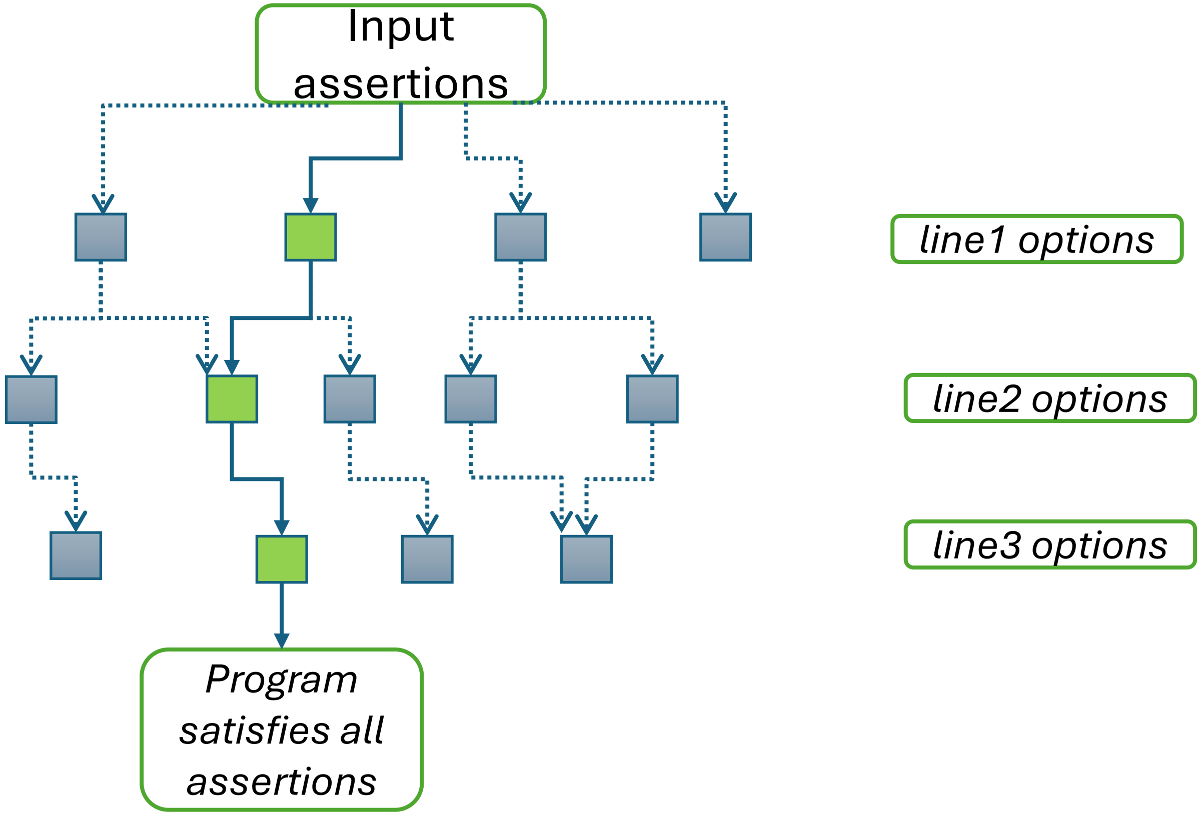
LLMs can generate code from input-output specifications, but …

they can fail

Some techniques for improving chances of success:

- Retry, as in **self-reflection**
- Reason, as in **chain-of-thought**
- Search, as in **tree-of-thought**
- **…**
- **Execution-guided within-prompt search**

The Framework

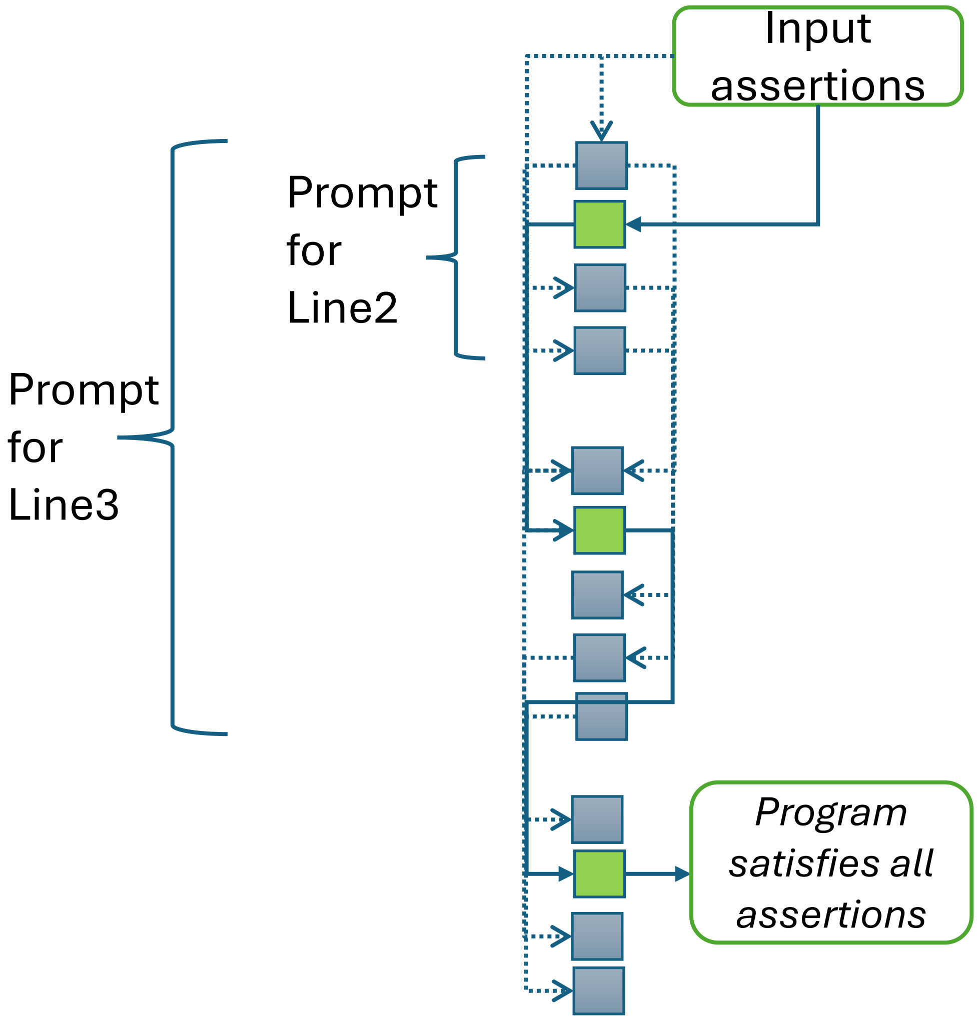


Questions:

How can we use LLMs to ***efficiently search*** for the correct program?

How can we ***guide*** the LLMs to the correct completion?

Idea 1: Within-Prompt Search



Bloated Program

1. Keep all options
2. Suitably rename variables to avoid name clashes
3. Use bloated program to predict options for next line

- LLM is the **policy** that generates actions
- Bloated program forces LLM to act as a **value function** and pick promising states

Idea 2: Execution-Guided Search

```
assert f(a = '1 2 3', b = '2 3') == '13'  # e1

def f(a, b):
    v1 = a[0]  # { "e1": "1" }
    v2 = a.split()  # { "e1": ["1", "2", "3"] }
```

2 options for Line1

Each line annotated with execution result

New options for next line generated

```
v3 = b.split()[-1]    v3 = b.split()    .........    v3 = b[4]    v3 = b.split()[0]
```

```
v3 = b.split()    # { "e1": ["2", "3"] }
v4 = v3[-1]       # { "e1": "3" }
v5 = v3[0]        # { "e1": "2" }
```

3 options for Line2

New options renamed, executed results added, and added to bloated program for next iteration

Bloated Program for Generating Line3

```
assert f(a = '1 2 3', b = '2 3') == '13'
```

```
def f(a, b):
    v1 = a[0]  # { "e1": "1" }
    v2 = a.split()  # { "e1": ["1", "2", "3"] }
    v3 = b.split()  # { "e1": ["2", "3"] }
    v4 = v3[-1]  # { "e1": "3" }
    v5 = v3[0]  # { "e1": "2" }
```

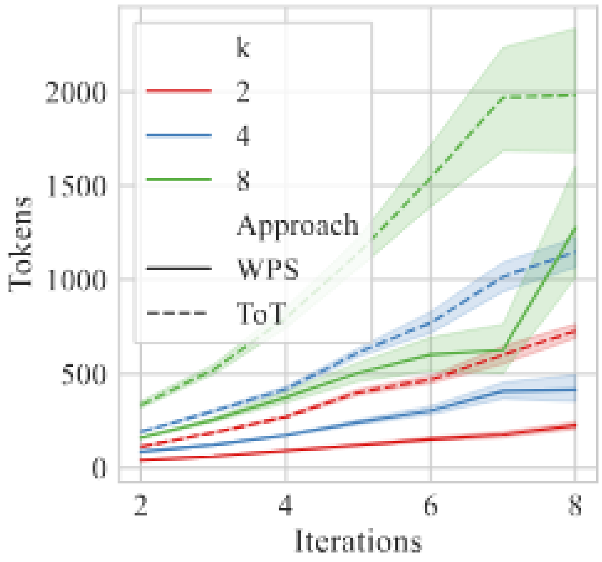
Line1 options

Line2 options

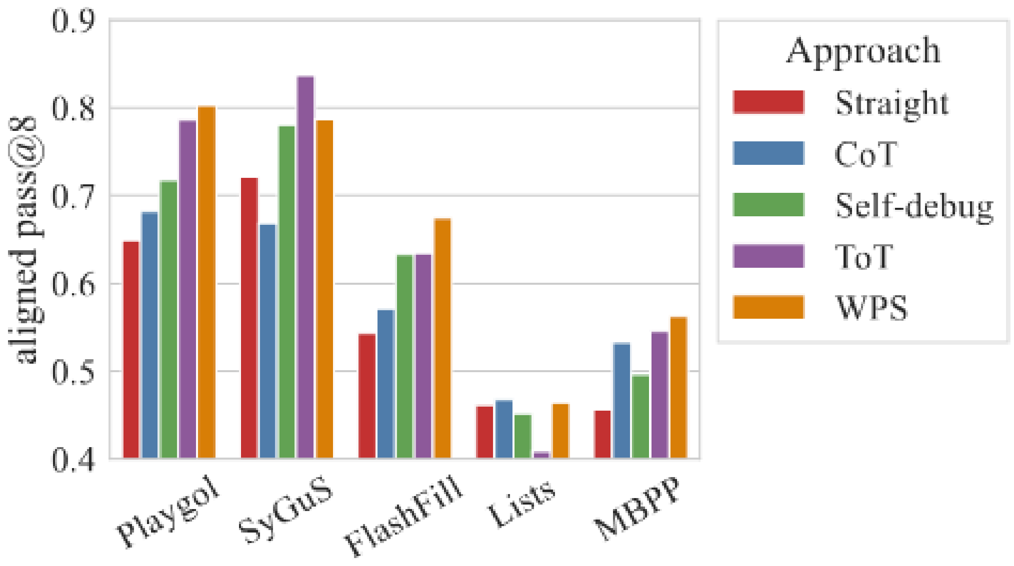
LLM decides:

- Which of v1-v5 should I use on Line3?
- Which of the values in blue comments can I use to generate '13'?

Results



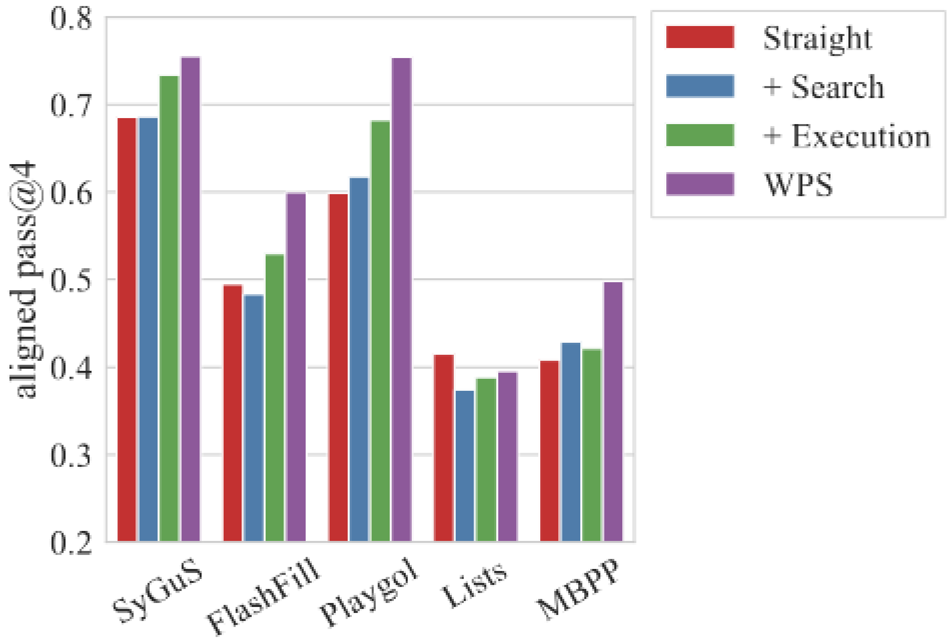
Main benefit of **within-prompt search (WPS)** is that it uses **less tokens** than outside-prompt search, such as **tree-of-thought (ToT)**

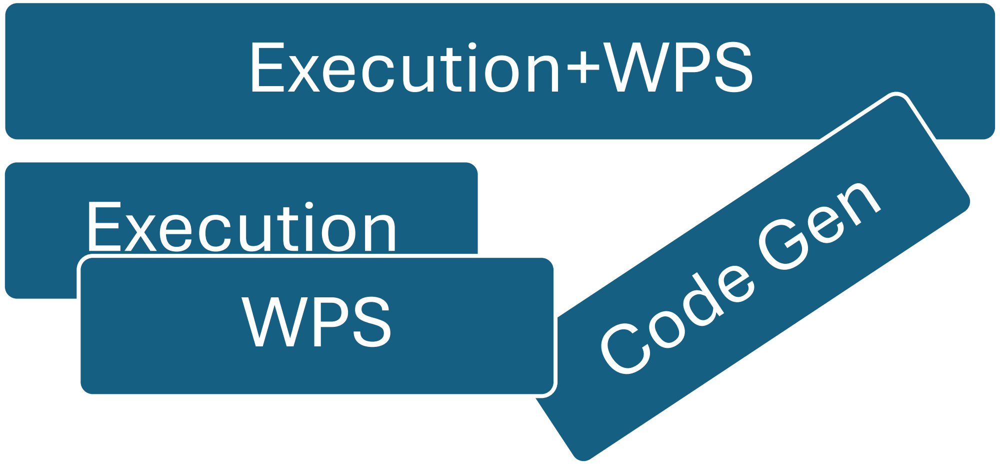


Aligned pass@k: normalize for token usage

- WPS performs best on 3/5 benchmark sets
- WPS is in top-2 in 5/5 benchmark sets

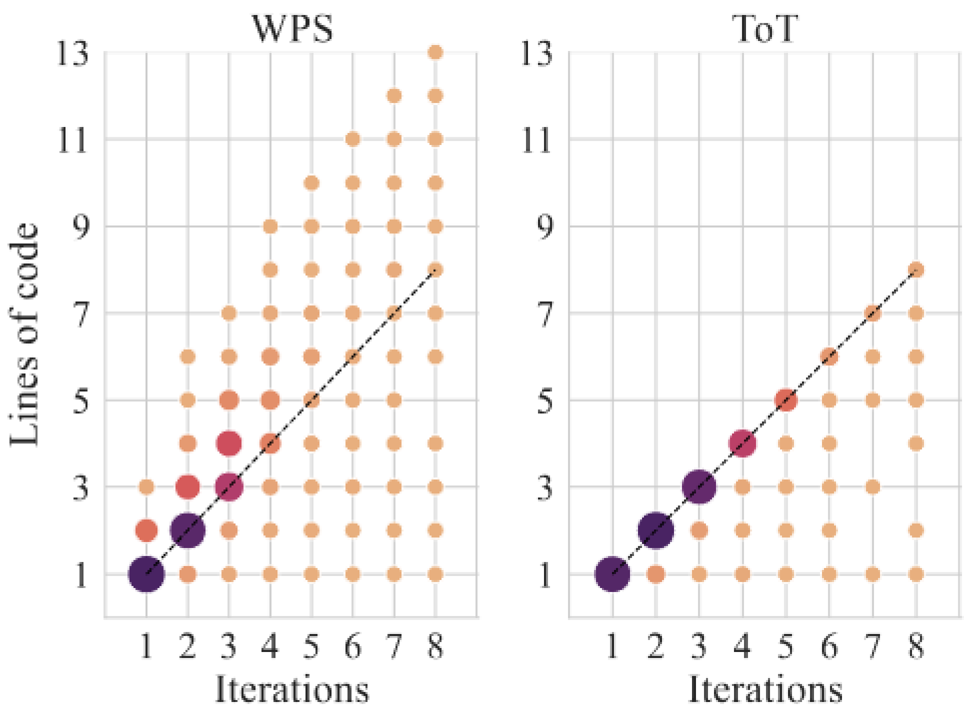
Execution and WPS are Both Important



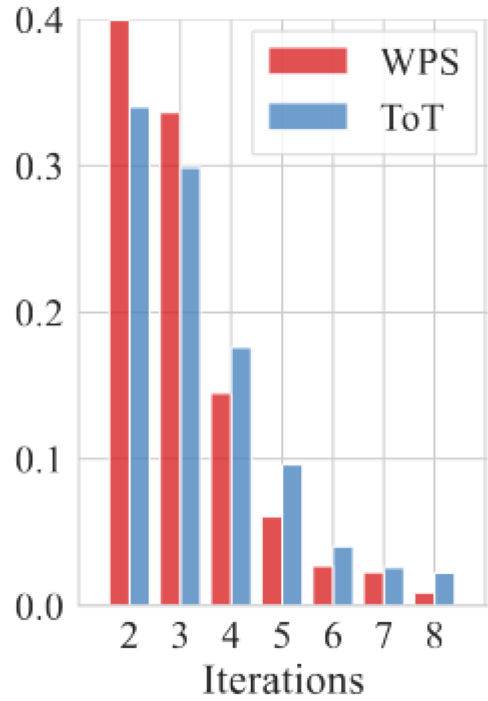


List benchmark: better to just generate full code in one go

WPS Backtracks and Forward Jumps!



LoC(generated program) > #iteration => forward jump
LoC(generated program) < #iteration => backtracked



%problem solved in different iterations

WPS uses fewer iterations

Conclusion

- Introduced **execution-guided within-prompt search**
- Allows LLM to function both as a
  - **policy**: generate multiple candidates
  - **value function**: pick most promising candidates to expand
- Is a way to **optimize tokens usage** when searching
- **Aligned pass@k** metric that normalizes for token usage
  - Our method does better on aligned pass@k
  - ToT does better on pass@1 ignoring token budget