Team Presentation

Crazy Thursday

2022-11-25

Our team

Liu, Yunzhi (Jacob)

- First year CompSci student
- University of Cambridge

Hu, Zheyuan (Peter)

- First year CompSci student
- University of Cambridge

Song, Boyao (Bobby)

- First year CompSci student
- University of Edinburgh

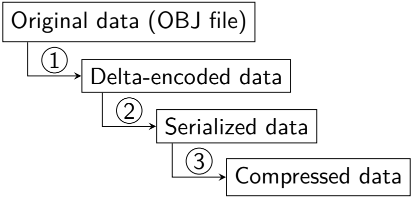
Jie, Haoran (Samuel)

- First year CompSci student
- University of Cambridge

The structure of our algorithm

1. The first step does preprocessing to remove redundant data.
2. The second step serializes the data.
3. The third step compresses the serialized data.
   Several candidate compression algorithms:
   - Huffman
   - LZSS

Decoding is simply reversing the steps.



Therefore, our presentation will be split into 4 parts:

- Preprocessing & Serialization
- Compression algorithms:
  - Huffman coding
  - LZSS coding
  - Optimizations to LZSS

The OBJ file format

The basic building blocks:

- `v <x> <y> <z>` specifies vertex coordinates;
- `vn <x> <y> <z>` specifies vertex normals;
- `vt <u> <v>` specifies texture mappings.
- `f <fv_1> <fv_2> <fv_3>`, where `<fv_N>` is in the form `#v/#vt/#vn`.

Preprocessing & Serialization

Removing duplicates, Delta coding and Serialization

Yunzhi (Jacob) Liu
(Team Crazy Thursday)

2022-11-25

Duplication

Upon inspection, the dataset contains many duplicated `v`, `vt`, `vn` entries. E.g.,

`vn` duplication

Very often the data would contain the same `vn` values in multiple rows.
Those `vn` entries are superfluous — we can delete them and relabel `f` commands accordingly.

This incurs no cost during decompression!

Delta Coding

Notice that: For each one value, the sequence it forms is roughly increasing.

- Store difference (delta) with previous term;
- Store marker and original data for larger jumps.

Todo

Maybe adopt a variable-length encoding similar to UTF-8?

Serialization

In an OBJ file, data is stored as ASCII text, yet all data can be expressed using numbers.

Use integers instead of floating-point numbers

All numbers have at most 6 decimal places $\Rightarrow$ Use fixed-point numbers.
We can multiply all numbers by $1\,000\,000$ and store their integer values in an `int32`.

Some data entries (namely `vn`) can be more tightly serialized by observing some range constraints.

Results

When the above algorithm is run without any further compression, the program achieved a score of 63.77.

These steps save a lot of space whilst incurring very little cost in performance.

Huffman Encoding

Boyao Song[1]

[1]School of Informatics
University of Edinburgh

November 19, 2022

Huffman encoding in brief

- Record the weight of each symbol
- Build Huffman tree based on weight result
- Produce a final file that contains both the `Dictionary` and `Literal`

Calculate Weight

Calculate weight w_i of each symbol a_i in the input code.

For example

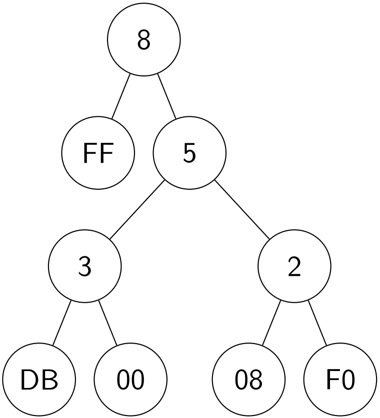
DB	00	00	08	F0	FF	FF	FF

a_i	w_i
00	2
FF	3
DB	1
08	1
F0	1

Build tree

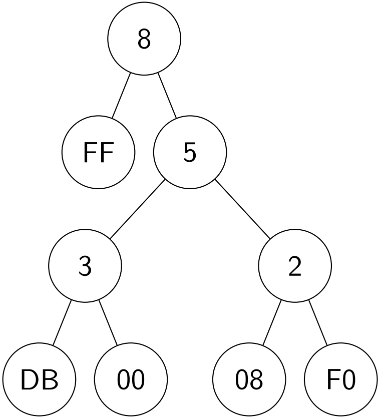
The symbol with the lowest weight goes to the bottom.

a_i	w_i
00	2
FF	3
DB	1
08	1
F0	1



Build tree

The symbol with the lowest weight goes to the bottom.



We can then get the encoded table from the tree

a_i	c_i
FF	0
DB	100
00	101
08	110
F0	111

Encode the original information

DB	00	00	08	F0	FF	FF	FF

$\rightarrow$ 100 101 101 110 111 0 0 0

a_i	c_i
FF	0
DB	100
00	101
08	110
F0	111

Encode the original information

In this case, we decreased the code length from 256 to 18, but do not forget to store the `Dictionary`!
Which has a length of $(32+3) \times 5 = 175$ in this case.

So, in total, we decreased the code length from 256 to 193.

With Huffman encoding, we can compress serialized `2CylinderEngine.obj` from 4 MiB to 1.2 MiB

Negative side of huffman encoding

Decompression time for a larger file is HUGE. As when doing decompression, we use many pointer jumps when navigating the Huffman tree on Heap.

Because we need to store a dictionary, this method is also not applicable on files with a small size.

After investigation, we found that we get the best tradeoff between time and compression rate when the file size is between 1 MiB and 4 MiB.

Improving the current code

1. Change all STL code to pure C code
2. Smarter indentation in the Dictionary
3. Algorithm level improvements

Further improvements from baseline Huffman encoding

Canonical Huffman code

a_i	c_i
A	11
B	0
C	101
D	100

$\rightarrow$

a_i	c_i
B	0
A	10
C	110
D	111

This method can decrease the dictionary size by a huge as now we only need to record the bit-length of each encoded message

Current state of Huffman encoding in our codebase

We found out the using huffman encoding will produce a high but slightly lower result when combing with other compression method.

So we did not put Huffman encoding in our final result.

6-Level LZSS with improvements

Effective Compression and Efficient Decompression

Peter Hu, Samuel Jie

First Year Computer Science Student
University of Cambridge
Team: Crazy Thursday

2022-11-25

Table of Contents

Matching Implementation

```
bool Matched_First_Every3Bytes(uint windows_size){
    bool res=false;
    sequence = uint32(input) & 0xffffff;
    hash = hash_func(sequence);
    ref = input_start + Hash.table[hash];
    if((input - ref)<windows_size &&
        sequence==ref_seq){res=true;}
    Hash.table[hash] = input - input_start;
    return(res);
}
```

A Simple Demostration

Short Match

M_2–M_0 W_{12}–W_8 | W_7–W_0

I␣am␣Sam\n	08 I am Sam 10(\n)
Sam␣	20 03 00 20(␣)
I␣am	40 0C
\nThat	04 10(\n) That

Note:

20 03: 0010 0000 0000 0011

M_2–M_0	W_{12}–W_8	W_7–W_0
001	00000	00000011

A Simple Demostration

Literal Run

000 L_4-L_0 | ...

```
I␣am␣Sam\n
Sam␣
I␣am
\nThat
```

```
08 I am Sam 10(\n)
20 03 00 20(␣)
40 0C
04 10(\n) That
```

Note:

00 20 : 0000 0000 $(20)_{10}$

000	L_4–L_0	...
000	00000	$(20)_{10}$

Level 1 Fundamental of LZSS Algorithm

Supported Window Size

Level 1: $2^{13} = 8192$, i.e., [0..8191].

Original Match	→	Encoded Len
1-2 Bytes	→	1 Bytes (Literal Run)
3-8 Bytes	→	2 Bytes (Short Match)
9-264 Bytes	→	3 Bytes (Long Match)

Match type	Literal run	Short match	Long match
Decode[0]	$000L_4-L_0$	$M_2M_1M_0W_{12}-W_8$	$111W_{12}-W_8$
Decode[1]		W_7-W_0	M_7-M_0
Decode[2]			W_7-W_0

Basic Idea of Implementation

Lots of Low Level Operations:

i. Using **pointers** of C without STL from C++.

ii. **Bitwise Operators**, dealing with Binary Numbers.

iii. **Hash Table**, storing sequences that appeared before.

iiii. **Fixed width integer types**, like uint8_t, uint32_t

To achieve Faster Decompression Speed

How We Choose Between Levels

Relationship between Original File and Encoded File

Original Match	→	Encoded Len
1-2 Bytes	→	1 Bytes (Literal Run)
3-8 Bytes	→	2 Bytes (Short Match)
9-264 Bytes	→	3 Bytes (Long Match)

By using the First 3 bits of the Literal Run at the beginning of the Input File as Flag:

```
level 1 000
level 2 100 *(ubyte*)OUTPUT_ |= (1 << 7);
level 3 010 *(ubyte*)OUTPUT_ |= (1 << 6);
level 4 001 *(ubyte*)OUTPUT_ |= (1 << 5);
```

Level 2 Extended Window Size with Infinite Match Length

Part 1 | Infinite Match Length

Level 2: $2^{13} = 8192$, i.e., $[0..(8191 - 1)]$
(11111 0xff)is used as flag for whether it's Extended Window.

Match type	Literal run	Short match	Long match
Decode[0]	$\mathtt{000L_4\text{-}L_0}$	$\mathtt{M_2M_1M_0W_{12}\text{-}W_8}$	$\mathtt{111W_{12}\text{-}W_8}$
Decode[1]		$\mathtt{W_7\text{-}W_0}$	$\mathtt{M_{n+7}\text{-}M_n}$
Decode[..]			$\mathtt{M_7\text{-}M_0}$
Decode[2]			$\mathtt{W_7\text{-}W_0}$

Level 2 Extended Window Size with Infinite Match Length

Part 2 | *Extended Window Size

Level 2: $2^{16} = 65536$, i.e., $[8191..8191 + (65535 - 1)]$.
(11111 0xff)is used as flag for End.

Match type	Literal run	Short match*	Long match*
Decode[0]	$000L_4$-L_0	$M_2M_1M_0$11111	111 11111
Decode[1]		11111111	M_{n+7}-M_n . . .
Decode[..]			M_7-M_0
Decode[2]		W_{15}-W_8	11111111
Decode[3]		W_7-W_0	W_{15}-W_8
Decode[4]			W_7-W_0

Level 3 Extra Windows Size

Extra Windows Size

Level 3: $2^{16} = 65536$, i.e., $[8191 + (65535 - 1)..8191 + (65535 - 1) + (65535 - 1)]$.
(11111 0xff)is used as flag for End.

Match type	Literal run	Short match*	Long match*
Decode[0]	$000L_4$-L_0	$M_2M_1M_0$11111	111 11111
Decode[1]		11111111	M_{n+7}-M_n...
Decode[2]		0*xff*	11111111
Decode[3]		0*xff*	0*xff*
Decode[4]		W_{15}-W_8	0*xff*
Decode[5]		W_7-W_0	W_{15}-W_8
Decode[6]			W_7-W_0

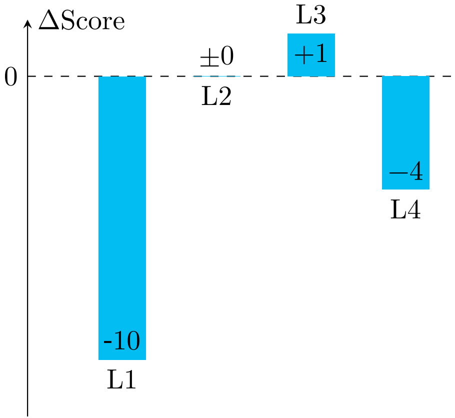
Level 4 Ultra Windows Size

Ultra Windows Size

Level 4: $2^{16} = 65536$, i.e., $[8191 + (n-1) * (65535-1)..8191 + n * (65535-1)]$.
(11111 0xff)is used as flag for End.

Match type	Literal run	Short match*	Long match*
Decode[0]	$000L_4$-L_0	$M_2M_1M_0$11111	111 11111
Decode[1]		11111111	M_{n+7}-M_n...
Decode[2]		0xff	11111111
Decode[3]		0xff...	0xff
Decode[4]		W_{15}-W_8	0xff...
Decode[5]		W_7-W_0	W_{15}-W_8
Decode[6]			W_7-W_0

2 Extended Levels of our LZSS Algorithm



Level 5 Direct Match

Direct Match

The data structure of Match Length is not efficient enough

Match type	Literal run	Short match	Long match
Decode[0]	$000L_4$–L_0	$M_2M_1M_0W_{12}$–W_8	$111W_{12}$–W_8
Decode[1]		W_7–W_0	...M_7–M_0
Decode[2]			W_7–W_0

Level 5 Direct Match

Direct Match

The data structure of Match Length is not efficient enough

Match type	Literal run	Short match	Long match
Decode[0]	$000L_4$-L_0	$M_2M_1M_0W_{12}$-W_8	$111W_{12}$-W_8
Decode[1]		W_7-W_0	...M_7-M_0
Decode[2]			W_7-W_0

Level 5 Direct Match

Direct Match

Level 5: A better solution to Match Length Data Structure
Reserve Flag 110 in Short Match for the new Direct match; Same for the Extended Window Part

Match type	...Long match	Direct match	Long match*
Decode[0]	111 W_{12}-W_8	110 W_{12}-W_8	111 W_{12}-W_8
Decode[1]	M_7-M_0	D_{15}-D_8	11111111
Decode[2]	W_7-W_0	D_7-D_0	...M_7-M_0
Decode[3]		W_7-W_0	W_7-W_0

Level 5 Direct Match

Direct Match

Level 5: A better solution to Match Length Data Structure

Reserve Flag 110 in Short Match for the new Direct match; Same for the Extended Window Part

Match type	… Long match	Direct match	Long match*
Decode[0]	111 W_{12}–W_8	110 W_{12}–W_8	111 W_{12}–W_8
Decode[1]	M_7–M_0	D_{15}–D_8	11111111
Decode[2]	W_7–W_0	D_7–D_0	...M_7–M_0
Decode[3]		W_7–W_0	W_7–W_0

Level 6 Direct Long Match

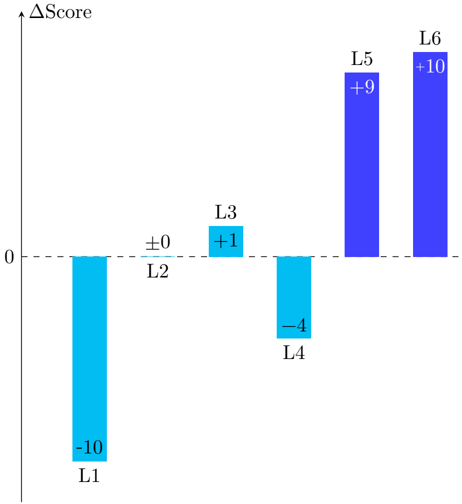
Direct Long Match

Level 6: Introduce a 4 Bytes threshold $D_{31} - D_0$ for Match Length

Reserve Flag 0x00 in the first $M_{n+7} - M_0$ for the Max of Direct Long Match; Same for the Extended Window Part

Match type	Long match	Direct Long match*
Decode[0]	111 W_{12}-W_8	111 W_{12}-W_8
Decode[1]	11111111	11111111
Decode[2]	...M_7-M_0	00000000
Decode[3]	W_7-W_0	$D_{31} - D_0$
Decode[4]		W_7-W_0

2 Further Levels of our LZSS Algorithm



Compiler Optimizations: Branch Prediction

```
if (len > 264 - 2)[[unlikely]]
//Max of Match Length 256+8
while (len > 264 - 2) {
output[index++] = (7 << 5) + (distance >> 8);
output[index++] = 264 - 2 - 7 - 2;
output[index++] = (distance & 0xff);
len -= 264 - 2;
}
cmp =(__builtin_expect(!!(distance < windows_size), 1))
? read_uint32(ref) & 0xffffff : 0x1000000;
return(sequence==cmp);
```

Compiler Optimizations: Some Helper Functions for Accessing Data

```c
#define read_uint64(ptr) ((uint64_t*)(ptr))[0]
#define read_uint32(ptr) ((uint32_t*)(ptr))[0]
```

Concurrent Decompression: Parallelization

i. Divide the Input into n subgroups(via 2D arrays.)
ii.Use Multithreading to decode the n subgroups and put into the same vector.

```
vector<thread> threads;
auto lambda=[&](int tid){
    Decompress(buffer3+tid*each_numbytes,
    compressed_size[tid],
    buffer4+tid*each_numbytes);
};
for(int tid=0;tid<num_threads;tid++)
    threads.push_back(thread(lambda,tid));
for(int tid=0;tid<num_threads;tid++)
    threads[tid].join();
```

Concurrent Decompression: Parallelization

```
chrono::time_point<std::chrono::system_clock> begin_time=
std::chrono::system_clock::now();
// Some Operations Done here
auto end_time = std::chrono::system_clock::now();
chrono::duration<double, std::milli> duration_mili =
    end_time - begin_time;
printf("Duration=%ldms", duration_mili.count());
```

sleep(10)	Duration = 10 ms
num_threads=1	Duration = 5 ms
num_threads=2	Duration = 3 ms
num_threads=8	Duration = 2 ms
num_threads=14	Duration = 1 ms

Ending: Reflection and Acknoledgement

Core values: Collaboration, Innovation, Continuous Learning

Thank you! Mentors, Huawei, and other Participants