NeurIPS 2025 – Google Code Golf Championship

5th-place solution: Compressed Code Golf Techniques

Team: HIMAGINE THE FUTURE.

Hiroki Shibata / shibh308 (Kyushu Univ.)

slides

our writeup

About our team

	 shibh308	 keymoon	 Yu	 kq5y
Background Kaggle / ML				
Background CTF	✓	✓	✓	✓
Background Problem Solving	✓	✓	✓	
What we did Code Golf		✓	✓	✓
What we did Compressed Code Golf	✓			

Compressed Code Golf: Concept

zlib performs best for most codes

- Standard compressors (zlib/lzma/bzip) can be utilized within the code
- For long code, we can reduce the code length through compression
  - Hard-coding the compressed binary data
  - Decompressing and executing at run-time

→ Producing shorter compressed binary makes shorter solution!

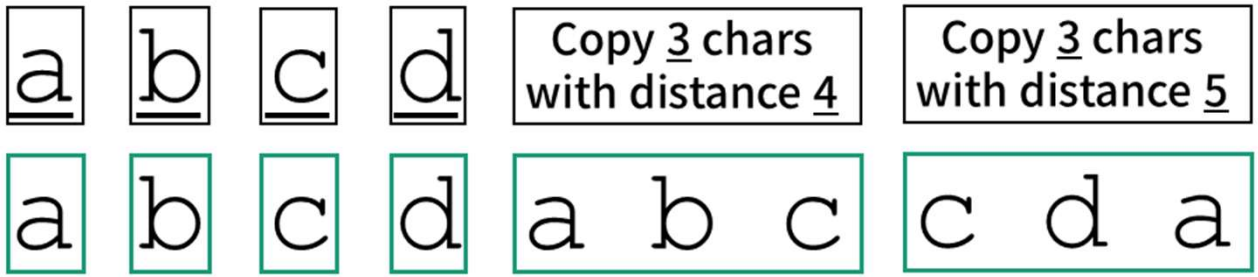
```
#coding:L1
import zlib
exec(zlib.decompress(bytes("<compressed code>",'L1'),-9))
```

Deflate file format (1)

zlib produces gzip-compatible binary

Deflate: the data format used in gzip

- **Parse the text with two types of factors: "single literals" or "copies of previous substrings"**
- **Represent each factor with [literal] or [length,distance]**
  - **In the example, encoded representation is [a][b][c][d][3,4][3,5]**

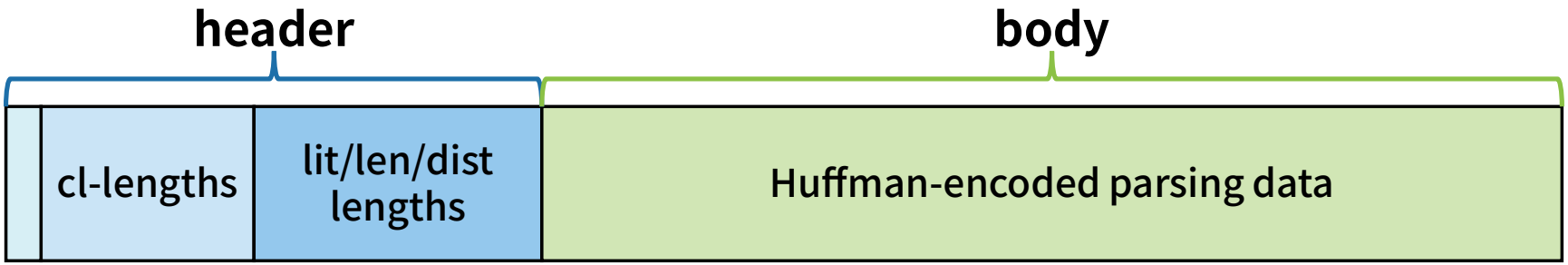


Deflate file format (2)

Note: We assume that the binary consists of a single dynamic Huffman block based on practical observation.

Deflate: the data format used in gzip

- Deflate data consists of two parts
  - **Header** mainly stores code length data for the Huffman codes (30-50 bytes)
  - **Body** stores factor data in Huffman-encoded form (100-200 bytes)
- Typically, existing compressors focus on reducing the body size



Our optimization strategy

We can outperform existing compression methods (zlib, zopfli) by:

1. Rewriting the original program before compression
2. Optimization focusing on the internal structure of deflate binary
3. Combining each optimization method with metaheuristics

Key Optimization 1: Reassign variable names

We can edit the input code as long as the output result remains unchanged

- Reassign variable names to satisfy these two objectives:
  - Skew character frequencies to shorten Huffman code length
  - Use adjacent characters in ASCII order (to facilitate run-length encoding)
    - because the Huffman code lengths for each literal are stored in run-length encoded form with ASCII order
- Implement static analysis to detect variables used in input Python code and their dependencies (written by Codex☺)

Key Optimization 1: Reassign variable names

```python
def p(g):
 u=[[*s]for s in g]
 for _ in range(80):
  u=[[*s]for s in zip(*u[("2, "*7in str(u[-1]))-2::-1])]
 for _ in range(8):
  for y in range(len(u)-2):
   for x in range(len(u[0])-2):
    if(u+[[1]*80]*3)[y-1][x+1]*(u+[[1]*80]*3)[y+3][x+1]:
     for i in range(len(g)-2):
      for j in range(len(g[0])-2):
       if all([2==g[i+k][j+m],2!=g[i+k][j+m]|2][u[y+k][x+m]>0]for k in range(3
         )for m in range(3)):
        for k in range(3):
         for m in range(3):
          u[y+k][x+m]=g[i+k][j+m]
          g[i+k][j+m]=0
 u=[[*s]for s in zip(*u[::-1])]
 return u
```

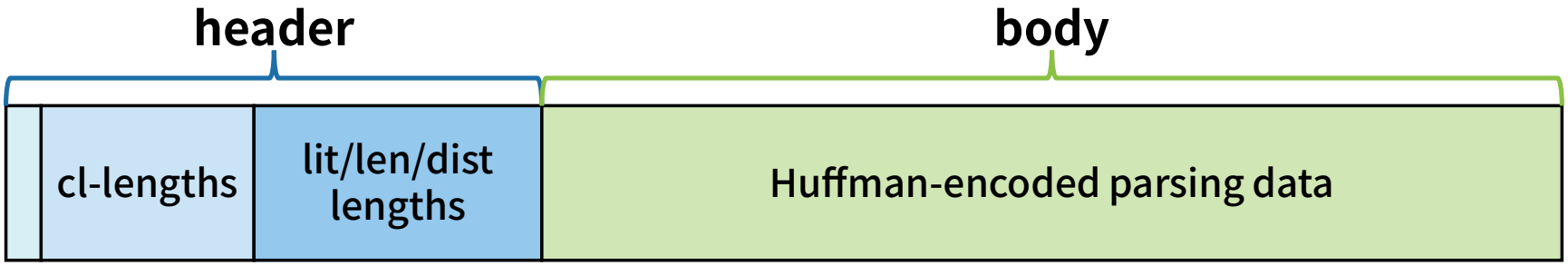
Before: 225 bytes (by zopfli)

```python
def p(r):
 n=[[*i]for i in r]
 for o in range(80):
  n=[[*i]for i in zip(*n[("2, "*7in str(n[-1]))-2::-1])]
 for o in range(8):
  for f in range(len(n)-2):
   for u in range(len(n[0])-2):
    if(n+[[1]*80]*3)[f-1][u+1]*(n+[[1]*80]*3)[f+3][u+1]:
     for g in range(len(r)-2):
      for s in range(len(r[0])-2):
       if all([2==r[g+i][s+e],2!=r[g+i][s+e]|2][n[f+i][u+e]>0]for i in range(3
         )for e in range(3)):
        for i in range(3):
         for e in range(3):
          n[f+i][u+e]=r[g+i][s+e]
          r[g+i][s+e]=0
 n=[[*i]for i in zip(*n[::-1])]
 return n
```

After: 221 bytes (by zopfli)

Key Optimization 2: Internal Structure Optimization

- Once the input text is fixed, the binary is determined by these parts:
  - cl-lengths
  - Literal/length/distance Huffman code lengths (in run-length encoded form)
  - Parsing of the text



Breakthrough: we can compute optimal lengths/parsing by dynamic programming when we fix other parts!

Key Optimization 3: Metaheuristics Approach

- Our algorithms optimize each part of the text/compressed binary
- We use the genetic algorithm (GA) to combine these subroutines
  - Mutation: Adapting each optimization subroutine
  - Perturbation: changing code lengths and variable assignment
  - Crossover: Taking cl-code lengths from another solution
- Why GA? Not local search?
  - Designing large neighborhood operations of local search is difficult☹
  - Crossover operation makes large “jump” in the solution space

Final Results

- Our method reduces the compressed file size by about 5-10%
- Our method improves the score by about 700 points compared to zlib, and about 500 points compared to zopfli across about 50 tasks
- Adapting our method to all-team-best solutions reduces 98 bytes across 21 tasks, even compared to the 1st-place compressed binary optimization method

Thank you for organizing the contest!

slides

our writeup