Updating Permutation-Compressed Hypergraphs

Enno Adler, Stefan Böttcher, and Rita Hartel

Paderborn University,
Warburger Straße 100, Paderborn, Germany
{enno.adler, stefan.boettcher, rita.hartel}@uni-paderborn.de

Abstract. Hypergraphs model complex, non-binary relationships such as co-authorships, social group memberships, and recommendation systems. Like traditional graphs, hypergraphs can grow large, posing challenges for storage, transmission, and query performance. Moreover, the data can change as new papers are published, new social groups form, existing social groups accept or remove members, and new products are published. In this paper, we extend the hypergraph compression method HyperCSA to perform updates of the compressed hypergraph. Dynamic HyperCSA computes the exact compressed hypergraph representation that the steps *decompress-update-compress* of HyperCSA produce, but is orders of magnitude faster. Thereby, dynamic HyperCSA achieves the same compression ratios of 26% to 79% as HyperCSA on the modified, real-world hypergraphs and the query performance of being 6 to 40 times faster on real-world hypergraphs than other approaches.

Keywords: hypergraph compression, update operations, permutation-based compression

1 Introduction

Hypergraphs are powerful tools for modeling complex, non-binary relationships in data. Unlike traditional graphs, where edges connect only two nodes, hypergraphs allow edges to connect any number of nodes, enabling a more accurate representation of real-world systems. This modeling flexibility has made hypergraphs useful in a range of domains, including social networks [12], music [11] and news [6] recommendation systems, academic authorship networks, and tagged data repositories from online platforms [4]. To store and query large datasets, Adler et al. [1] recently introduced HyperCSA, a permutation-based hypergraph compression scheme based on the compressed suffix array by Sadakane [9]. However, HyperCSA cannot modify the underlying hypergraph yet. In this paper, we show how to alter the compressed representation of HyperCSA to reflect changes in the hypergraph data.

Herein, our main contributions are as follows:

- Algorithms[1] to add and remove nodes or edges from a hypergraph and algorithms to add and remove a single node from an edge in a hypergraph compressed by HyperCSA.
- Theoretical worst-case runtimes for all mentioned algorithms.
- A practical evaluation showing that the proposed algorithms significantly outperform the naive decompress-update-compress approach of updating a hpergraph compressed with HyperCSA.

[1] An implementation of all algorithms proposed in this paper is available at `https://github.com/adlerenno/hypercsa`.

Enno Adler, Stefan Böttcher, Rita Hartel: Updating Permutation-Compressed Hypergraphs, pp. 97–111.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

2 Preliminaries

A *hypergraph* is a pair $G = (V_G, E_G)$ with nodes $V_G = \{0, 1, \ldots, p\} \subset \mathbb{N}_0$ and edges $E_G = \{e_0, \ldots, e_{m-1}\}$. For every hyperedge $e_i = \{v_0, \ldots, v_{n-1}\} \subseteq V_G$, $e_i \neq \emptyset$ and $\text{rank}(e_i) = |e_i| = n$. We call all hyperedges in E_G edges regardless of their rank. HyperCSA can also handle hypergraphs that contain the same edge multiple times. In Figure 1, we visualize our ongoing example.

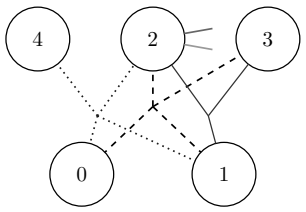


Figure 1: The example hypergraph H with nodes $V_H = \{0, 1, 2, 3, 4\}$ and hyperedges $E_H = \{\{0, 1, 2, 3\},\ \{1, 2, 3\},\ \{2\},\ \{0, 1, 2, 4\},\ \{2\}\}$.

We define $\text{degree}(v) = |\{e \in E_G : v \in e\}|$, $R_G = \max\{\text{rank}(e) : e \in E_G\}$, $\Delta_G = \max\{\text{degree}(v) : v \in V_G\}$, and $M_G = \sum_{e \in E_G} \text{rank}(e) = \sum_{v \in V} \text{degree}(v)$. For the example hypergraph H, we have $\text{degree}(2) = 5$, $R_H = 4$, $\Delta_H = 5$, and $M_H = 13$.

A string T of length $|T|$ over an alphabet Σ with total order $<$ is a sequence of characters $T[i] \in \Sigma$ for $0 \leq i < |T|$, so $T = T[0] \cdot T[1] \cdots T[|T| - 1]$. We write ϵ for the empty string, $T[i, j]$ for the substring $T[i] \cdot T[i + 1] \cdots T[j]$ of T, $T[i, j] = \epsilon$ if $i > j$, and $T[i..] = T[i, |T| - 1]$ for the suffix starting at position i. We call a position i an S-type, if $T[i] < T[i + 1]$, and L-type otherwise, also if $i = |T| - 1$.

We define $\text{rank}_T(x, c) = |\{i < x : T[i] = c\}|$ as the number of times c occurs in the first x characters of T (excluding the current position). Next, let $\text{select}_T(i, c) = \min\{x < |T| : \text{rank}_T(x, c) = i\} - 1$ be the position of the i-th c in T.

The suffix array SA [8] of a string T is a permutation of $\{0, \ldots, |T| - 1\}$ such that the i-th smallest suffix of T is $T[SA[i]..]$ with $i \in \{0, \ldots, |T| - 1\}$. As permutations are invertible, we denote the inverse of the suffix array by SA^{-1}. We use the term *T-position* for a position in T and *SA-position* for a position in SA. The compressed suffix array CSA by Sadakane [9] of a string T consists of a bitvector D of length $|T|$ and a permutation Ψ_{CSA} of $\{0, \ldots, |T| - 1\}$ (and other here omitted data structures). The bitvector D is a succinct representation for T using the suffix array SA: $D[i] = 1$ marks the first suffix in the sorted suffixes of T that start with a new character, so $D[i] = 1$ if and only if $T[SA[i]] > T[SA[i - 1]]$ or $i = 0$. We assume that all symbols of Σ have at least one occurrence in T. By this assumption, we get $T[SA[i]] = \text{rank}_D(i, 1) - 1$. $\Psi_{CSA}[i]$ stores the SA-position j that points to the next T-position $SA[i] + 1$ after the T-position $SA[i]$ at SA-position i. In one term, $\Psi_{CSA}[i] = SA^{-1}[(SA[i] + 1) \mod |T|]$. Ψ_{CSA} is a cyclic permutation and by computing the ranks on D for the SA-positions i in order of the cycle of Ψ_{CSA}, we retrieve a rotation of T.

The HyperCSA approach by Adler et al. [1] is inspired by RDFCSA by Brisaboa et al. [2], which is a self-index for the triples representing an RDF graph. As HyperCSA is a generalization of RDFCSA, all presented algorithms can be adapted for RDFCSA. HyperCSA's construction consists of three steps: First, we sort the nodes in an edge in ascending order and then, we sort the edges in lexicographically descending order. Thereafter, we concatenate all edges into one string T. For our example, we get

T-position i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	2	2	1	2	3	0	1	2	4	0	1	2	3
$SA[i]$	9	5	10	2	6	1	0	11	3	7	12	4	8

SA-position i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$D[i]$	1	0	1	0	0	1	0	0	0	0	1	0	1	1
$\text{node}_D(i)$	0	0	1	1	1	2	2	2	2	2	3	3	4	5
$\Psi_{CSA}[i]$	2	4	7	8	9	3	5	10	11	12	1	0	6	
$\Psi[i]$	2	4	7	8	9	5	6	10	11	12	0	3	1	

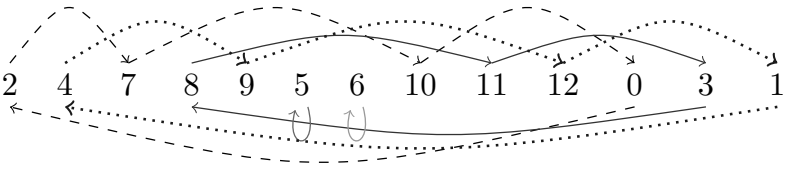


Figure 2: HyperCSA data structure for the hypergraph H. The node at index i is $\text{node}_D(i) = \text{rank}_D(i+1, 1) - 1$. The black numbers in Ψ_{CSA} are the numbers adjusted by the third step of HyperCSA's construction. The style of every edge in Ψ is the same as in Figure 1. The arrows of SA-positions with $\Psi[i] \leq i$ are drawn below the numbers, all others are drawn above.

$T = 2|2|123|0124|0123$, where the vertical bars are inserted as visual aid to identify the edges. Second, we compute the compressed suffix array of T and append a 1 to D for easier querying. We show D and Ψ_{CSA} of the compressed suffix array for our example T in Figure 2. Third, we modify Ψ_{CSA} such that each edge gets its own cycle in the permutation Ψ. We depict these cycles in Figure 2 and provide the algorithm in Agorithm 1. The adjustment from Ψ_{CSA} to Ψ only requires one change $\Psi_{CSA}[j] = i$ for every hyperedge e, where j is the single L-type position of e and i is the smallest position that belongs to e in Ψ_{CSA}.

Algorithm 1: HyperCSA.*adjust*Ψ

```
   Input: Ψ_CSA (in the following renamed to Ψ)
 1 current_position ← Ψ[0]
 2 next ← Ψ[current_position]
 3 last_first_node_position ← 0
 4 while current_position is not 0 do
 5 |   if next < current_position then
 6 |   |   Ψ[current_position] ← last_first_node_position
 7 |   |   last_first_node_position ← next
 8 |   current_position ← next
 9 |   next ← Ψ[next]
```

We define $\text{node}(i) = \text{rank}_D(i+1, 1) - 1$ and $\text{edge}(i) = \{\text{node}_D(\Psi^k[i]) : k = 0, 1, \dots\}$. We call the SA-position i a representative of the edge $\text{edge}(i)$ for all $i < |T|$. For example, 0, 2, 7, and 10 are all representatives of the edge $\{0, 1, 2, 3\}$ in Figure 2.

For $v \in V$, let $S(v) = \{\text{select}_D(v+1,1), \ldots, \text{select}_D(v+2,1)-1\}$ be the set of the SA-positions i where $T[SA[i]] = v$. For example, for $v = 1$, we have $S(1) = \{2,3,4\}$. By construction of the (compressed) suffix array, every $S(v)$ is a contiguous set of natural numbers, as can be seen in Figure 2.

HyperCSA uses the δ-code by Elias [3] on the differences $\Psi[i] - \Psi[i-1]$ between two values to encode Ψ. This works well because for node $v \in V$, $\Psi[S(v)]$ is a contiguous increasing integer sequence [1]. However, for this paper, we will work with the numerical Ψ without any encoding.

3 State of the Art on Updateable Hypergraph Compression

Many hypergraph compression approaches do not have an update algorithm for their compressed structure. This includes the approaches HybHuff of Zhao et al. [13] and the Hygra implementation using Ligra by Shun [10]. Liu et al. [7] propose an approach to reorder the elements of an incidence list and thereafter compress it. In comparison to the other mentioned methods, Liu et al. provides a maintenance algorithm that can be used to update the order of elements in the incidence lists after modification. The worst-case runtime of their compression algorithm and their maintenance algorithm is

$$\mathcal{O}\left(\sum_{u \in V} (mdeg(u))^2 + n^2\right)$$

with $mdeg(u)$ being the sum of the ranks of hyperedges that are incident to the node $u \in V$. Regardless of the type of modification, their approach always uses the same maintenance algorithm. In Table 1, we show all worst-case runtime bounds for the operations of HyperCSA and compare them to the worst-case runtime bounds for the incidence list. We highlight that every update operation of HyperCSA has a lower worst-case bound than the approach of Liu et al. [7].

4 Dynamic HyperCSA

We present the modification operations in the following order:

1. deleting an edge,
2. removing a node from an edge,
3. inserting a node to an edge,
4. adding an edge,
5. deleting a node, and
6. adding a node.

We remark that modifying the HyperCSA structure normally renders all prior by queries obtained representatives useless.

We will use our example from Figure 1 of a hypergraph H containing the hyperedges $\{2\}$, $\{2\}$, $\{0,1,2,3\}$, $\{1,2,3\}$, and $\{0,1,2,4\}$. We have $D = 10100100001011$ and $\Psi = [2,4,7,8,9,5,6,10,11,12,0,3,1]$.

4.1 Deleting an Edge

In a list of hyperedges, deleting the third edge is easy and unambiguous. However, in the HyperCSA data structure, it is not clear, which edge is the third. Therefore, any representative p of an edge edge(p) can be used to call deleteEdge(p).

method	incidence list	HyperCSA
construct	$\mathcal{O}(M_G)$	$\mathcal{O}(\lvert E_G\rvert R_G(\lg(R_G)+\lg(\lvert E_G\rvert)))$
edge(p)	$\mathcal{O}(\text{rank}(\text{edge}(p)))$	$\mathcal{O}(\text{rank}(\text{edge}(p)))$
decompress(G)	$\mathcal{O}(M_G)$	$\mathcal{O}(M_G)$
degree(v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
rank(i)	$\mathcal{O}(1)$	$\mathcal{O}(\text{rank}(i))$
contains(v)	$\mathcal{O}(\text{degree}(v))$	$\mathcal{O}(\text{degree}(v))$
contains(Q)	$\mathcal{O}(n + R_G \min_i(\text{degree}(v_i)))$	$\mathcal{O}(n + R_G \min_i(\text{degree}(v_i)))$
exists(Q)	$\mathcal{O}(n \min_i(\text{degree}(v_i)))$	$\mathcal{O}\left(\sum_{i=0}^{n} \log(\text{degree}(v_i))\right)$
deleteEdge(i)	$\mathcal{O}(R_G \cdot \Delta_G)$	$\mathcal{O}(M_G)$
removeNode(n, i)	$\mathcal{O}(\text{rank}(i) + \text{degree}(n))$	$\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$
insertNode(n, i)	$\mathcal{O}(\text{rank}(i) + \text{degree}(n))$	$\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$
addEdge(e)	$\mathcal{O}(\text{rank}(e))$	$\mathcal{O}(M_G)$
deleteNode(n)	$\mathcal{O}(M_G)$	$\mathcal{O}(\text{degree}(n) \cdot (M_G + R_G^2 \cdot \Delta_G))$
addNode(n)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 1: Comparison of worst-case runtimes of the incidence list and HyperCSA. Runtimes of queries are included from Adler et al. [1]. The space requirements are $2M_G + |E_G| + |V_G|$ integers for the incidence list and $M_G(H_0(T) + 1) + \mathcal{O}(M_G \log(H_0(T)))$ bits for HyperCSA (with $H_0(T)$ being the zero-order empirical entropy of T). We use Q as a shortcut for $\{v_1, \ldots, v_n\}$.

We delete an edge containing position p in two steps: First, we mark all positions we are going to delete. Second, we shorten the HyperCSA data structure. Both steps can be done for multiple deletions at once, but we present the algorithm for only one deletion.

First, we create a bitvector Z with $Z[i] = 1$ if $i = \Psi^k[p]$ for some k and $Z[0] = 0$ otherwise. Thereby, Z encodes the positions of the edge edge(p), which are the positions we delete from HyperCSA.

As an example, we show deleteEdge(3) on the hypergraph H of Figure 2. Because $\Psi^1[3] = 8$, $\Psi^2[3] = \Psi[8] = 11$, and $\Psi^3[3] = \Psi[11] = 3$, we get $Z = 0001000010010$. In the following paragraphs, we use D', Ψ', and Ψ'' for presenting and unambiguously referencing steps of our approach. However, our implementation does not generate new arrays for them. Our implementation alters the existing D and Ψ array.

The second step is to compute a new D' from D: The length of D' is the length of D minus rank(e), the rank of the edge we want to delete, which is the length of the cycle containing p. Retrieving D' is easy: We always append the next value to D', so we build D' left to right starting at position 0. We iterate over $Z[i]$ and $D[i]$: If $Z[i] = 0$, we append $D[i]$ to D'. If $Z[i] = 1$, we skip the value of $D[i]$ except if $D[i] = 1$, because $D[i] = 1$ indicates the start of the interval of the next node v. In this case, we add a 1 and skip the following position if it fulfills $D[i + 1] = 0$. If $D[i+1] = 1$, our call deleteEdge(p) requires reducing the degree of the node v to zero. Deleting the 1 of its span would cause all larger node IDs to shrink by 1, which would alter the data significantly. There is no way to compensate for this in HyperCSA, because this deletion violates the minimality of V. Therefore, we abort the deletion with an exception in this case. We discuss a solution in Section 4.6.

In our example, we get $D' = 10101000111$. In words, the nodes 0 and 1 occur two times, the node 2 is connected to four hyperedges, and the node 3 and 4 to only one hyperedge each. The information about which node is connected to which hyperedge is saved using Ψ.

The third step is to obtain a new Ψ'. This works similar to computing D' from D, but is even more straightforward: We iterate over $Z[i]$ and $\Psi[i]$ at the same time and we append $\Psi[i] - \text{rank}_Z(\Psi[i], 1)$ to Ψ' if and only if $Z[i] = 0$. Adjusting $\Psi[i]$ by subtracting $\text{rank}_Z(\Psi[i], 1)$ is necessary because we deleted $\text{rank}_Z(\Psi[i], 1)$ positions before position $\Psi[i]$ and we need to compensate this in Ψ'.

For example in Step $i = 2$, we have $\Psi[2] = 7$.

$$\begin{aligned}\Psi'[2] &= \Psi[2] - \text{rank}_Z(\Psi[2], 1)\\ &= 7 - \text{rank}_Z(7, 1)\\ &= 7 - 1 = 6.\end{aligned}$$

Overall, we obtain $\Psi' = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$.

Clearly, all data structures use $\mathcal{O}(M_G)$ space and we only iterate once over the deleted edge, D, and Ψ, so deleting the edge takes $\mathcal{O}(M_G)$ time.

4.2 Removing a Node from an Edge

The call removeNode(p, v) supposes us to delete the node v from the edge $e = \text{edge}(p)$. As an example, we will show removeNode$(3, 2)$. First, we check if e contains v. This is achieved by iterating over the positions of the edge starting at p by applying Ψ recursively on the result. For such a position i, we determine the corresponding node node(i) and check if node$(i) = v$. If we reach p again and we have not found the node v, there is no node to delete and we can finish. If we found v, we also acquired position q representing the connection of edge e to node v. In our example, we retrieve the edge at positions $(3, 8, 11)$, and $q = 8$.

Next, we calculate the new D' from D. Mostly, we copy the values from D, but we skip the value at position q if $D[q] = 0$. If $D[q] = 1$ and $D[q + 1] = 0$, we put a 1 at $D'[q]$ and skip the position $q + 1$ in D. If $D[q] = D[q + 1] = 1$, deleting q reduces the degree of v to zero. Like for deleting edges, this is not allowed and we abort. We discuss a solution in Section 4.6.

In the example, removeNode$(3, 2)$ means we remove node 2 from the edge $(1, 2, 3)$, which results in the new edge $(1, 3)$. We adjust D to $D' = 1010010001011$, where we deleted one 0 after the third 1, which means that the node 2 occurs one fewer time.

The third step is adjusting Ψ. The idea is simple: The jump pointing to q should now point to $\Psi[q]$, so the cycle skips position q afterwards. The main work on adjusting Ψ results from the small shifts of positions. We generate $\Psi'[i]$ by iterating over $\Psi[j]$, where the indices might differ because we skip position q, so $j = i$ for $i < q$ and $j = i + 1$ for $i \geq q$.

1. If $\Psi[j] < q$, we set $\Psi'[i] = \Psi[j]$.
2. If $\Psi[j] > q$, we set $\Psi'[i] = \Psi[j] - 1$.

Both cases are just a simple copy of Ψ to Ψ' considering that we skip position q. If Ψ points to a position after q, we subtract one, because all such positions will be shifted once to the left.

3. If $\Psi[j] = q$ and $\Psi[\Psi[j]] < q$, we set $\Psi'[i] = \Psi[\Psi[j]]$
4. If $\Psi[j] = q$ and $\Psi[\Psi[j]] > q$, we set $\Psi'[i] = \Psi[\Psi[j]] - 1$

These two cases handle the position j with $\Psi[j] = q$, which is the position we want to skip. Obviously, only one of the two cases occurs per computation of Ψ', and this case

applies to only one position, so they are the last two cases tested due to the efficiency. We skip the value q, which is why we use $\Psi[\Psi[j]]$. In principle, the case distinction is the same as in the previous two cases: If the next position is higher, we need to subtract one for the shift. The latter two cases also answer the question, whether we delete the highest node of the edge, because we always get the case that $\Psi[\Psi[j]] > q$ unless v is the highest node.

In the example, we get

$$\Psi' = [2, 4, 7, 10, 8, 5, 6, 9, 11, 0, 3, 1]$$

from

$$\Psi = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1].$$

At position $q' = 4$ where $\Psi[q'] = q = 8$, we retrieve for $\Psi'[4]$ the value $\Psi[\Psi[4]] = 11$ with the fourth case from above. Because $11 \geq 8$, we decrease 11 by 1, so we get $\Psi'[4] = 10$. Also, all references to positions greater than 8 are reduced by 1. But this has the negative effect that the subsequence $7, 10, 8$, which belongs to node 1, is no longer a contiguous increasing subsequence. Therefore, we now explain the function for modifying Ψ' such that the returned Ψ'' fulfills the contiguous increasing subsequences property again. The following operations will be used by other operations like insertNode($\cdot$) as well. Also, these steps are the last steps necessary for completing removeNode($\cdot$).

4.3 Interval Upward and Downward Sort

Assume that we are given a new D', Ψ', a position p and the information that p is the only position that violates the contiguous increasing subsequence property. In this section, we will explain the algorithm for the case that $\Psi'[p] > \Psi'[p+1]$, which we call *interval upward sort*. The case $\Psi'[p] > \Psi'[p-1]$ works analogously and is called *interval downward sort*. From the modifications to the HyperCSA structure, we always know if we need the upward or downward sort algorithm. We know by $\Psi'[p] > \Psi'[p+1]$ that the value $\Psi'[p]$ should have a higher position within the interval $S(v)$ of the node v that belongs to position p. This is the reason we call this modification the *interval upward* sort.

In our example, we get $D' = 1010010001011$, $\Psi' = [2, 4, 7, 10, 8, 5, 6, 9, 11, 0, 3, 1]$, and $p = 3$. We also know that the subsequence $[7, 10, 8]$ is not sorted correctly and that the $10 = \Psi[3]$ needs to be moved to a higher position within the interval $S(1)$.

First, we find the position q to switch the value of $\Psi'[p]$ to, which is the position within $S(v)$ such that $\Psi'[q] < \Psi'[p] < \Psi'[q+1]$. We find the position q by scanning $S(v)$ starting at p upwards towards the end of $S(v)$. Note, that if the scanning results in $p = q$, then everything is already sorted correctly. Also, finding q can be done in at most $|S(v)| = \text{degree}(v)$ steps. In our example, we get $q = 4$.

Second, we move every value in $(p, q]$ to the next smaller position and move the value $\Psi'[p]$ to q. We call the newly updated function Ψ''. This sorts the interval $S(v)$ correctly. This step obviously also needs only degree(v) steps. However, it creates the problem that all positions z with $\Psi'[z] \in [p, q] \subseteq S(v)$ need to be updated, because we changed the order of these values. We fix this with the steps three and four.

In our example, $(p, q] = (3, 4]$, so we only swap the values at the positions 3 and 4. Thus, we get $\Psi' = [2, 4, 7, 8, 10, 5, 6, 9, 11, 0, 3, 1]$. This joins the two cycles of the edges edge(3) and edge(4), so we need to adjust the jumps to the swapped values.

Third, we fix the jumps $\Psi'[z]$ towards the values in $(p, q]$. Note that we shifted the intended destination of the jumps down by one position, so the values $\Psi'[z]$ need to be decreased by one. We find such a position z with $\Psi'[z] \in (p, q]$ by iterating over Ψ'' until $(\Psi'')^k[y] \in (p, q]$ for a $y \in [p, q)$. We know $k = \text{rank}(e)$ for the edge e that belongs to position y, because the cycle of the edge is correct except for the last jump at $z = (\Psi'')^{k-1}[y]$. This means, we get the position z in at most R_G steps. We then set $\Psi''[z] = \Psi'[z] - 1$. This small decrease in value cannot cause a new violation of the contiguous increasing subsequence property at the positions z, because all values in Ψ' are unique and, by a decrease by one and restoring the uniqueness in step four by only one adjustment, the increasing order of the values around z cannot have changed, as this would require a change of at least two. A consequence of the order at positions z with $\Psi''[z] \in [p, q)$ not causing a new violation is, that we can update the values and do not have any branches of additional necessary updates other than step four. Thereby, we only have at most degree(v) edges to update and need at most $\mathcal{O}(R_G)$ steps for such an update.

In our example, we only need to fix the jump to position 4, which is now at position 3. We therefore traverse from the value starting at $\Psi''[3]$ until we get to position z with $\Psi''[z] = 4$, which requires the lookups at 8, 11 and 1. Then, we set $\Psi''[1] = 4 - 1 = 3$, so we have $\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 3, 1]$. Note that we have a second 3 here, which we will update in the next step.

Fourth, for the value z_p with $\Psi'[z_p] = p$, we set $\Psi''[z_p] = q$. Like in step three, we use the partial cycle in Ψ'' containing q to get to position z_p and then set $\Psi''[z_p] = q$. Note that we increased the value from p to q here. This means that z_p may now be a position that violates the contiguous increasing subsequence property in the same way p did before our adjustment. Thus, we set $p = z_p$ and restart with the first step.

In our example, we get $z_p = 10$ with $\Psi''[z_p] = p = 3$ after just one step by traversing $\Psi''[q] = \Psi''[4] = 10$. Now, we set $\Psi''[z_p] = q = 4$ and get

$$\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1].$$

We would continue with $p = z_p$. We can end iterating here, because $[0, 4]$ is a contiguous increasing subsequence and we are finished with the interval upward sort.

It is not trivial to see how long we must sort intervals. First, we prove that we need at most $R_G \cdot \Delta_G$ update steps. Then, we will prove that this algorithm needs at most $2R_G$ such update steps in case of removing or inserting a node to an edge.

Theorem 1. *Interval upward sort terminates after at most $R_G \cdot \Delta_G$ iterations. Its total runtime is in $\mathcal{O}(R_G^2 \cdot \Delta_G^2)$.*

Proof. First, if e is the edge that belongs to position p of the first iteration, all subsequent values of p are also representatives of edge e. The first p can thereby only move to the end of the interval $S(v)$ and, in the worst-case, is only moved one step per visit of the interval $S(v)$. After at most R_G steps, we visit $S(v)$. Thus, after at most $R_G\Delta_G$ iterations, we aborted early or we have moved p to the end of $S(v)$.

The runtime of each iteration is dominated by the third step, which takes $\mathcal{O}(R_G \cdot \Delta_G)$ steps. As we have at most $R_G \cdot \Delta_G$ iterations, the worst-case time is $\mathcal{O}(R_G^2 \cdot \Delta_G^2)$. □

Theorem 2. *We get an upper bound of $2R_G$ iterations if the interval upward or downward sort is called by removing or adding a single node. In this case, the interval upwards or downwards sort's worst-case runtime is $\mathcal{O}(R_G^2 \cdot \Delta_G)$.*

We can get an upper bound of $2R_G$ iterations if the interval upward or downward sort is called by removing or adding a single node. Then, interval upwards or downwards sort's worst-case runtime is $\mathcal{O}(R_G^2 \cdot \Delta_G)$.

Proof. First, if e is the edge that belongs to position p of the first iteration, all subsequent values of p also represent the same edge e. Next, we consider the suffixes of T: The move from position p to q sorts e towards the edges that have at least the same first symbol of the suffix, due to the value of Ψ. So, after $\text{rank}(e)$ steps, we update p inductively correctly by all $\text{rank}(e)$ symbols of its suffix. The former suffix during the construction is no longer necessary to consider as we used it only to obtain the contiguous increasing subsequence property in the first place. Thus, the reference of p is moved to the correct position at step $\text{rank}(e)+1 \leq R+1$. The second round of updates is adjusting the numbers according to this now corrected position. The next visit of $S(v)$ does not require any further changes in the interval. □

The overall result of removeNode$(3, 2)$ is $\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1]$ and $D' = 1010010001011$.

4.4 Inserting a Node to an Edge

With the call insertNode(p, v), we add the node v to the edge edge(p). As an example, we will show insertNode$(4, 2)$ on the result of the removeNode$(3, 2)$ example, so we have $D = 1010010001011$ and $\Psi = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1]$ initially. The call insertNode$(4, 2)$ will revert the change done by removeNode$(3, 2)$, so the result will be $D' = 10100100001011$ and $\Psi'' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$. We highlight that the representative 3 for the removal and the representative 4 for the insertion represent the same edge, but the actual value was changed due to the modifications made during the removal in the last section.

First, we need to find the pair of positions (i, j) such that $\Psi(i) = j$ and $\text{node}_D(i) < v < \text{node}_D(j)$. If such a pair does not exist, because for all nodes $v' \in \text{edge}(p)$, we have either $v' < v$ or $v' > v$, we want the pair (i, j) such that $\Psi(i) = j$ and $\text{node}_D(i) > \text{node}_D(j)$. During this traversal, if we already have $v \in \text{edge}(p)$, we abort the insertion early.

The next major goal is to insert j into the subsequence $\Psi[S(v)]$. Therefore, we first search the position $q \in S(v) \cup \{\max(S(v)) + 1\}$ with $\Psi[q-1] < j < \Psi[q]$ (to simplify, we assume $\Psi[-1] = -1$ and $\Psi[M_G] = M_G$ for the boundaries here). Because $\Psi[S(v)]$ is a contiguous increasing subsequence, we can use a binary search to find q.

In the example, we have $(i, j) = (4, 10)$. We search the position q in the subsequence $[5, 6, 9, 11]$ where we should put the value 10 using a binary search. We want to insert the 10 between 9 and 11, so $q = 8$ in Ψ.

Second, we copy D to D', but at the v-th span of D, we increase the length of that span by one. In the example, we get $D' = 10100100001011$.

Third, we do a similar step for Ψ': If both $z, \Psi[z] < q$, we set $\Psi'[z] = \Psi[z]$. If $z < q$ and $\Psi[z] \geq q$, we set $\Psi'[z] = \Psi[z] + 1$. If $z \geq q$ and $\Psi[z] < q$, we set $\Psi'[z+1] = \Psi[z]$. If both $z, \Psi[z] \geq q$, we set $\Psi'[z+1] = \Psi[z] + 1$. In other words, we move values higher than q up by one to create the space we need for the new node in the edge. We can now set $\Psi'[i] = q$ and $\Psi'[q] = j + 1$ and thereby inserted the node v to the edge.

In the example, the first copy steps results in $[2, 3, 7, 9, 11, 5, 6, 10, , 12, 0, 4, 1]$ and the two explicit assignments produce $\Psi' = [2, 3, 7, 9, 8, 5, 6, 10, 11, 12, 0, 4, 1]$. Now, the subsequence $[7, 9, 8]$ of node 1 is not ordered properly.

By the binary search, we know that the subsequence of the interval $S(v)$ is properly ordered. In the same way as for deleting a node from an edge, the increase by one cannot violate the contiguous increasing subsequence property, so the only position that might be incorrect is position i. Again, we can use the interval downward or upward sort here. We need the interval downward sort if we reduced $\Psi[i]$, as in our example, or the interval upward sort, if we increased $\Psi[i]$. Increasing $\Psi[i]$ happens if and only if we add a new highest node to the edge. In the example, we get $\Psi'' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$ as the result after the interval downward sort.

The runtime of insertNode(p, v) is dominated by the steps of copying D to D' and Ψ to Ψ', which both take $\mathcal{O}(M_G)$ time, or by the interval upward or downward sort using $\mathcal{O}(R_G^2 \cdot \Delta_G)$ time. So, in total we get $\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$ in the worst-case.

4.5 Inserting a new Edge

The call addEdge(e) inserts a new edge e into the existing hypergraph G. Compared to the previous three modifications of the structure, we do not have a representative p to work with. We assume that the nodes in e are sorted, so $e[0] < e[1] < \cdots < e[\text{rank}(e)-1]$. As an example, we will insert the edge $\{1, 2, 3\}$ into the hypergraph with the edges $\{2\}$, $\{2\}$, $\{0, 1, 2, 3\}$, and $\{0, 1, 2, 4\}$, which is given by $\Psi = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$ and $D = 10101000111$. This will undo the step of deleteEdge(3) in Section 4.1.

For e, we need one new position for every node $e[i]$. More precisely, for $i < \text{rank}(e)$, we insert one new element in every subsequence in $\Psi[S(e[i])]$. Using this information about the location of the newly inserted elements, we compute D' from D as follows: Let D' be a bitvector of length of $M_G + 1 + \text{rank}(e)$. We iterate over D and set the current position of D' to the bit occurring at $D[i]$ and we skip one position of D' if $D[i] = 1$ and $\text{node}_D(i) \in e$. Because the nodes in e are ordered, we will find the $e[i]$ in order of their index i. This results in $D' = 10100100001011$ from $D = 10101000111$, where we increased the length of the spans associated with the nodes 1, 2, and 3.

The HyperCSA structure was constructed using suffix sorting. To get the insert positions for the nodes of e, we will use this construction idea. The problem is that we modified Ψ_{CSA} to Ψ: From any position p in Ψ, we only know a prefix of the suffixes of the construction string T. For example, for any edge e', the longest exactly known prefix of a suffix starts at $e'[0]$. On the contrary, we know the construction rules of T, so we know that after edge e' was an edge e'' with $e' > e''$ or e' was the lexicographically smallest edge.

Our goal is to find the insert position p_0 of $e[0]$ in $S(e[0])$. We will use a binary search on the interval $S(e[0])$ and compare e to the current position. This position might not be a first node of an edge: In our example, the interval $S(e[0]) = S(1) = [2, 3]$ contains representatives to the edges $\{0, 1, 2, 3\}$, and $\{0, 1, 2, 4\}$, which both do not have the node 1 as their first node. Thus, we always compare the edge e to a part s of another edge e' that contains the node $e[0]$.

Let e' be another edge that contains $e[0]$. We only consider the part s of e' that consists exactly of the nodes of e' larger or equal to node $e[0]$. Therefore, we call s an *edge-suffix*. We refer with $\text{rank}(s)$ to the length of s. We end up in one of several cases:

- If $e[m] < s[m]$ for an $m < \min(\text{rank}(e), \text{rank}(s))$ and $e[i] = s[i]$ for all $i < m$: e has a smaller suffix than s.
- If $e[m] > s[m]$ for an $m < \min(\text{rank}(e), \text{rank}(s))$ and $e[i] = s[i]$ for all $i < m$: e has a larger suffix than s.

- If $e[i] = s[i]$ for all $i < \min(\text{rank}(e), \text{rank}(s))$:
  - If $\text{rank}(e) < \text{rank}(s)$: e has a smaller suffix than s.
  - If $\text{rank}(e) > \text{rank}(s)$: e has a higher suffix than s.
  - If $\text{rank}(e) = \text{rank}(s)$:
    - $e \neq e'$: e has a higher suffix than e'.
    - $e = e'$: we can choose, because both decisions will give us the same result.

The first two cases are the classic suffix sorting cases, so the first unequal node determines the order of the suffixes. Next, if all nodes are equal, one suffix can be longer than the other, which is the case if the ranks of e and s are different. Let us consider the case $\text{rank}(s) > \text{rank}(e)$ and $e[i] = s[i]$ for all $i < \text{rank}(e)$: The next node in s is larger than the previous and the next node n after the last node of e is smaller or equal to its previous node due to the construction of T, so $s[\text{rank}(e)] > s[\text{rank}(e)-1] = e[\text{rank}(e) - 1] \geq n$. Thus, e has the smaller suffix. The case $\text{rank}(s) < \text{rank}(e)$ works analogously. Here, the special case is if $\text{rank}(e) = \text{rank}(s)$, so we have $e = s$. If $s \neq e'$, there is a node $n \in e'$ that is strictly smaller than any node in e and we get $e > e'$. In T, the suffix after e is higher than the suffix after e' by the downward sort of the edges and thereby we have that e has the higher suffix. If $e' = s$, then $e' = e$. This means, their cycles would be next to each other in Ψ. In this rare case, we can use the cycle of e' to get all positions of e' easily, either all are before or all after the cycle of e'. This results in the same Ψ in both cases.

In the example of inserting $e = \{1, 2, 3\}$, the interval is $S(e[0]) = S(1) = [2, 3]$ and we first compare e to the edge containing position 3. Because $\Psi[3] = 7 \in S(2)$, we also have $e[1] = s[1]$. Next, $\Psi[7] = 9 \in S(4)$, so we have $e[2] = 3 < 4 = s[2]$, so e has the smaller suffix. By the binary search, we now compare to $\Psi[2] = 6$. As $6 \in S(2)$ and $\Psi[6] = 8 \in S(3)$, we did $\text{rank}(e)$ comparisons. Because $\Psi[8] = 0 < 8$, we also have reached the end of the edge-suffix s here, so $\text{rank}(e) = \text{rank}(s)$. If $e' = s$, then we should have $\Psi[8] \in S(1)$, but we have $\Psi[8] = 0 < S(1)$, so $e > s$. Thereby, the insert position is 3, which means we insert e before the edge e at position $p_0 = 3$.

Using the S-type position to determine the end of the edge-suffix s and the fact that the implication $e' \neq s \Rightarrow e' \neq e$ requires $e = s$, we can efficiently obtain the position of $e[0]$ without further extension of the cases above. To conclude, we can decide within at most R_G steps if e or s has the higher suffix and we need at most $\mathcal{O}(\log(\Delta_G))$ steps for the binary sort, so we can get the insert position p_0 of $e[0]$ in $\mathcal{O}(R_G \cdot \log(\Delta_G))$ steps.

For the other nodes, we use the search from insertNode($\cdot$): We know the position p_0 of $e[0]$, so we can search in the subsequence $\Psi[S(e[\text{rank}(e)-1])]$ for position q with the smallest value higher than or equal to p_0 using a binary search. We set $p_{\text{rank}(e)-1} = q$, because we want the contiguous increasing subsequence property. $\Psi[S(e[\text{rank}(e)-1])]$ fulfills the property, and we will set $\Psi'[p_{\text{rank}(e)-1}] = p_0$. Having $p_{\text{rank}(e)-1}$, we can search for $p_{\text{rank}(e)-2}$. So we iterate downward to search the needed positions until we find p_1. For the purpose of storing and using the information best, we use a bitvector Z of length $|D| = M_G + 1$ that is initially 0, and we set $Z[p_i] = 1$ for the found insert positions p_i for all $i < \text{rank}(e)$.

In our example, we saw that $p_0 = 3$. As $\text{rank}(e) = 3$, we next search the position p_2. Thus, we search in Ψ in the interval $S(e[2]) = S(3) = [8, 9)$ for the value $p_0 = 3$ using a binary search. Because $\psi[8] = 0$, we get $p_2 = 9$. Thereafter, we continue for p_1: Here, $S(2) = [4, 8)$ and we get $p_1 = 7$, because $\Psi[6] = 8$, $\Psi[7] = 9$, and we searched for the value $p_2 = 9$. To conclude, we do not get the values p_i, but we get $Z = 00010001010$.

Given Z, the construction of Ψ' is straightforward: We iterate over Ψ with index i, and if at the current position i the value $Z[i]$ is 1, we skip a position in Ψ', so the index j of Ψ' is increased by an additional step. This produces the yet unassigned positions in Ψ' for the new edge e. Additionally, we need to adjust the value $\Psi[i]$ before we can set it to $\Psi'[j]$. We must increase $\Psi[i]$ by the number of added unassigned positions up to $\Psi[i]$, so we set $\Psi'[j] = \Psi[i] + \text{rank}_Z(\Psi[i] + 1, 1)$, where the 1s in Z are exactly the number of unassigned positions. We increment $\Psi[i]$ by one, because if $Z[i] = 1$, we also need to consider that $Z[i] = 1$ means shifting this character by an additional position for a new unassigned position. We remark that we use a data structure to answer the rank queries in $\mathcal{O}(1)$ time, which highlights the advantage of encoding all insert positions p_i by a single bitvector Z.

In the example, we use the bitvector $Z = 00010001010$ to transform

$$\Psi = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$$

into

$$\Psi = [2, 4, 7, _, 9, 5, 6, 10, _, 12, 0, _, 1].$$

The underscores stand for the unassigned insert positions.

As the last step of inserting edge e, we assign values to the unassigned positions of the new edge. We use $p_{i-1} = \text{select}_Z(i, 1) + i$ for $1 \leq i \leq \text{rank}(e)$ to get the position of node $e[i-1]$. So, we set $\Psi'[p_{i-1}] = i_i$ for all $0 \leq i < \text{rank}(e)$ and $\Psi'[p_{\text{rank}(e-1)}] = p_0$.

In the example, we get $\Psi' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$. Together with $D' = 10100100001011$, we correctly inserted $e = \{1, 2, 3\}$ and reverted the changes.

Overall, for any edge $e \subseteq V$, we have

$$(\Psi', D') = \text{addEdge}(\Psi, D, e)$$

and

$$\text{deleteEdge}(\Psi', D', \text{exists}(\Psi', D', e)[0]) = (\Psi, D),$$

so we can revert any addition of a new edge. Also, if $e = \text{edge}(p)$, we have for any representative p,

$$\Psi', D' = \text{deleteEdge}(\Psi, D, p),$$

and

$$(\Psi, D) = \text{addEdge}(\Psi', D', e),$$

so the deletion is invertible if and only if we are allowed to delete the edge that belongs to p. We extend this by a special call $\text{exists}(\Psi, D, e)$, which gives us a set of edge representatives of edges that are equal to the edge represented by p. We remark that we are not allowed to delete an edge if we thereby delete the last connection of a node. In the next paragraph, we present an idea to overcome this issue.

4.6 Unconnected Nodes in HyperCSA

With deleteEdge$(\cdot)$ or removeNode$(\cdot)$, we can reach the case that we want to delete the last connection of a node n. In this chapter, we present an easy solution to that problem. Let N be a bitvector of length $|V_G|$. If the node $n \in V$ has at least one incident hyperedge e, then $N[n] = 1$. Up to this point, this was a requirement, but

we allow nodes n that are not connected to any hyperedge by setting $N[n] = 0$. Consequently, we can obtain the node degrees as

$$\text{degree}(n) = \begin{cases} 0 & N[n] = 0 \\ |S(\text{rank}_N(n,1))| & N[n] = 1 \end{cases}.$$

Thus, we can now delete the last connection of a node n by setting $N[n] = 0$. This occurs while deleting a span from D that had only length 1, so if we had $D[i] = D[i+1] = 1$ for $i = \text{select}_D(n, 1)$.

Allowing unconnected nodes needs a small adjustment for queries and for modifications of HyperCSA. First, we look at the queries: Given a sequence s of nodes n_i for any query, we check for every node n_i in s that $N[n_i] = 1$, otherwise the query obviously has no matching edge by definition of N. If all nodes are connected, we need to translate them, because there might be lower nodes that are not connected to any edge. We set $n'_i = \text{rank}_N(n_i, 1)$ and s' to be the sequence of the n'_i. The rank$(\cdot)$ reduces n_i by the number of zeros in N before n_i. We can then perform the query using s'. Without the translation, we search within the wrong spans of D.

We adjust the values of nodes using the transformation from n_i to n'_i of the previous paragraph. In the case of adding at least one node n to the structure that has $N[n] = 0$, we set $N[n] = 1$ and add a new span of length 1 to D at the right location besides the changes to Ψ described in Section 4.4 or Section 4.5.

It is now also possible to add a new node to G by appending a zero to N. If the node should be inserted in between other nodes, we shift the values of higher positions in N to the next higher position. We delete a node n from G that has $N[n] = 0$, by deleting the n-th bit in N, so we shift the bits with a position higher than n to the position one smaller and shorten the length of N by one. If we want to delete a node with $N[n] = 1$, we first detach all connected edges from that node. A naive solution calls removeNode$(\text{select}_D(n, 1), n)$ for $S(n)$ times. Each such call will detach one edge from the node n. This method is naive, because we could detach all edges from the node n at the same time and reduce the update cost for the remaining parts of D and Ψ, which dominates the runtime. We do not present an advanced algorithm here, because in our opinion, deleting a connected node happens rarely in real-world scenarios.

5 Experimental Evaluation

Our tests were performed on a Debian 5.10.209-2 machine with 512GB RAM and 128 Cores Intel(R) Xeon(R) Platinum 8462Y+ @ 2.80GHz. We evaluated the social network community hypergraphs CoFr and CoOr, which are available from SNAP [5]. The statistics of the datasets are given in Table 2.

hypergraph G	$\lvert V_G\rvert$	$\lvert E_G\rvert$	M_G	R_G	Δ_G
com-friendster (CoFr)	7,944,949	1,620,991	23,479,217	9,299	1,700
com-orkut (CoOr)	2,322,299	15,301,901	107,080,530	9,120	2,958

Table 2: Used datasets from SNAP [5].

Construction of the CoFr dataset takes 3 minutes and 33.951 seconds. Deleting an edge from CoFr in the HyperCSA format takes only about 26.8349 seconds. We added

an edge within about 27.1503 seconds. Inserting a node takes about 38.0888 seconds. Attempting to delete a node from an edge to which the node is not connected takes only 7.366 milliseconds.

The construction of the CoOr dataset takes about 51 minutes. Deleting an edge takes about 93.056 seconds. We add a new edge to the HyperCSA-compressed CoOr hypergraph in about 93.2257 seconds. Attempting to delete a node from an edge to which the node is not connected takes only 17.381 milliseconds, whereas adding a new node to an edge takes about 126.184 seconds.

The results of both CoFr and CoOr show that our update-algorithms are significantly faster than a decompress-update-compress approach, which needs the time to compress the datasets and the updates have only a negligible influence on the compression time. If multiple updates are needed, for example deleting multiple edges at a time, these steps can be combined to reduce the overall time needed.

6 Conclusion

In this paper, we introduced algorithms to add and remove nodes or edges and algorithms to add or remove single nodes from edges in a hypergraph that was compressed using HyperCSA. We provided worst-case runtimes for all presented algorithms and showed by a practical evaluation that the algorithms are much faster than the naive decompress-update-compress approach.

References

1. E. Adler, S. Böttcher, and R. Hartel: *Compressing hypergraphs using suffix sorting*, in Data Compression Conference, DCC 2026, Snowbird, UT, USA, March 24-27, 2026, IEEE, 2026, pp. 113–122.
2. N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro: *A Compact RDF Store Using Suffix Arrays*, in String Processing and Information Retrieval, Cham, 2015, Springer International Publishing, pp. 103–115.
3. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Transactions on Information Theory, 21(2) 1975, pp. 194–203.
4. G. Ghoshal, V. Zlatic, G. Caldarelli, and M. E. J. Newman: *Random hypergraphs and their applications.* CoRR, abs/0903.0419 2009.
5. J. Leskovec and A. Krevl: *SNAP Datasets: Stanford Large Network Dataset Collection.* http://snap.stanford.edu/data, June 2014, Last Accessed: 19.04.2026.
6. L. Li and T. Li: *News recommendation via hypergraph learning: encapsulation of user behavior and news content*, in Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM '13, New York, NY, USA, 2013, Association for Computing Machinery, p. 305–314.
7. Y. Liu, Q. Luo, M. Xiao, D. Yu, H. Chen, and X. Cheng: *Reordering and compression for hypergraph processing.* IEEE Trans. Computers, 73(6) 2024, pp. 1486–1499.
8. U. Manber and G. Myers: *Suffix Arrays: A New Method for On-Line String Searches*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA, 1990, pp. 319–327.
9. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* J. Algorithms, 48(2) 2003, pp. 294–313.
10. J. Shun: *Practical parallel hypergraph algorithms*, in PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020, 2020, pp. 232–249.
11. S. Tan, J. Bu, C. Chen, B. Xu, C. Wang, and X. He: *Using rich social media information for music recommendation via hypergraph model.* ACM Trans. Multimedia Comput. Commun. Appl., 7S(1) Nov. 2011.

12. S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen: *Mapping users across networks by manifold alignment on hypergraph*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28, 2014.
13. T. Zhao, D. Zhao, L. Guo, and N. R. Tallent: *Hybhuff: Lossless compression for hypergraphs via entropy-guided huffman-bitwise coordination.* CoRR, abs/2506.15844 2025.