Dictionary Management for LZ78 and LZW

Jordan Kaisman
Institute for Computational and Mathematical Engineering
Stanford University

1 Introduction

The LZ78 family of compressors is characterized by the storage of previously encountered phrases in a dictionary. Unlike the LZ77 family, which can match phrases to any earlier substring of the message, matches for the LZ78 family must extend a specific, previously parsed phrase. This requirement necessitates an explicit dictionary of past phrases. While theoretically unbounded, practical implementations are constrained by available memory, presenting the fundamental problem of how to manage the dictionary once it becomes full. We evaluate three candidate strategies: freeze, flush, and least recently used (LRU) pruning.

2 Literature Review

The LZ78 algorithm, introduced by Ziv and Lempel in 1978 [1], operates by incrementally building a dictionary of previously seen phrases. It checks if a symbol is already in the dictionary. We require that each new entry is a one-character extension of an existing phrase, and seek the longest such phrase. In other words, we seek the longest matching prefix in the dictionary. Any phrase can thus be represented as a pair (p, c), where p is the parent phrase and c the extending character. Equivalently, it can be represented as $(idx(p), c)$, where $idx(p)$ is the index of p. This form is what enables compression.

2.1 Dictionary Structures

The dictionary is naturally represented as a *Trie (Prefix Tree)*, where nodes represent phrases and edges represent extending characters [2]. The phrase corresponding to a node is defined by the path from the root to that node. Parsing then involves traversing the tree from the root using the input characters until a leaf or missing edge is encountered.

Though tries are conceptually simple, they are often memory inefficient. Array-based implementations require preallocating pointers for every possible character (e.g., 256 for ASCII), resulting in wasted space for sparse nodes. Conversely, object-based implementations (standard in Python) avoid empty slots but incur significant overhead.

A robust alternative is the *Hash Map Dictionary.* Instead of using explicit node objects or arrays, a single hash table maps the tuple $(\text{parent\_index}, \text{char})$ directly to a child_index. This ensures that memory is consumed only by edges that actually exist, avoiding both empty pointer waste and object overhead.

2.2 LZW and the kwkwk Edge Case

The LZW algorithm, introduced by Welch [3], reduces the overhead of a match by preloading the dictionary with all possible single symbols. Instead of transmitting c alongside $idx(p)$, it uses c as the start of a new phrase. Essentially, it places the partition between phrases before c rather than after it. While it still adds the same concatenated phrase pc to the dictionary, it transmits only the parent pointer, thereby saving the cost of transmitting explicit literals.

The drawback of this approach is a synchronization issue in one specific edge case: when the encoder uses a new dictionary entry immediately after creating it. We can derive the specific structure of the problematic string explicitly. Let m denote the most recently parsed phrase and $d = p_1 p_2$ the next two parsed phrases, and $k = p_2[0]$ the first character of the p_2.

- **Encoder Protocol:** Send idx(p_1) $\rightarrow$ Add p_1k to Dict $\rightarrow$ Send idx(p_2).
- **Decoder Protocol:** Receive idx(p_1) $\rightarrow$ Add $mp_1[0]$ to Dict $\rightarrow$ Receive idx(p_2) and extract first letter k $\rightarrow$ Add p_1k to Dict.

Since k is bundled within phrase p_2, the decoder must receive p_2 before it can construct p_1k and add it to the dictionary. This logic fails if p_2 turns out to be exactly the new phrase being added ($p_2 = p_1k$). In this case, the decoder attempts to access p_2 at an index it has not yet encountered.

If $p_2 = p_1k$, then since k is the first letter of p_2, p_1 must also begin with k. If we let $\omega = p_1[1:]$ (the remainder of p_1), we have:

$$p_1 = k\omega \quad \text{and} \quad p_2 = k\omega k$$

Substituting these back into d, we obtain the pathological string $d = k\omega k\omega k$. This form will cause the decoder to access an unoccupied index. Fortunately, this form is the only possible cause of such an error. Upon reading an invalid index the decoder can easily reconstruct the intended phrase, provided it remembers the previous phrase and its first symbol.

3 Methods

We implement an LZ78 compressor with a fixed maximum dictionary size and evaluate three memory management strategies: Freeze, Flush, and LRU Prune [4].

3.1 Dictionary Implementation

We utilize a *hash map* as the primary data structure. To minimize the overhead of Python objects, we abstract the hash logic using Python's native dictionary class. The dictionary maps the unique signature of a phrase to its assigned index:

$$\text{Dict} : (\text{parent\_index}, \text{character}) \rightarrow \text{child\_index}$$

3.2 Memory Management Strategies

- **Freeze:** Once the dictionary reaches its maximum capacity N, no further entries are added. The dictionary becomes static, effectively turning the compressor into a static lookup table.
- **Flush:** When the dictionary fills, it is completely cleared (reset to initial state). This is equivalent to segmenting the file and compressing each segment independently. While effective for heterogeneous data (e.g., distinct languages concatenated), it is on average only half full. Especially in the immediate aftermath of a reset event, a very small dictionary means limited compression.
- **LRU Prune:** When the dictionary is full, we identify the *Least Recently Used (LRU)* leaf node, remove it, and reuse its index for the new phrase. Crucially, we only prune *leaf nodes*. Pruning an internal (parent) node would break the prefix chain for all its descendants, invalidating multiple dictionary entries.

3.3 Efficient LRU Pruning: The Implicit Doubly Linked List

A naive LRU implementation using timestamps would require a $O(N)$ scan to find the oldest leaf. To achieve $O(1)$ pruning complexity, we must maintain the access order of leaves. We utilize a *Doubly Linked List (DLL)*, where the head represents the most recently used (MRU) leaf and the tail represents the LRU leaf.

Memory-Efficient Parallel Arrays Layout

Implementing a standard DLL in Python using `Node` objects incurs significant memory overhead (approx. 64 bytes per object) due to object headers and metadata. To circumvent this, we implement the DLL implicitly using parallel arrays.

We maintain five auxiliary arrays, all indexed by child_index:

1. **`prev` and `next` (DLL Pointers):** Integers representing the indices of the previous and next leaves in the LRU chain. A value of -1 indicates the end of the list.
2. **`parent` and `char` (Reverse Map):** When a leaf is pruned by its index (the `tail`), we must also remove its entry from the main Hash Map. Since the Hash Map is keyed by $(\text{parent}, \text{char})$, we need a reverse lookup mechanism child_index $\to (\text{parent}, \text{char})$. Storing this in arrays avoids a costly search.
3. **`child_count`:** An integer counter tracking how many children a node has. This allows $O(1)$ verification of whether a node is a leaf (child_count == 0) and is thus eligible for pruning.

Pruning Logic

When the dictionary is full and a new phrase must be added:

1. **Identify Victim:** Access the `tail` of the DLL to find the index of the LRU leaf.
2. **Remove from Dict:** Use the `parent` and `char` arrays to retrieve the key for the victim and delete it from the main Hash Map.
3. **Assess Parent of Target:** Decrement the `child_count` of the victim's parent. If the parent's count drops to zero, it becomes a leaf.
4. **Recycle Index:** The victim's index is reused for the new phrase.
5. **Insert New:** The new phrase is added to the Hash Map, its metadata is written to the arrays, and it is inserted at the `head` of the DLL.
6. **Assess Parent of New Node:** Increment the `child_count` of the new node's parent. If count is 1 (parent was a leaf), remove parent from the DLL.

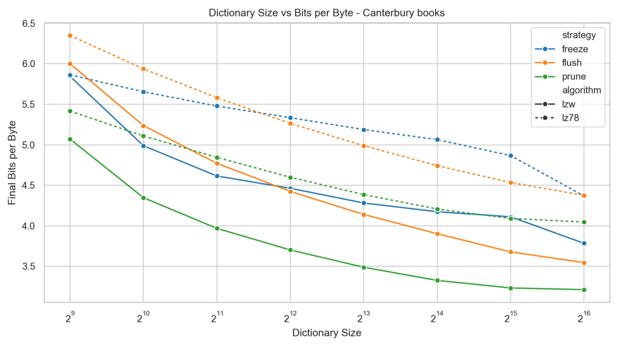
This structure ensures that all dictionary operations—lookup, insertion, and pruning—occur in $O(1)$ time while maintaining minimal memory overhead.

Leaf Reversion: The Parent Invariant

We have neglected one critical detail: in the case that the target's parent reverts to a leaf in step 3, how should we know where in the DLL to place it? Indeed, this suggests that we should actually keep track of relative access of all nodes. Fortunately, we have a simple invariant: the time of last access of an internal node is the minimum among its children. This is because an internal node is only accessed when it is gaining a new descendant. In particular, an internal node with only one child has the exact same time of last access. If the child is pruned, the internal node becomes a leaf and inherits the position of its child. In particular, it takes the place of its child at the tail of the list.

4 Results and Analysis

4.1 Dictionary Size



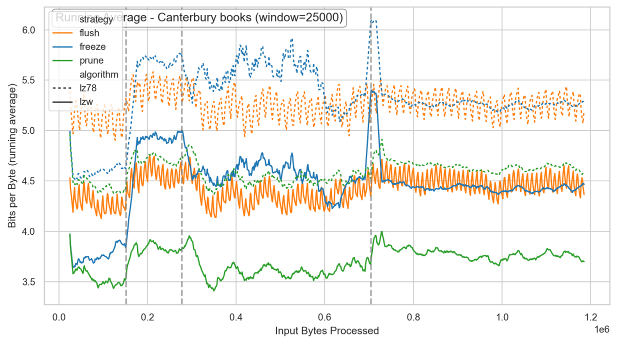
There are two competing factors that influence compression: match length and pointer length. Ideally, match lengths should be large while pointer lengths remain small. Increasing the dictionary size enables longer matches, but this comes at the cost of larger pointers; specifically, a dictionary of size 2^n requires n-bit pointers. Thus, pointer size grows logarithmically with dictionary capacity.

For the range of sizes considered, compression efficiency improves logarithmically with dictionary size. This indicates that enlarging the dictionary yields diminishing returns. Since the compression ratio is determined by the relationship between pointer length and match length, and pointer size grows logarithmically, these results imply that the average match length must also be growing logarithmically to maintain improved performance.

Note: LZ78 and LZW can be implemented with variable-length pointers, where the pointer bit-width increases only when the dictionary reaches the next power of two. While theoretically significant for unbounded dictionaries, the early gain from smaller pointers is negligible for freeze and prune in a finite-memory setting, as these dictionaries fill quickly and remain full. For flush, however, the lack of variable-length pointers causes underestimation of true compression efficiency.

4.2 Local Compression Efficiency

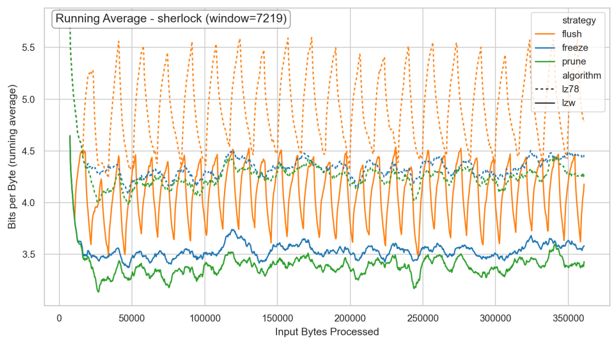
We track the evolution of compression efficiency via a running bpb average.



The Canterbury Corpus consists of four concatenated books. Flush is better suited to this heterogeneity, achieving better compression than freeze over much of the data.

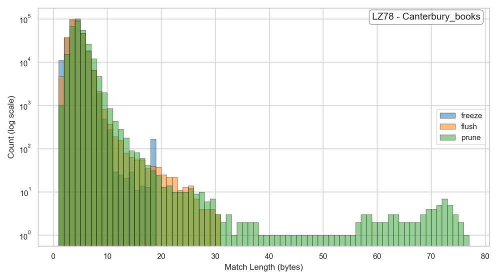
The Sherlock text is homogeneous, allowing freeze to compress better than flush. In both cases prune consistently outperforms the others. Since all matches are transmitted as pointers of a fixed length, better compression requires longer matches. In order to produce a lower bpb, pruning must achieve longer matches on average. This means that shorter phrases of lower frequency are pruned and replaced by longer phrases of higher frequency.

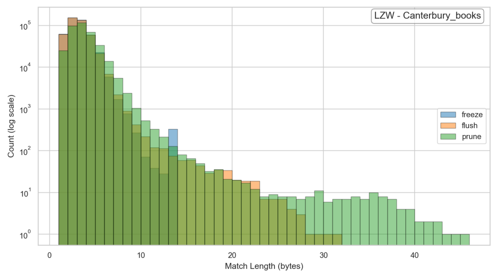
We also observe that the trajectories of all 6 variants are highly correlated despite their distinct phrase matching schemes. This suggests that although individual curves have small random fluctuations, larger trends are determined by the compressibility (entropy) of the underlying data.



4.3 Match Length Distribution

Motivated by the realization that pruning produces longer matches on average, we inspect the terminal distribution of match lengths in finer detail.

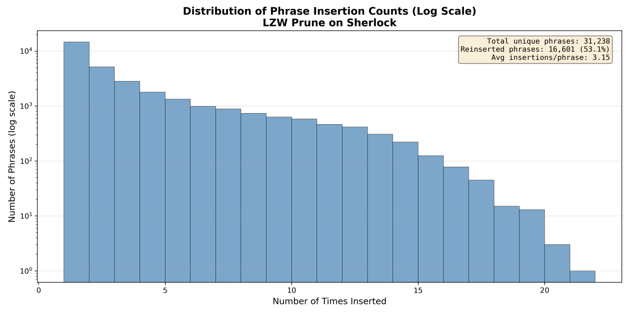




Pruning generates a much thicker tail.

4.4 Dictionary Reinsertion

We track the frequency with which the original dictionary entries (before the first prune) are reinserted into the dictionary. This reveals that reinsertion is actually quite common: many original short phrases are pruned and later reinserted, sometimes multiple times.



5 Conclusions

We have shown LRU pruning to be an effective strategy for improving compression in both LZ78 and LZW. The data indicates that the flexibility of pruning enables longer matches, and in turn a better compression ratio. We have also established that the prune dictionary has a tendency to reinsert previously pruned phrases, suggesting that it is in a sense bound to the original freeze dictionary. It can shed phrases temporarily, but many of the phrases encountered while initially building up the dictionary will likely occur again. They are not fundamental phrases, but they are small or mildly common enough to reappear periodically.

References

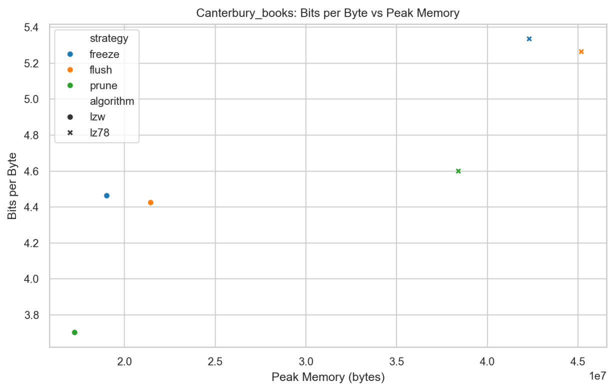
[1] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, pp. 530–536, Sept. 1978.

[2] D. Salomon and G. Motta, *Handbook of Data Compression*, 5th ed. London, UK: Springer, 2010.

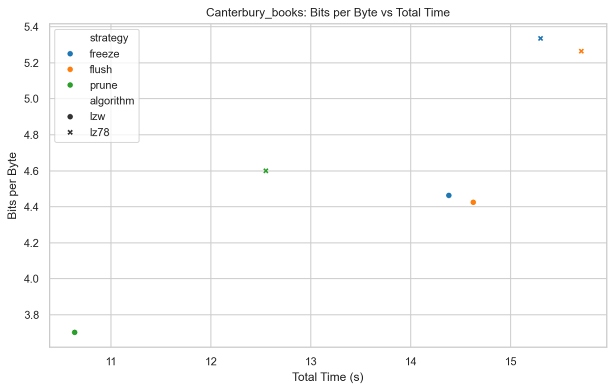
[3] T. A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[4] S. Bunton and G. Borriello, "Practical dictionary management for hardware data compression," *Commun. ACM*, vol. 35, pp. 95–104, 1990.

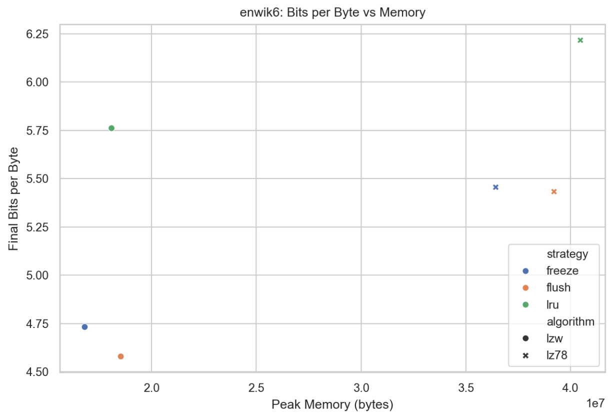
A Appendix



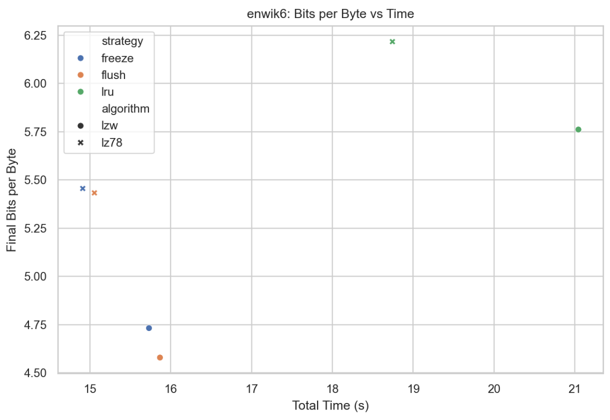
Canterbury_books: Bits per Byte vs Peak Memory
strategy
freeze
flush
prune
algorithm
lzw
lz78
Bits per Byte
Peak Memory (bytes)
1e7



Canterbury_books: Bits per Byte vs Total Time
strategy
freeze
flush
prune
algorithm
lzw
lz78
Bits per Byte
Total Time (s)



enwik6: Bits per Byte vs Memory
strategy
freeze
flush
lru
algorithm
lzw
lz78
Final Bits per Byte
Peak Memory (bytes)
1e7


enwik6: Bits per Byte vs Time
strategy
freeze
flush
lru
algorithm
lzw
lz78
Final Bits per Byte
6.25
6.00
5.75
5.50
5.25
5.00
4.75
4.50
15
16
17
18
19
20
21
Total Time (s)