Stack operation optimization for Michelson

Sato Sota (DaiLambda, Inc.)

4/18 @ lab8 lunch talk

Background

- Michelson
  - for Tezos smart-contract development
  - Stack based
    - c.f.) Forth / Java bytecode / OCaml bytecode
  - Statically typed
  - Hard to write by hand
    - Various high-level languages & compilers are developed

Michelson code example

- https://smartpy.io/ide
  - Online IDE for Python → Michelson compiler

Michelson in Real World

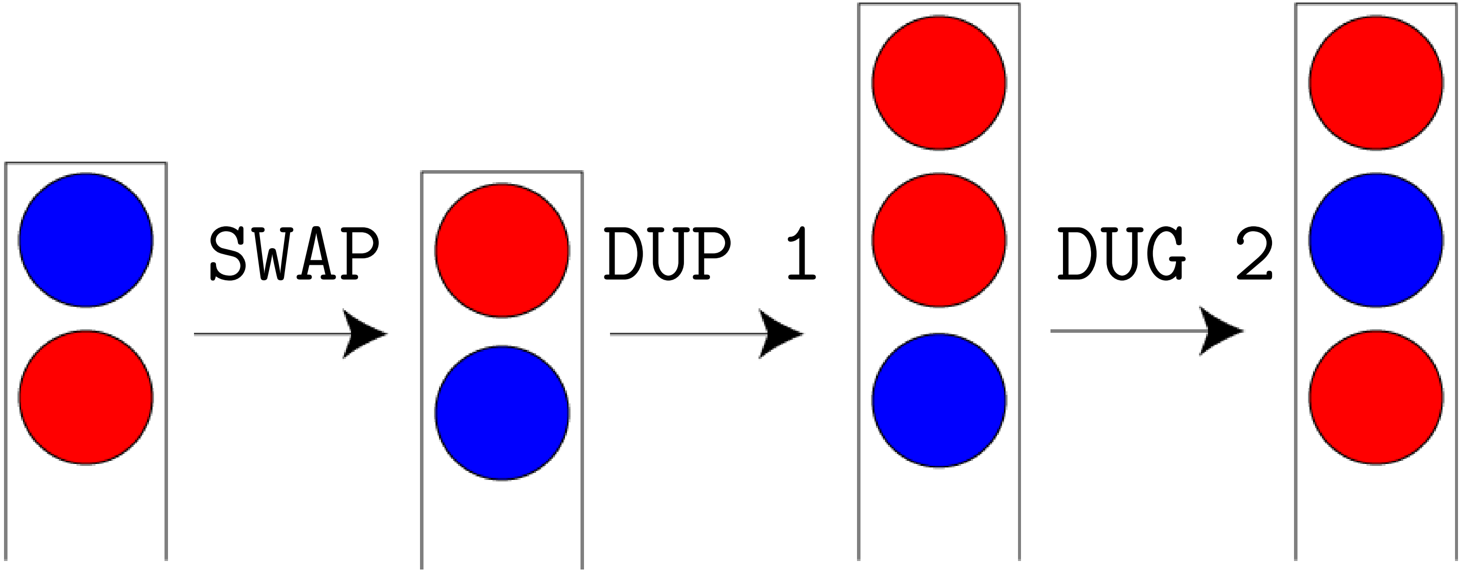
- Program costs "storage burn" in proportion to its size
  - 1byte ~ 0.001$ ~ 0.1 yen
- Unoptimized contracts are deployed
  - Compiler emits unoptimized code
  - Optz (our optimizer) reduces avg. 5% of size

Optz

- https://dailambda.jp/optz/
  - Online editor: https://dailambda.jp/optz-js/
- Optz optimizes Michelson code in 3 ways
  1. pattern matching
     - { SWAP; LT; } → { GT; }
     - { DROP n; DROP m; } → { DROP (n+m); }
  2. Exhaustive search on stack manip op seq
  3. Special case optimization

Optimization Example of Exhaustive search

- Before) SWAP; DUP 1; DUG 2



- After) DUP 2

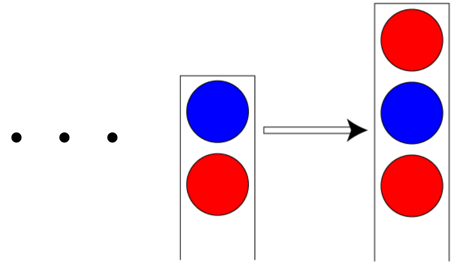
Target of exhaustive search

Sequence of stack manipulation operations

- Target
  - PUSH, DUP, DROP ... insert/delete elements
  - SWAP, DIG, DUG ... rearrange elements
- Non-target
  - ADD, MUL, CMP ... calculation
  - IF, LOOP ... control operators

Stack manipulation function

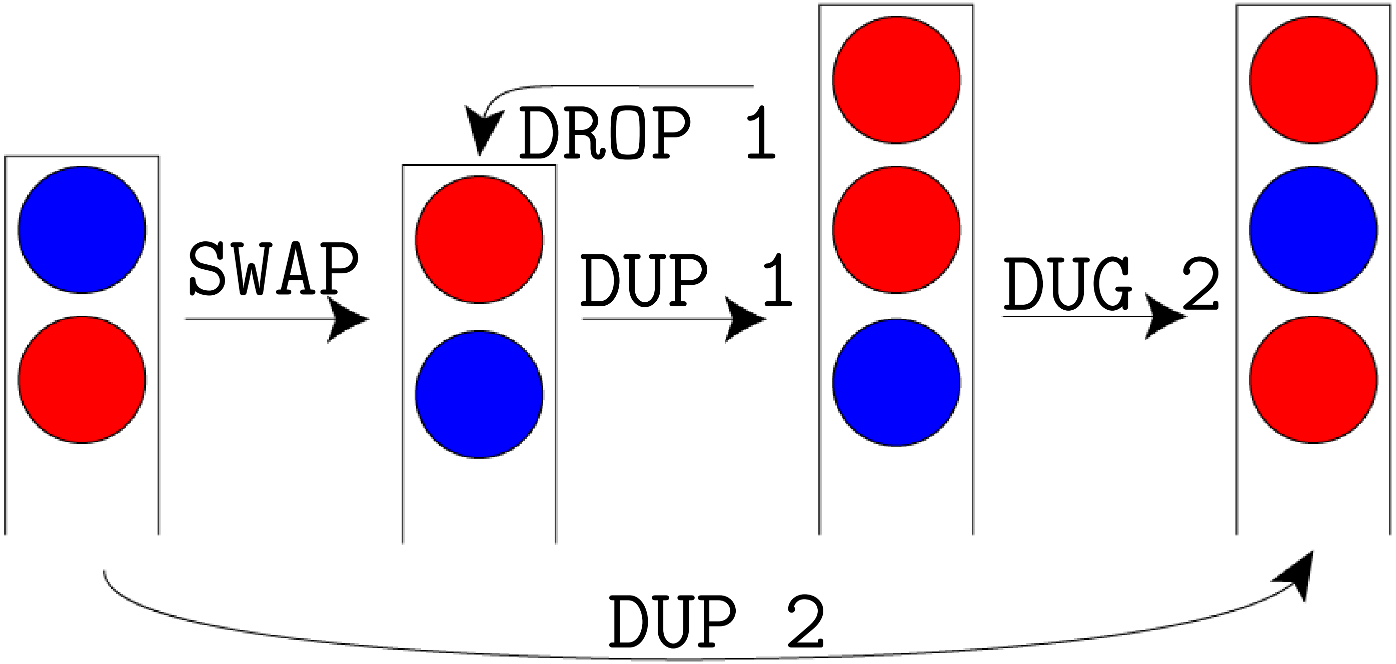
- Stack manipulation operation sequence represents stack → stack function
  - Symbolic execution result of stack
  - e.g.) { SWAP; DUP 1; DUG 2; }
    { DUP 2; }



- Find cheapest stack op seq which represents same function

Exaustive Optimization using Dijkstra search

- Find best op with Dijkstra search
  - Cost of edge is `op size + op exec cost`



Computational Complexity of exhaustive search

- $O(N^2 \log N)$ where N is "the num of nodes in graph"
- N is upper bounded by
  `L := num_of_variable_variations ^ max_stack_length`
- We use `L` to prevent time-consuming optimization.
- Empirically, we set `L` to 10000.0
  - Every sampled contracts is optimized within 1[s].

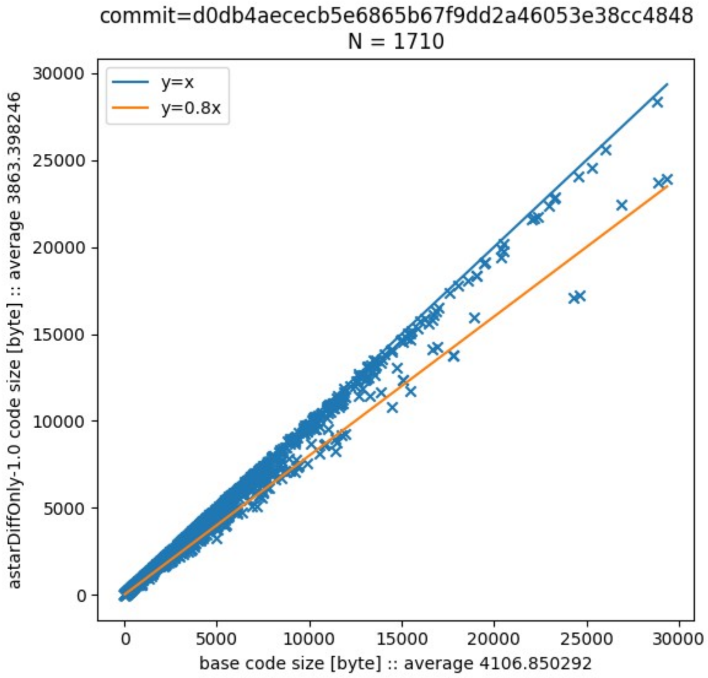
Special Case Optimization

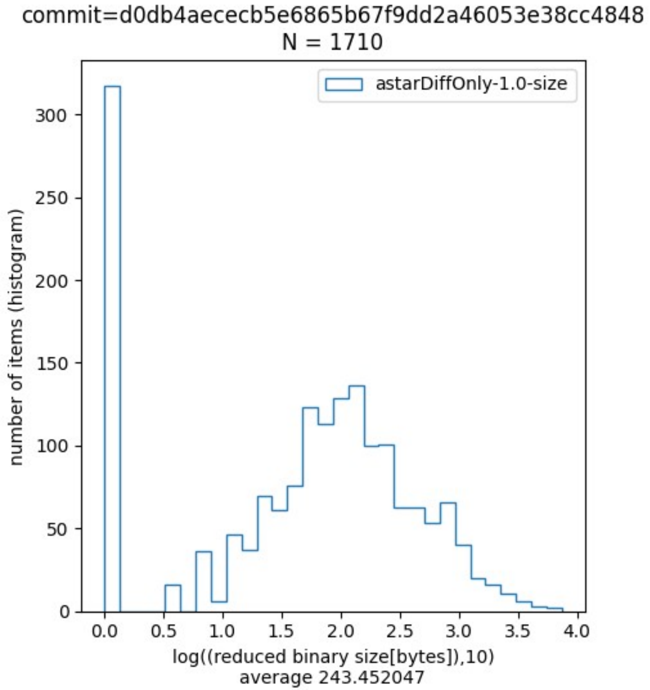
- Drop-only op seq
  - Result stack is sub-sequence of start stack
  - e.g.) { SWAP; DROP; SWAP; DROP; … }
  - L is too big to optimize with exhaustive search. Instead, we use ad-hoc optimization.
  - Such unoptimized drop-only seq is generated from stack cleaning in function epilogue.

Result of optimized code size

Contracts deployed in 2021-02-18 - 2022-02-17

Avg. size reduction is 5% / 243byte (~ 24 yen)





A* search for faster optimization

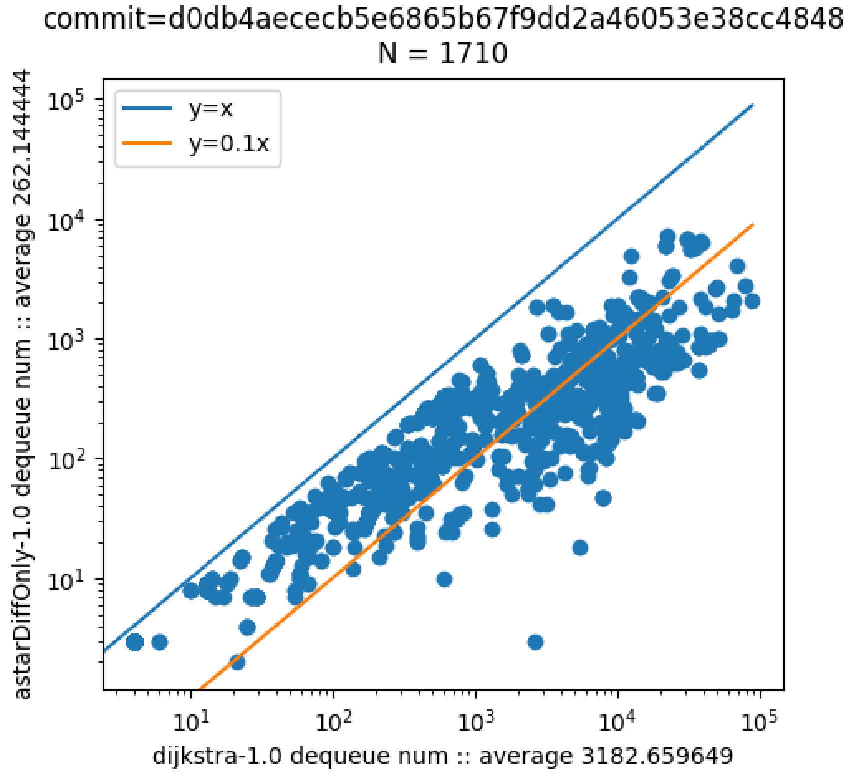
- Dijkstra search
  - Search from the node whose cost(node) is the smallest.
- A* search
  - Estimate score(node) which satisfies
    0 <= score(node) <= actual_distance(node,goal)
    - e.g.) solving maze → score(node) := Manhattan distance to goal
  - Use cost(node) + score(node)
    instead of cost(node) for searching

Good score function for stack modification op seq

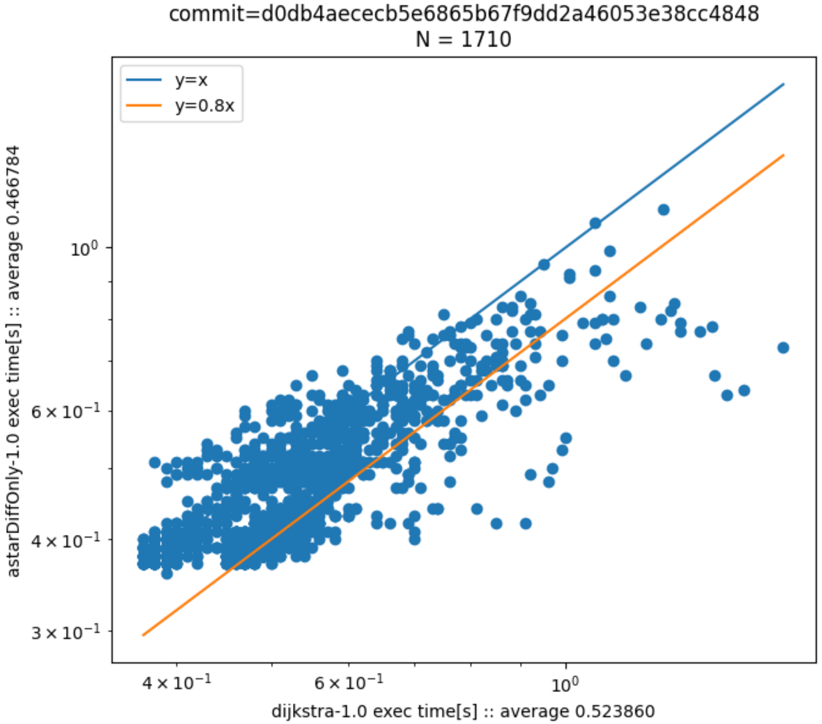
- Required number of PUSH operations
  - score(stack) :=
    size(set(goal) - set(stack)) * cost(PUSH)
  - The vars appear in goal,
    and not appear in current stack,
    should be pushed.

Result of Optimization Speed
A* v.s. Dijkstra

Reduced Searched Node :: 90%



Reduced Time :: 10%



Summary

- Optz uses 3 types of optimizations, pattern matching, exhaustive search, and drop-only sequence.
- Optz reduces avg. 5% size of code.
- A* reduces 10% of exhaustive search time.

Appendix: Materials / References

- https://dailambda.jp/optz/
  - Blog post for optz
- https://dailambda.jp/optz-js/
  - Web interface for optz
- https://gitlab.com/dailambda/scaml/-/blob/master/src/michelson/optimize.ml
  - Source code of the optimizer part of optz
- https://medium.com/hackernoon/optimizing-stack-manipulation-in-michelson-31ba7ff11a3a
  - Source of the idea for using *A

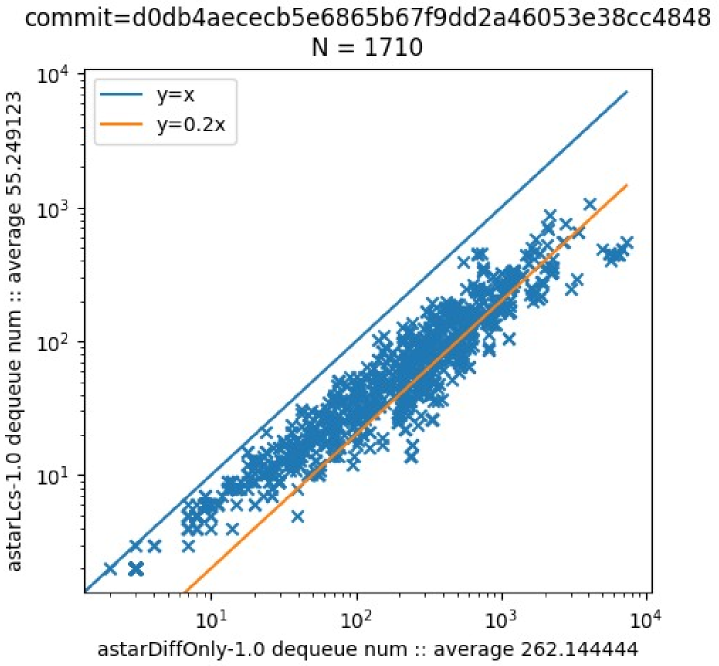
Appendix: score function with LCS

- Estimate with LCS
  - Good op sequence will preserve stack ordering
  - score’(st) :=

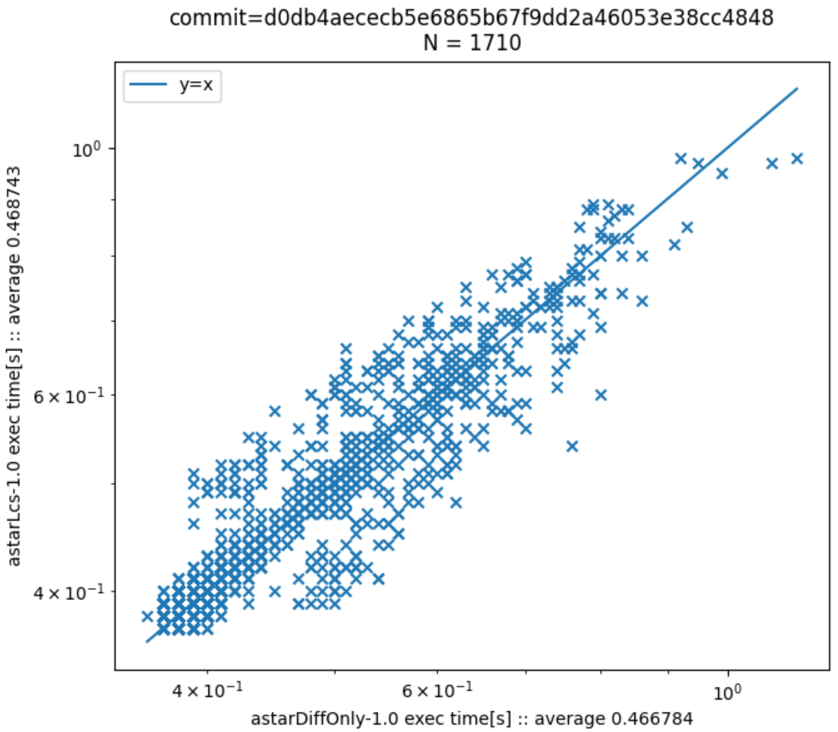
```
let l = LCS(st,goal) in
let push = |set(goal)-set(st)| in
(len(st)-l) * cost(DROP) +
(len(goal)-push-l) * cost(DUP) +
push * cost(PUSH)
```

Appendix: LCS score v.s. PUSH diff only score

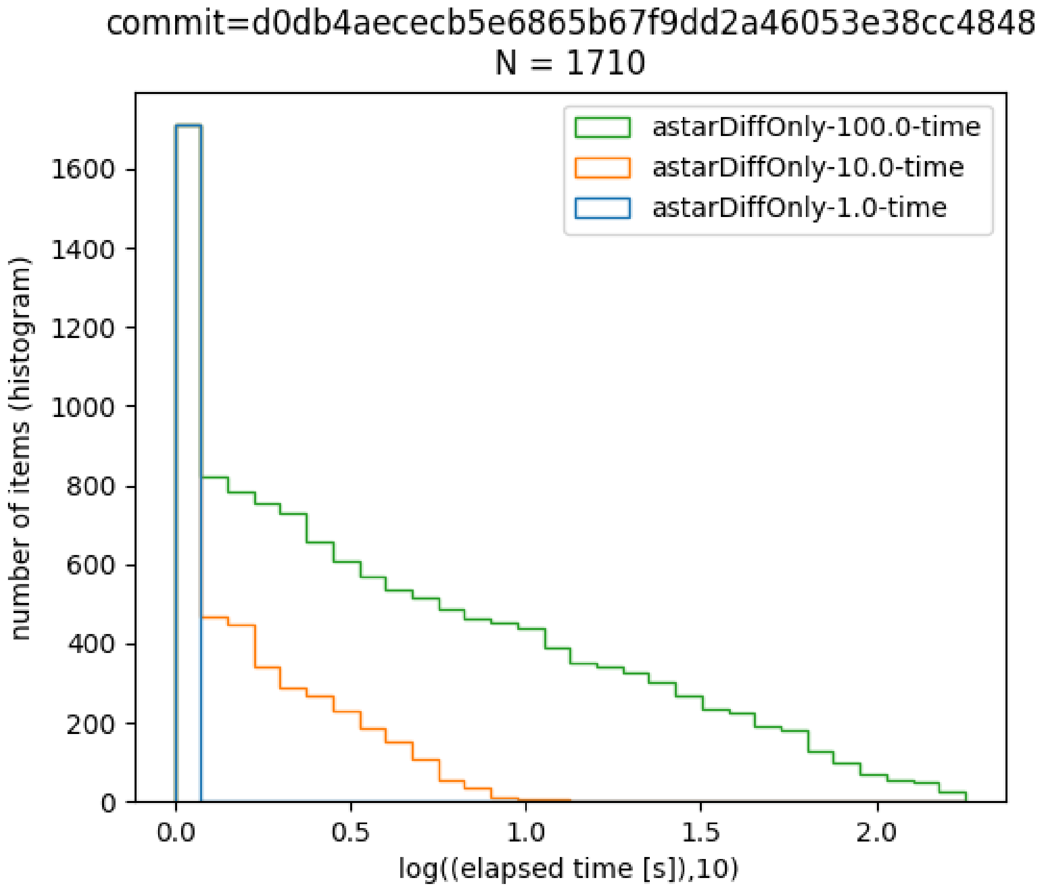
Reduced Searched Node :: 80%



Reduced Time :: 0%



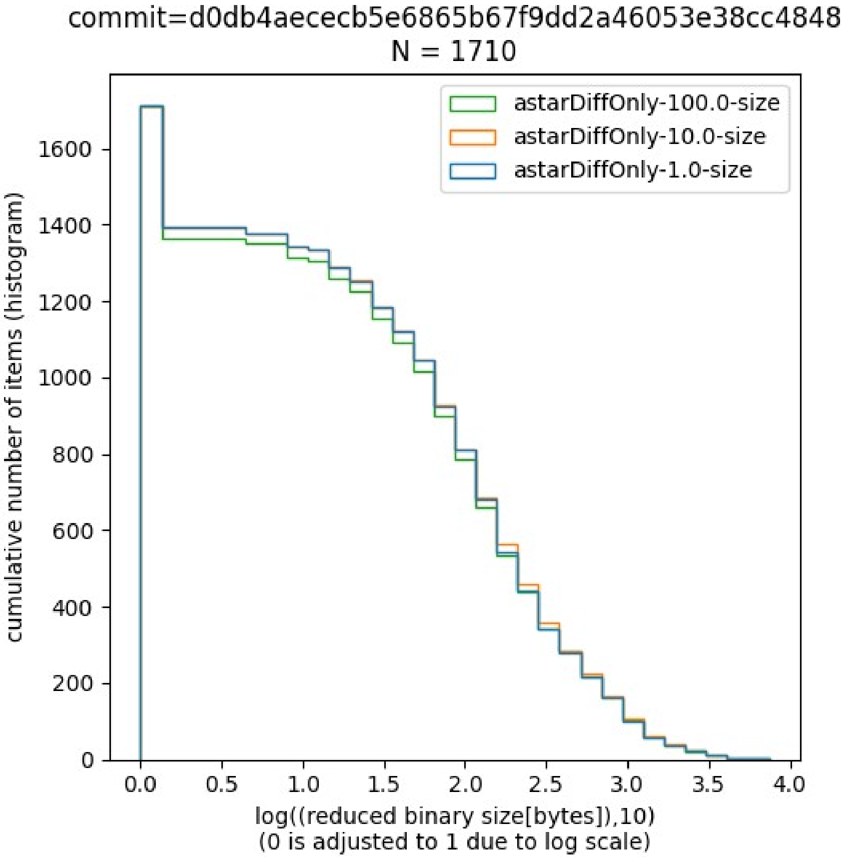
Appendix:
Best parameter for stack limit L



Tested on L := 10000.0,
100000.0,
1000000.0

Execution time seems to be proportional to L.

Appendix: Best parameter for stack limit L



Increasing L
scarcely reduces code size.

i.e.

Speeding up optimization is
not so urgent.

Appendix: Future work

- Extend target ops of exhaustive search
  - Data type constructing / deconstructing ops
    PAIR / CAR / CDR / CONS
  - Calculation ops
    ADD / SUB / MUL / DIV / LE
- Faster algorithm for exhaustive search