

On Optimally Partitioning a Text to Improve Its Compression

Davide Cologni,¹ PhD Candidate

Venice, Italy – 23 December 2025

¹RAVEN Group – Ca’ Foscari University of Venice,
Github: github.com/colobrodo,
Email: davide.cologni@unive.it

Problem: Text Partitioning

We have a compressor \mathcal{C} and a Text T of size n , it's possible to divide T into $k \leq n$ parts, $T[1..i_1 - 1]T[i_1..i_2 - 1]...T[i_{k-1}..n]$ and compress each of them individually with \mathcal{C} to improve the overall compression?

Intuitively we can group the most similar parts of the string together so each partition is better compressed by \mathcal{C} .

We do **not** permute the string we are only interested on partitioning it.

Text Partitioning Example

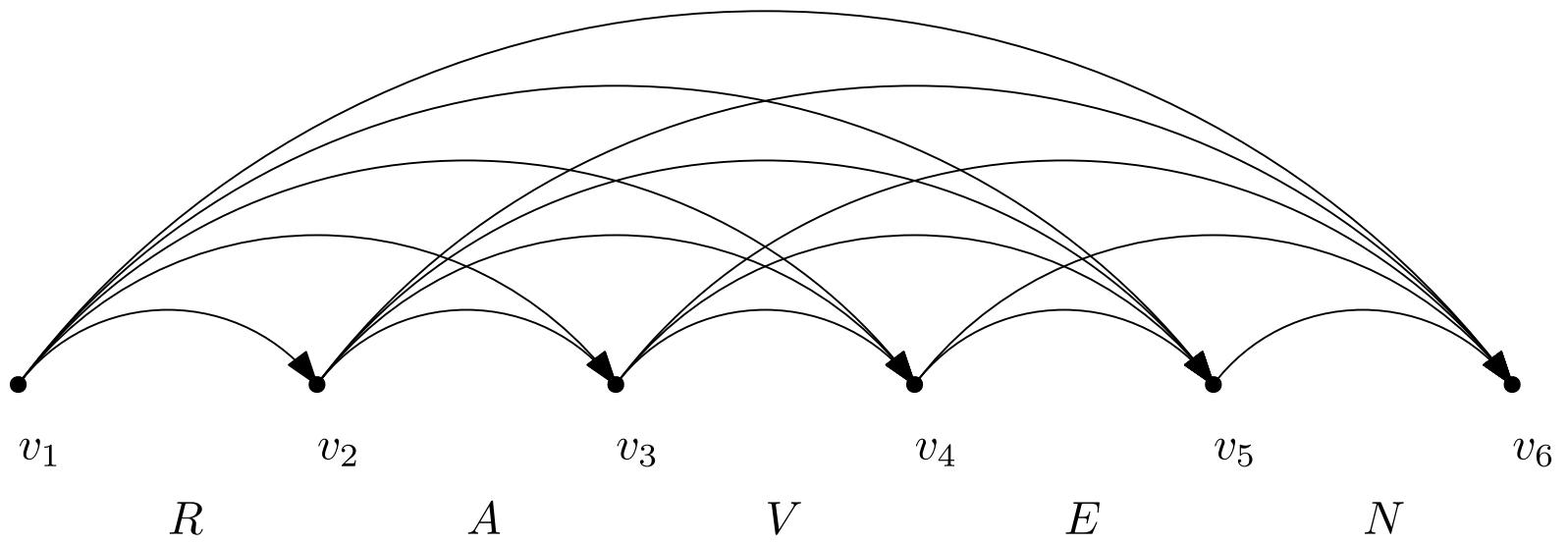
TODO We have a compressor \mathcal{C} and a Text T of size n , it's possible to divide T into $k \leq n$ parts so to improve the overall compression?

Intuitively we can group the most similar parts of the string together so each partition is better compressed by \mathcal{C} .

Reduction to SSSP

We can model each partition problem as a directed graph with $n + 1$ vertices, where an edge exists between v_i and v_j only if

$$1 \leq i < j \leq n + 1$$



Reduction to SSSP - Bijection between paths and partitions

We can then show that there exists a bijection from each path $\pi = (v_1, v_{i_1}) \dots (v_{i_k}, v_{n+1})$ in the graph, and a partitioning of the text T

$$T[1..i_1 - 1]T[i_1..i_2 - 1]\dots T[i_{k-1}..n]$$

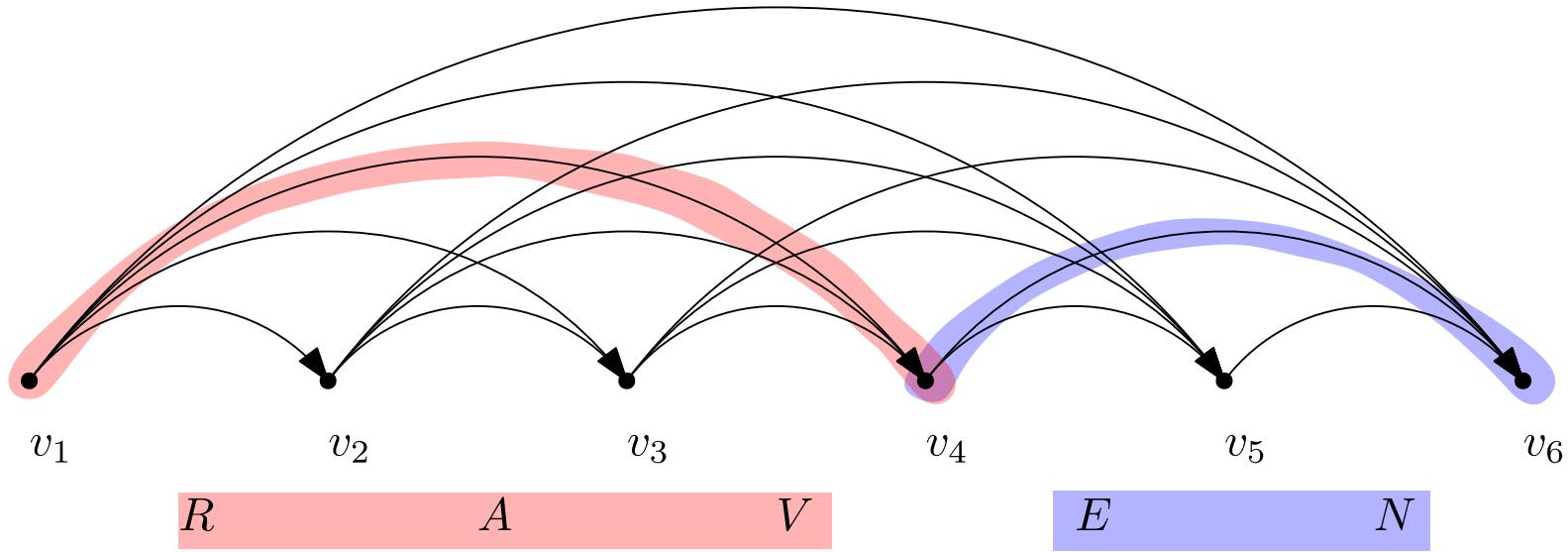


Figure 2: We can map the path $\pi = (v_1, v_4) (v_4, v_6)$ to the partitioning of the string $T[1, 3] T[4, 5]$

Reduction to SSSP - Bijection between paths and partitions

If we weight each edge (i, j) of the graph by the cost of compressing the corresponding text segment $w(i, j) = \mathcal{C}(T[i, j - 1])$, we can solve the partitioning problem *optimally* computing the **Single Source Shortest Path (SSSP)**

It can be computed efficiently in $O(|E|)$ time using a classic dynamic programming algorithm.

Problems:

1. Our graph has $O(n^2)$ nodes by construction
2. To initialize the weight $w(i, j)$ we should execute \mathcal{C} on every substring of the text

Assumption on \mathcal{C}

- Our compressor is *monotonic*: the compressed output on a suffix or a prefix of the string is always smaller than the compression on the whole string:

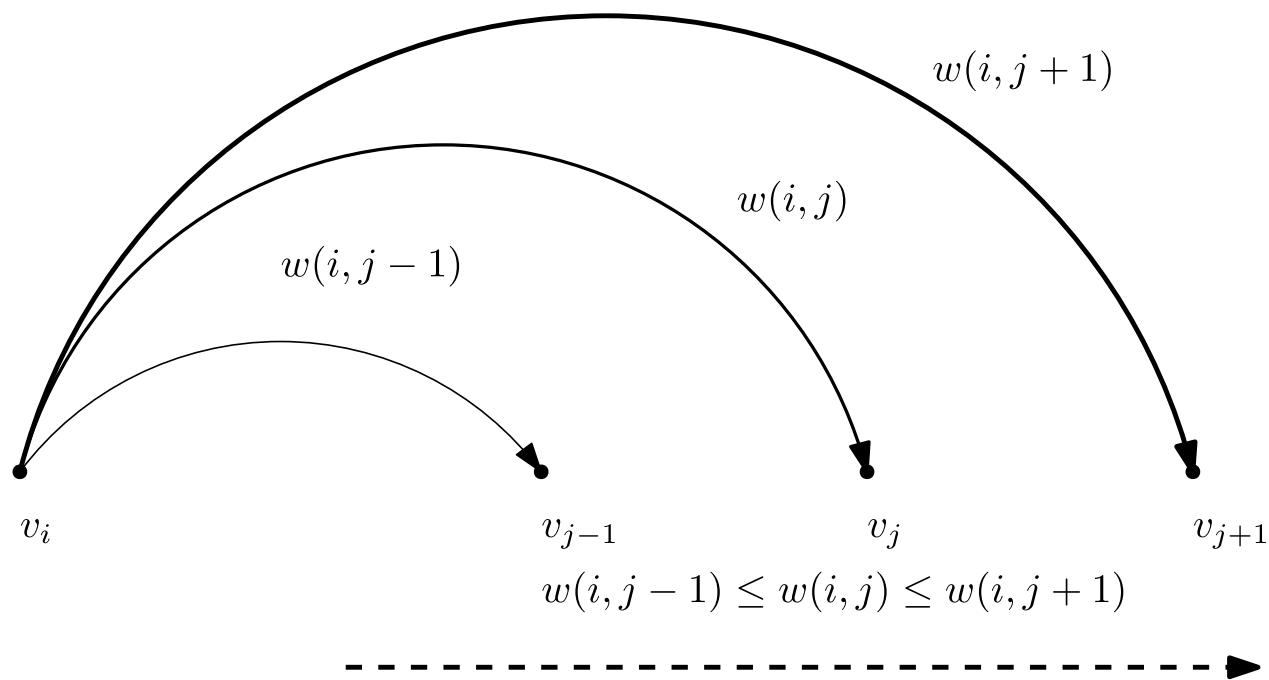
$$\mathcal{C}(T[i, j]) \geq \mathcal{C}(T[i, j - 1])$$

$$\mathcal{C}(T[i, j]) \geq \mathcal{C}(T[i + 1, j])$$

- We can compute the size of the compressed output incrementally: computing $\mathcal{C}(T[i, j])$ from $\mathcal{C}(T[i - 1, j])$ or $\mathcal{C}(T[i, j - 1])$ take constant time

Monotonicity of w

Due to the monotonicity of the compressor for every node $1 \leq i < k < j \leq n + 1$ we have that $w(i, k) \leq w(i, j)$



Sparsification of the DAG

Thanks to this property we can obtain an approximated algorithm by **sparsifying** the graph thus selecting only some edges.

We are able to obtain a $(1 + \varepsilon)$ -approximation, for every $\varepsilon \geq 0$, with a time complexity of $O(n \log_{1+\varepsilon} L)$

where $L = w(1, n)$, so the cost of compressing the entire text.

This algorithm can be applied to every dynamic programming algorithm in the form $E[j] = \min_{1 \leq i < j} (E[i] + w(i, j))$ when w is *monotone*!

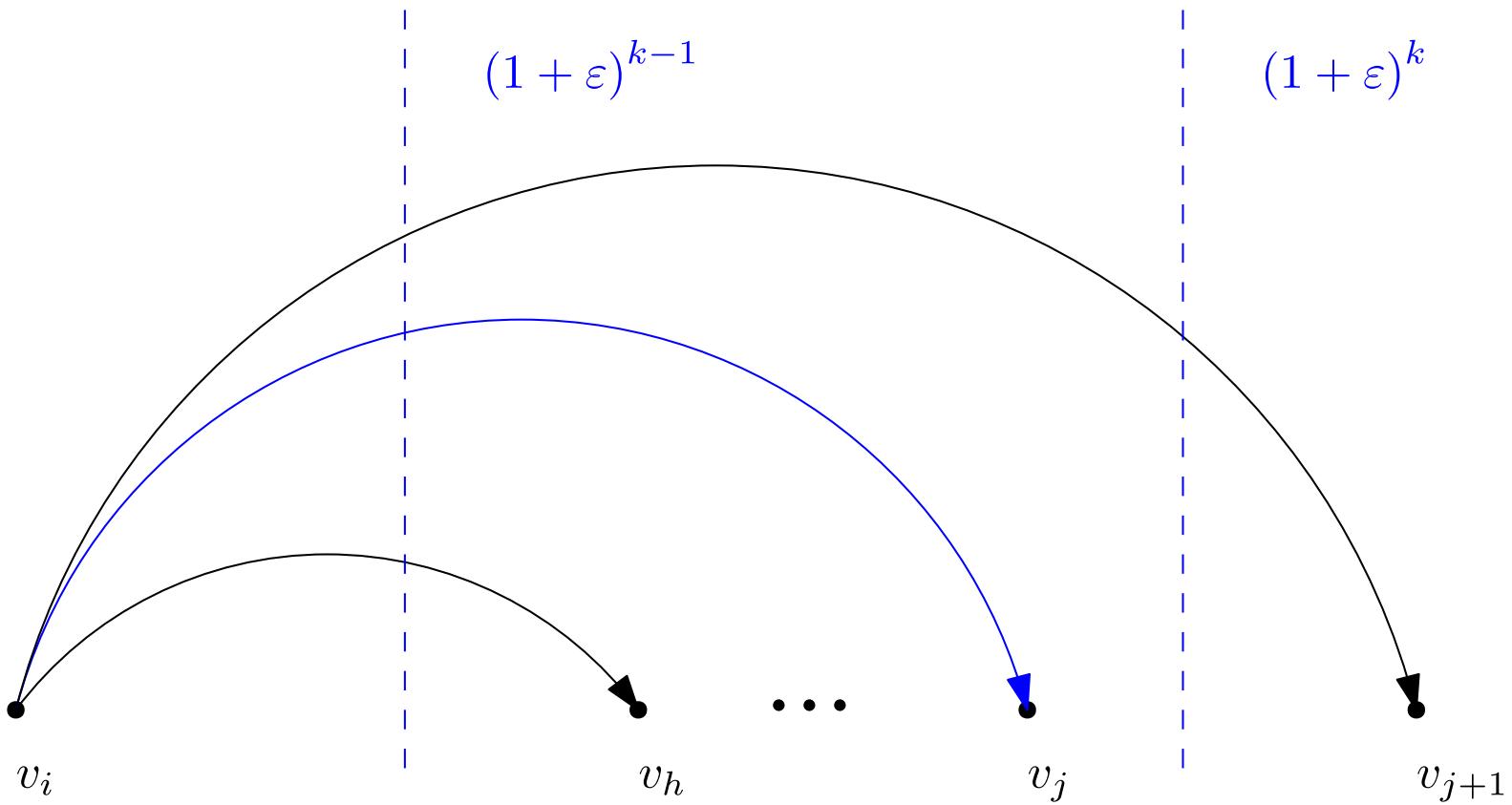
Key Idea: ε -maximal edges

How we can select some edges to obtain the $(1 + \varepsilon)$ approximation factor?

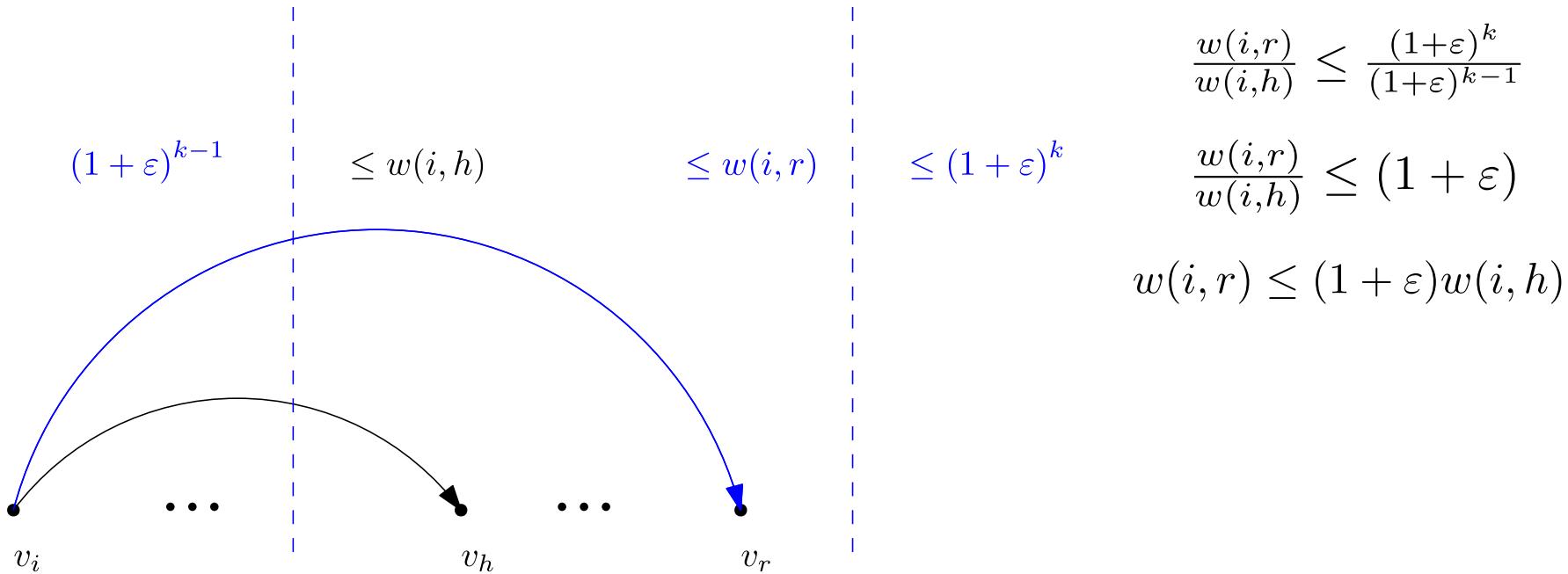
For each node i select the ε -maximal edges, so the outgoing edge from i that satisfy one of these conditions:

- The edges (i, j) such that $w(i, j) \leq (1 + \varepsilon)^k < w(i, j + 1)$ for any integer $k \geq 1$
- The last outgoing edge: $(i, n + 1)$

So we select the best approximations of the powers of $(1 + \varepsilon)$ from below: We then have at most $\log_{1+\varepsilon} L$ outgoing edges for each node.



Each edge is then “covered” by an ε -maximal edge: The weight of the edge is then approximated by $(1 + \varepsilon)$ times the weight of the maximal edge that covers it.



Lemma 1

Let $d_{\mathcal{G}}(i)$ be the shortest path in our graph \mathcal{G} from v_i to v_{n+1} then

For all the vertices $i, j : 1 \leq i < j \leq n + 1$, $d_{\mathcal{G}}(i) \geq d_{\mathcal{G}}(j)$

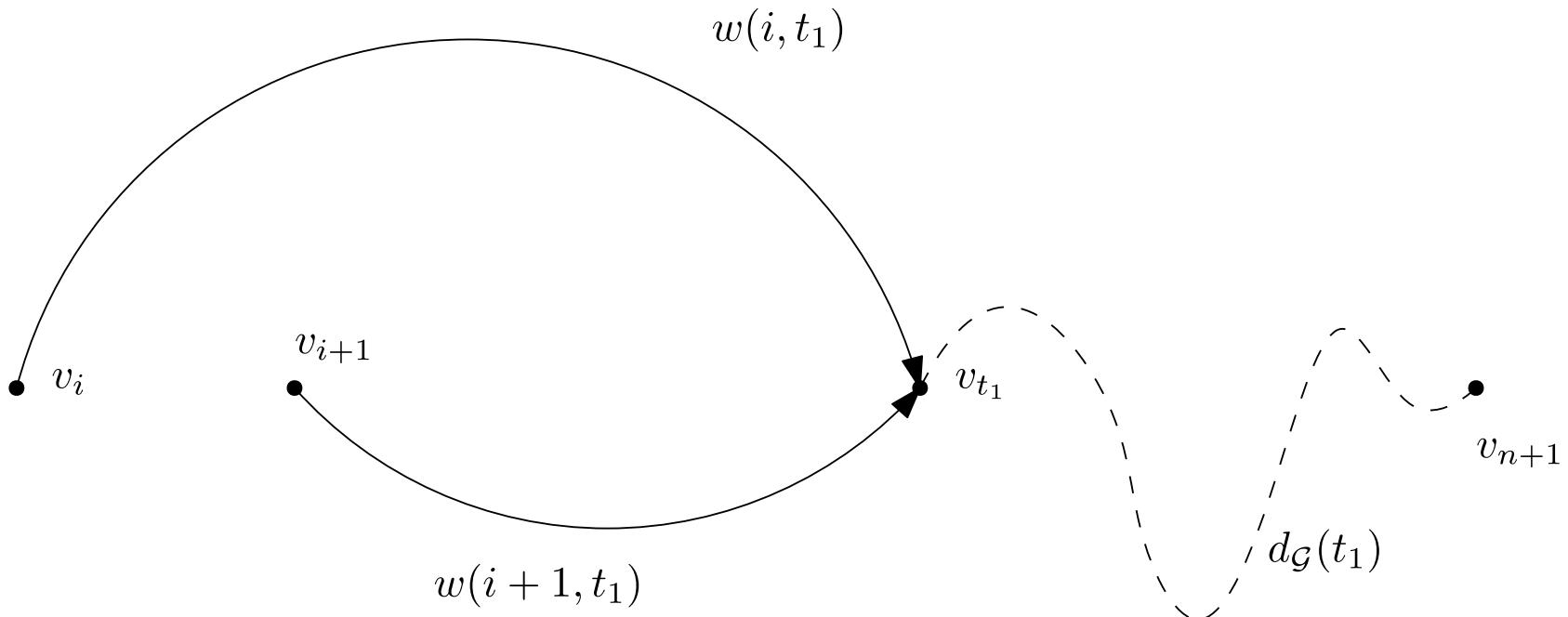
Proof by induction:

- Base, trivial case for $n + 1$
- Then we need to show that $d_{\mathcal{G}}(i) \geq d_{\mathcal{G}}(i + 1)$

Let $d_{\mathcal{G}}(i)$ be $(v_i, v_{t_1})(v_{t_1}, v_{t_2}) \dots (v_{t_k}, v_{n+1})$

- Trivial if $t_1 = i + 1$

- If $t_1 > i + 1$ then we can construct a shortest path $(v_{i+1}, v_{t_1})(v_{t_1}, v_{t_2}) \dots (v_{t_k}, v_{n+1})$ because thanks to the definition of *monotonicity* we know that $w(i, t_1) \geq w(i + 1, t_1)$



Theorem

Let \mathcal{G}_ε be the graph containing only ε -maximal edges, then $d_{\mathcal{G}_\varepsilon}(i) \leq (1 + \varepsilon)d_{\mathcal{G}}(i)$ for every $1 \leq i \leq n + 1$.

Proof by induction:

- Base, trivial case for $n + 1$
- Then let $d_{\mathcal{G}}(i) = (v_i, v_{t_1})..(v_{t_h}, v_n) = w(i, t_1) + d_{\mathcal{G}}(t_1)$. We choose the ε -maximal node r that covers t_1 : So $r > t_1$ and we already know that $w(i, r) \leq (1 + \varepsilon)w(i, t_1)$.

By *Lemma 1*: $d_{\mathcal{G}}(r) \leq d_{\mathcal{G}}(t_1)$

By inductive hypothesis $d_{\mathcal{G}_\varepsilon}(r) \leq (1 + \varepsilon)d_{\mathcal{G}}(r) \leq (1 + \varepsilon)d_{\mathcal{G}}(t_1)$

In the end $d_{\mathcal{G}_\varepsilon}(i) = w(i, r) + d_{\mathcal{G}_\varepsilon}(r) \leq (1 + \varepsilon)(w(i, t_1) + d_{\mathcal{G}}(t_1))$

Problem: DAG Construction

We still have two problems:

1. if we construct naively this graph we should remove edges from a $O(n^2)$ graph
2. We should compute the weight of the graph

We can solve both these problems efficiently at once!

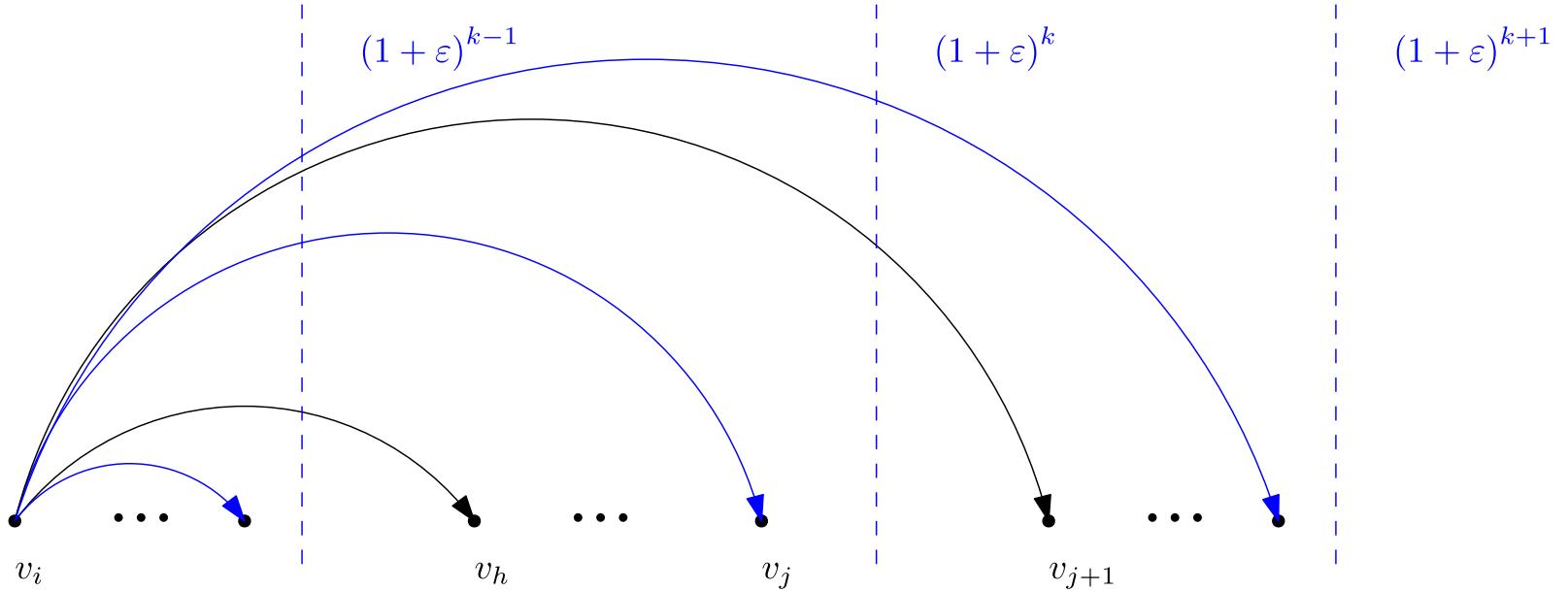
Sliding windows

We keep $\log_{1+\varepsilon} L$ sliding windows all starting at v_i , but ending in a different position. The h -th window advance on the right to reach the last edge smaller than the h -power of $(1 + \varepsilon)$ so finding the h -th maximal edge starting from node i .

After completing the dynamic programming step for the node i we advance all the windows from the start.

For each compressor we should implement 2 operations `advance_left`, `advance_right`: if they had respectively a complexity of $O(L)$ and $O(R)$ our algorithm execute asymptotically $O(nL + n \log_{1+\varepsilon} R)$ steps

The authors provide several implementations of the sliding windows approach to estimate the size of different compressors, among the others statistical compressors (using 0-th order and k-order entropy)



w_{k-1}

w_k

w_{k+1}

Thank You!