Escape from Zurg: An Implementation in Python

Overview

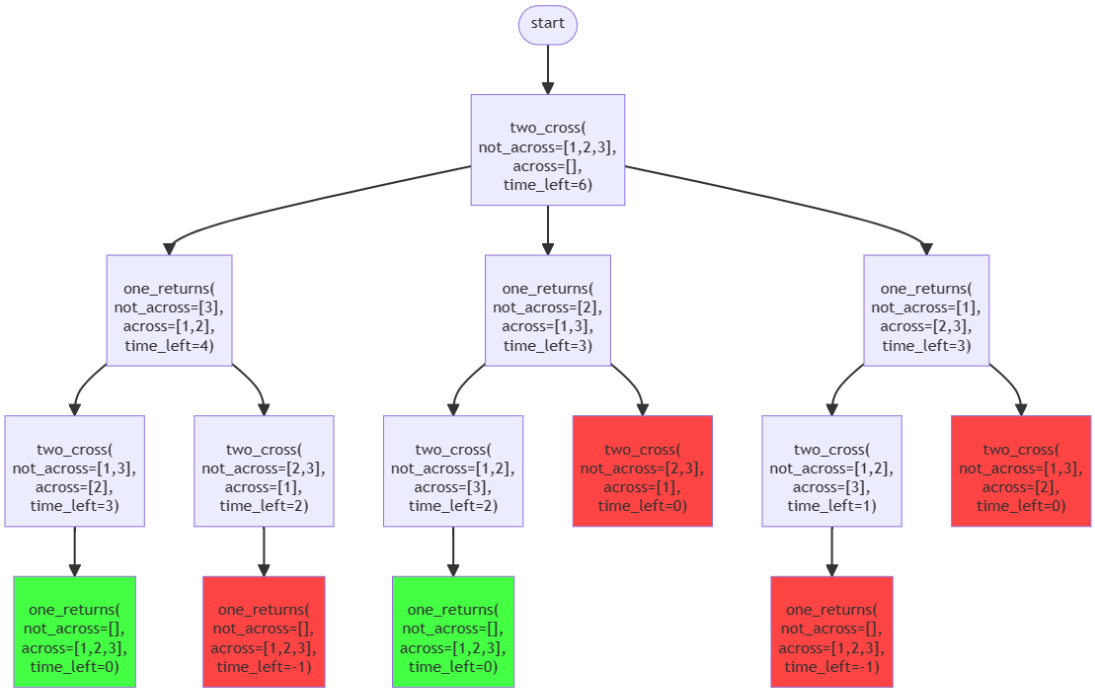
"Escape from Zurg" is a puzzle described in Escape from Zurg: An Exercise in Logic Programming in which the author compares two solutions, one in Prolog and one in Haskell, to see if functional programming languages are effective for search problems. Inspire by this, this project compares the author's Haskell solution to my own Python solution in terms of language features, readability, and performance. Python has a mix of imperative, functional, and OOP features that make for an interesting comparison.

Implementation

I met all the objectives laid out in my project proposal. I implemented the "Escape from Zurg" puzzle in Python and then used appropriate testing and profiling to optimize my solution for readability and performance. Finally, I compared my solution to the Haskell solution (see the "Conclusions" section at the end).

In `zurg.py`, I implemented a depth-first search using indirect recursion. `zurg()` defines the puzzle parameters and initiates the solution by calling `two_cross()`. `two_cross()` finds all the possible pairs that could cross and calls `one_returns()` for each. `one_returns()` calls `two_cross()` once for each toy that has crossed and could return. The recursion continues until each branch terminates, either because the "time limit" for the toys to cross the bridge has expired or because a solution has been found.

My solution is generalized and is not confined to the exact "Escape from Zurg" puzzle parameters. Calling `two_cross()` directly, instead of calling `zurg()`, allows you to input custom search parameters. For an example of this, see `TestTwoCross::test_recursive_case_with_time_limit` in `zurg_test.py`. Below is a chart that illustrates this test case visually. Red nodes represent time limit terminations while green nodes represent solutions found.



Testing

To run the "Escape from Zurg" solution, call `make run`. To run the full test suite, run `make test`. To run the tests follow by the solution, run `make`.

Dependencies

- `make`
- `python` >= 3.7 (for `pytest`)
- `pytest`

Within `zurg_test.py` there are system, integration, and unit tests. The `TestZurg` class contains system tests that ensure the solution matches the expected output, derived from `zurg.hs`, a copy of ZurgDirect.hs. These are black box tests, as they do not require knowledge of the implementation, just knowledge of the output format.

The `TestTwoCross` and `TestOneReturns` classes contain a mixture of integration and unit tests, where integration tests test the interplay between multiple user defined functions (in this case, `two_cross()` and `one_returns()`), and unit tests test a single user defined function in isolation. Due to the indirectly recursive nature of both `two_cross()` and `one_returns()`, both integration and unit tests make one function call. The distinction is in the behavior triggered by the input parameters. All the `recursive_case` tests are integration tests. Most of the `termination_case` tests are unit tests, with the exception of `TestTwoCross::test_termination_case_solution_found`.

The tests in `TestTwoCross` and `TestOneReturns` are gray box tests. They only need knowledge of the output format for the test cases but need implementation knowledge to distinguish between integration and unit tests.

Listing

Files

- `docs/`
  - `example.mer`: Mermaid.js source for example.png.
  - `example.png`: A chart showing program flow on a simplified puzzle.
  - `ZURG.md`: Docs for `zurg.py`.
- `tests/`
  - `zurg_test.py`: The test suite for `zurg.py`.
  - `ZurgDirect.hs`: Puzzle specific Haskell solution from here, used as the ground truth for testing.
- `Makefile`: Provides `make run` to solve the puzzle, `make test` to run the test suite, and `make docs` to generate Markdown docs from the `zurg.py` docstrings.
- `README.md`: This file.
- `README.pdf`: A PDF generated from `README.md` using the Markdown PDF VS Code extension.

Functions (`zurg.py`)

```
def zurg() -> List[list]
```

Solves the "Escape from Zurg" puzzle.

Returns:

`List[list]`: A list of solutions

```
def two_cross(not_across: List[int], across: List[int], time_left: int,
              accumulator: list) -> List[list]
```

Splits the search problem by every pair of toys that could cross the bridge. Terminates early if the time limit for getting all the toys across the bridge is met.

Arguments:

- `not_across` (`List[int]`): Toys not yet across the bridge
- `across` (`List[int]`): Toys across the bridge
- `time_left` (`int`): Time left to get all the toys across the bridge
- `accumulator` (`list`): Partial solution being explored

Returns:

`List[list]`: A list of solutions

```
def one_returns(not_across: List[int], across: List[int], time_left: int,
                accumulator: list) -> List[list]
```

Splits the search problem by every toy that could return across the bridge with the flashlight. Terminates early if the time limit for getting all the toys across the bridge is exceeded. Also terminates if all the toys are across the bridge.

Arguments:

- `not_across` (`List[int]`): Toys not yet across the bridge
- `across` (`List[int]`): Toys across the bridge
- `time_left` (`int`): Time left to get all the toys across the bridge
- `accumulator` (`list`): Partial solution being explored

Returns:

`List[list]`: A list of solutions

Conclusions