Syntian

Syn'tian :

- Syn means syntax and synthesis;
- Syntian sounds like 'san tian', which means the project is finished in 3 days...

We have to acknowledge that we fail to implement a general solver for all the testbenches. We combine multiple case-specific solvers to pass the open cases, and we feel sorry about that. (We hope that we could get some score for our efforts. >_<)

Features

Top-down search

We extend the given top-down search from TA with the following features:

- BFS with a priority queue. We use the function `numOps * 4 + numNonTermial * 2 + depth` for priority evaluation.
  - See `src/bfsqueue.py` for implementation.
- Preprocessing. Remove the excess rules from the production sets. For example, remove ">=" rule if there exists "<=" for the same non-terminal symbol.
  - See `src/preprocess.py` for implementation.
- Equivalence reduction. We adopt a hash strategy and remove the equivalent candidates. In general, we flatten associative operations and sort the operands of a commutative operation. For examples, `(+ b (+ a c))` is flattened to `(+ b a c)`, and then sorted to `(+ a b c)` according to the hash values `h(a)<h(b)<h(c)`. The hash strategy works in a recursive manner. Only candidates with a hash value which has never appeared would be inserted to the queue.
  - See `src/optimize.py` for implementation.
- Strategy check. During the hash evaluation, the strategy is abandoned if it contains any invalid expressions, such as `ite cond x x`. The check procedure depends on our hash method.
  - See `src/optimize.py` for implementation.

The entry of the top-down search is in `src/search.py`. This method passs `max2.sl, three.sl, tutorial.sl` for less than 1s (`tutorial.sl` for 0.910137s).

Pattern-based search

We find it really hard to optimize our search algorithm to get good performance for `maxk.sl` and `array_search_k.sl` cases. Their solutions need much `ite cond then else` statements, which result in a pretty large search space. Thus we manually design a generation "pattern" for list enumeration cases. The idea origins from the functional programming (, we consider functions as our search objects), and domain-specific languages (DSL, such as Halide, Chisel, and etc.) (, we design a pattern-based generation method).

To be specific, we design a function `generate`, which takes a tuple `(ls, args, cond, f_then, f_else, bound)`, and generates a solution:

- `ls` represents the list, such as `[x2, x3, x4, x5]` for `max5.sl` and `[(x1, 0), (x2, 1), (x3, 2)]` for `array_search_3.sl`;
- `args` represents the initial arguments for the enumeration, such as `(x1)` for `max5.sl` and `(k1, 4)` for `array_search_3.sl`;
- `cond` represents the condition generator for the `ite` of each enumeration. It consists of operations and elements in `head(ls)` (which may be a tuple) and arguments. For example, the `cond` is `lambda hd, args: ['<=', hd, args[0]]` for `maxk.sl` cases.
- `f_then` and `f_else` determines the generator for the then/else of `ite`. They have the format `(0/1, func)`. `(0, func)` update the arguments with `func` and call the `generate` with `tail(ls)` (e.g., `ls[1:]`) and the updated arguments. `(1, func)` generates a "value" with `func` based on `head(ls)` (e.g., `ls[0]`) and arguments. For the `array_search_k.sl` cases, the `f_then` is `(1, lambda hd, args: hd[1])`, which returns the answer index. And for `maxk.sl` cases, the `f_else` is `(0, lambda hd, args: [hd])`, which updates the argument, the maximum.
- `bound` represents the generator for bound situations, which generates a "value" based on the arguments.

The key idea of the "pattern" is that it applied the same strategy for each element of the list. The strategy is described by the tuple. See `src/patten.py` for implementation. With the `pattern`, we greatly reduce the search space to the determination of the tuple. Our method supports flexible settings for the tuple's exploration space. And for case-specific good performance, we use two sets of `(ls, args)` and set a small space for the left four functions. This method passes `maxk.sl, array_search_k.sl` cases.

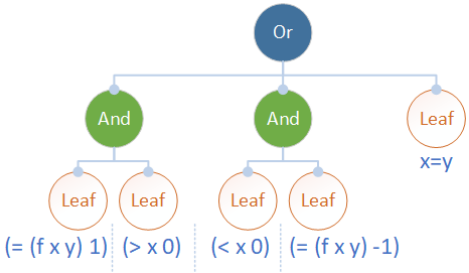
Constraint-guided Construction

For `s1.sl, s2.sl, s3.sl` cases, we observe that they have a common property:

- `f(...)` only appears in `(= f(A) B)` style, and `f` doesn't exist in `A, B`.

The property seems to imply that the constraints directly determine the value of `f(...)` under some conditions. We define the statement `(= f(A) B)` as an assignment. Therefore, we could extract conditions for each assignment, and merge them into one function by `ite` expressions. The algorithm contains steps (described by examples):

- Convert `(= f(1) (...))` to `(or (not (= x 1)) (= f(x) ...))`;
- Flatten `(and A (and B C))` to `(and A B C)`. Build a tree, where `and` layers and `or` layers alternate, as shown in the figure.
- Extract conditions for each assignment. Traverse from the root to the leaf, and add (`not`ed) branches which include no assignments into the conditions when meeting `and` (or `or`). The condition for the assignment `(= f(x, y) 1)` in the figure is `(and (> x 0) (not (= x y)))`.
- Merge the assignments with conditions into one function by `ite`s. The function for the figure is `(ite (and (> x 0) (not (= x y))) 1 -1)`.



Our constructed function is required to meet the original constraints, which has not been proved yet.

Shortcomings

Our implementation is weak for BV cases due to the limited depth of programs that BFS can generated. 3 days are too short for a more effective implementation that can generalize well.

Usage

```
python test.py
```

Noted that we slightly change the `test.py` to fit our file structure, the functionality keeps unchanged.