Efficient Succinct Data Structures on Directed Acyclic Graphs

Tesi Triennale in Matematica

Luca Lombardo

9 Maggio 2025

Dipartimento di Matematica
Università di Pisa

The Subset Membership Problem

Querying Collections Efficiently and Compactly

Consider a large universe of items $U = \{1, \ldots, n\}$, and a specific subset $S \subseteq U$ of m items.

Core Task & Desired Properties

- Quickly answer: "Is item x in S?" (**Membership Query**)
- Store S using minimal space (**Compact Representation**)

The Subset Membership Problem

Querying Collections Efficiently and Compactly

Consider a large universe of items $U = \{1, \ldots, n\}$, and a specific subset $S \subseteq U$ of m items.

Core Task & Desired Properties

- Quickly answer: ”Is item x in S?” (**Membership Query**)
- Store S using minimal space (**Compact Representation**)

To understand ”minimal space”, we turn to Information Theory. *What is the least number of bits needed to uniquely identify S?*

Definition (Shannon Entropy $H(X)$)

The average uncertainty, or information content, per symbol of source X:

$$H(X) = E_{P_X}[-\log_2 P_X(x)] = -\sum_{x \in \mathcal{X}} P_X(x) \log_2 P_X(x) \quad [\text{bits/symbol}]$$

Information-Theoretic Limits for Subsets

From General Entropy to Specific Subsets

We usually don't know the true source P_X. We only have the data sequence S.

Information-Theoretic Limits for Subsets

From General Entropy to Specific Subsets

We usually don't know the true source P_X. We only have the data sequence S.

Definition (Zero-Order Empirical Entropy $\mathcal{H}_0(S)$)

Information content of sequence S based on its symbol counts (n_s):

$$\mathcal{H}_0(S) = \sum_{s \in \Sigma} \frac{n_s}{n} \log_2 \frac{n}{n_s} \quad \text{[bits/symbol]}$$

Uses observed frequencies $\frac{n_s}{n}$ instead of unknown $P_X(x)$.

Information-Theoretic Limits for Subsets

From General Entropy to Specific Subsets

We usually don't know the true source P_X. We only have the data sequence S.

Definition (Zero-Order Empirical Entropy $\mathcal{H}_0(S)$)

Information content of sequence S based on its symbol counts (n_s):

$$\mathcal{H}_0(S) = \sum_{s \in \Sigma} \frac{n_s}{n} \log_2 \frac{n}{n_s} \quad \text{[bits/symbol]}$$

Uses observed frequencies $\frac{n_s}{n}$ instead of unknown $P_X(x)$.

For our subset S of m items from n:

- There are $\binom{n}{m}$ such distinct subsets.
- To uniquely identify one, we need at least $\lceil \log_2 \binom{n}{m} \rceil$ bits. This is the information content of specifying the subset.

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*

1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*

1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

If B is not explicit, how do we access $B[i]$? We use two foundational queries:

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*

1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

If B is not explicit, how do we access $B[i]$? We use two foundational queries:

- **rank**$_b(B, i)$: How many bits b are in the prefix $B[1..i]$?

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*

1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

$\text{rank}_1(15) = 9$

If B is not explicit, how do we access $B[i]$? We use two foundational queries:

- **rank**$_b(B, i)$: How many bits b are in the prefix $B[1..i]$?

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*

1	0	1	1	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

$\mathsf{rank}_1(15) = 9$

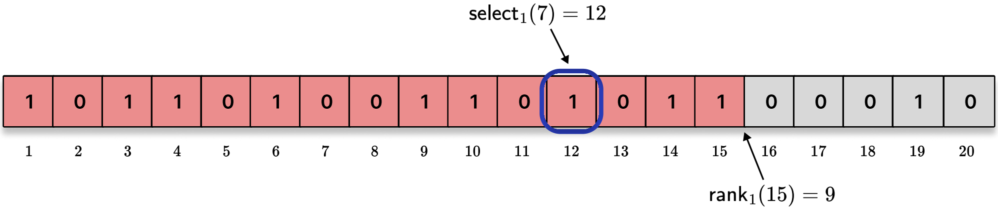
If B is not explicit, how do we access $B[i]$? We use two foundational queries:

- **rank**$_b(B, i)$: How many bits b are in the prefix $B[1..i]$?
- **select**$_b(B, j)$: What is the position of the j-th occurrence of bit b?

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*



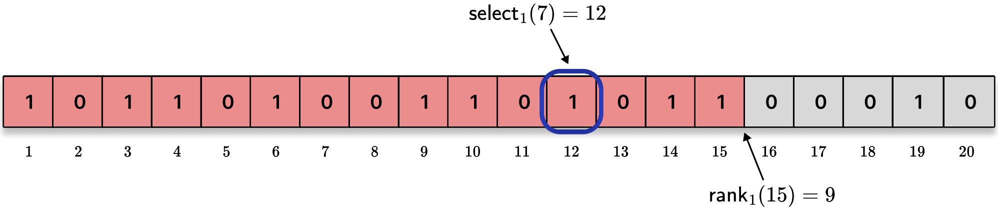
If B is not explicit, how do we access $B[i]$? We use two foundational queries:

- **rank**$_b(B, i)$: How many bits b are in the prefix $B[1..i]$?
- **select**$_b(B, j)$: What is the position of the j-th occurrence of bit b?

Bitvectors: Querying the Implicit Representation

From Compact Storage to Element Access

We can represent our subset S as a **bitvector** $B[1..n]$ ($B[i] = 1 \iff i \in S$). We are **encoding the choice of m positions for the '1's**, allowing us to store B using $\approx \lceil \log_2 \binom{n}{m} \rceil$ bits. *This means B is not stored as an explicit array of n bits.*



If B is not explicit, how do we access $B[i]$? We use two foundational queries:

Access Queries

$B[i] = 1 \iff \text{rank}_1(B, i) > \text{rank}_1(B, i-1)$

- **rank**$_b(B, i)$: How many bits b are in the prefix $B[1..i]$?
- **select**$_b(B, j)$: What is the position of the j-th occurrence of bit b?

RRR Structure: The Bitvector Solution

$n\mathcal{H}_0(B)$ Space & $O(1)$ Queries

Succinct Data Structure for Bitvectors

- **Goal:** Support rank and select in $O(1)$ time.
- **Space:** Close to the information-theoretic minimum for the bitvector.

RRR Structure: The Bitvector Solution

$n\mathcal{H}_0(B)$ Space & $O(1)$ Queries

Succinct Data Structure for Bitvectors

- **Goal:** Support rank and select in $O(1)$ time.
- **Space:** Close to the information-theoretic minimum for the bitvector.

Theorem (RRR Structure)

A bitvector $B[1..n]$ with m set bits can be represented using

$$B(n, m) + o(n) + O(\log \log n) \quad \textit{bits},$$

where $B(n, m) = \lceil \log_2 \binom{n}{m} \rceil$, while supporting rank and select queries in $O(1)$ time.

RRR Structure: The Bitvector Solution

$n\mathcal{H}_0(B)$ Space & $O(1)$ Queries

Succinct Data Structure for Bitvectors

- **Goal:** Support rank and select in $O(1)$ time.
- **Space:** Close to the information-theoretic minimum for the bitvector.

Theorem (RRR Structure)

A bitvector $B[1..n]$ with m set bits can be represented using

$$B(n,m) + o(n) + O(\log\log n) \quad \textit{bits},$$

where $B(n,m) = \lceil \log_2 \binom{n}{m} \rceil$, while supporting rank and select queries in $O(1)$ time.

A Cornerstone Result

RRR shows that **optimal space** *and* **efficient queries** are possible for subsets.

Why Succinct Data Structures?

The General Challenge & Goal

RRR is a specific solution to a general problem:

Massive Data & Auxiliary Structures Overhead

Modern datasets (Science, Web, AI...) are enormous. Complex analysis demands data in RAM, but auxiliary structures (indexes, trees) needed for queries often **occupy more space than the data itself**. $\Longrightarrow$ Fitting everything in RAM is a major bottleneck.

Why Succinct Data Structures?

The General Challenge & Goal

RRR is a specific solution to a general problem:

Massive Data & Auxiliary Structures Overhead

Modern datasets (Science, Web, AI...) are enormous. Complex analysis demands data in RAM, but auxiliary structures (indexes, trees) needed for queries often **occupy more space than the data itself**. $\Longrightarrow$ Fitting everything in RAM is a major bottleneck.

Classic Trade-off:

- *Compression:* Minimal space, but slow/no direct queries.
- *Traditional Data Structures:* Fast queries, but large space overhead.

Why Succinct Data Structures?

The General Challenge & Goal

RRR is a specific solution to a general problem:

Massive Data & Auxiliary Structures Overhead

Modern datasets (Science, Web, AI...) are enormous. Complex analysis demands data in RAM, but auxiliary structures (indexes, trees) needed for queries often **occupy more space than the data itself**. $\Longrightarrow$ Fitting everything in RAM is a major bottleneck.

Classic Trade-off:

- *Compression*: Minimal space, but slow/no direct queries.
- *Traditional Data Structures*: Fast queries, but large space overhead.

The Succinct Goal: Best of Both Worlds

Can we achieve **both**?

- Space near information-theoretic minimum.
- Efficient queries directly on compact data.

Beyond Bitvectors: General Alphabets

Wavelet Trees

What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?

Beyond Bitvectors: General Alphabets

Wavelet Trees

What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?

`abracadabra`　　　　`{a, b, c, d, r}`

Beyond Bitvectors: General Alphabets

Wavelet Trees

What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?

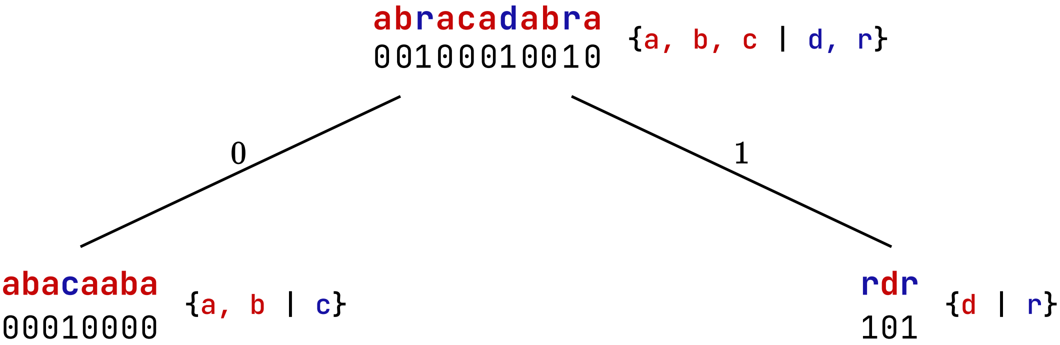
```
abracadabra
00100010010
```

{a, b, c | d, r}

Beyond Bitvectors: General Alphabets

Wavelet Trees

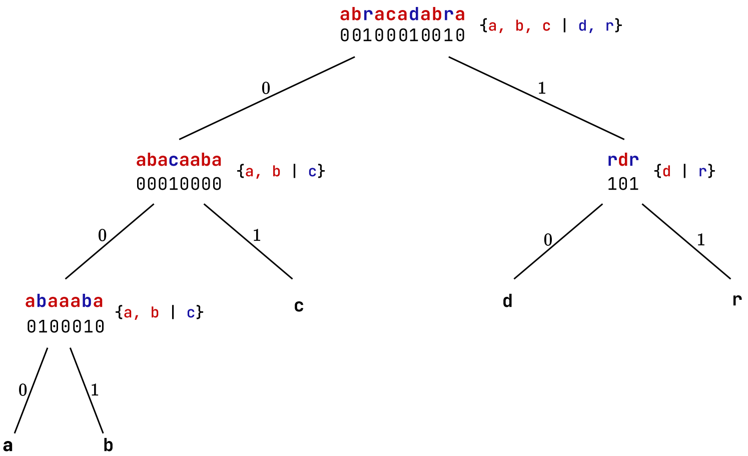
What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?



Beyond Bitvectors: General Alphabets

Wavelet Trees

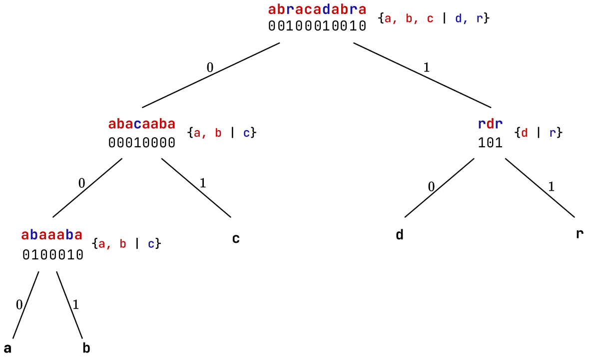
What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?



Beyond Bitvectors: General Alphabets

Wavelet Trees

What about sequences $S[1..n]$ over larger alphabets $\Sigma = \{1, \ldots, \sigma\}$?



$\mathcal{H}_0$-Compressed Wavelet Tree

Using RRR for bitvectors:

- **Space**: $n\mathcal{H}_0(S) + o(n \log \sigma)$ bits.
- **Query Time**: $O(\log \sigma)$ for access, rank$_c$, select$_c$.

Adapts space to the sequence's zero-order entropy.

Representing Sequence Variation: Degenerate Strings

Definitions and Core Operations

Definition and Rank & Select Adaptation

A **degenerate string** is a sequence $X = X_1X_2 \ldots X_n$, where each X_i is a *subset* of the alphabet Σ with cardinality σ. We can define the following operations:

- subset-rank$_X(i, c)$: Counts sets X_k $(k \leq i)$ where $c \in X_k$.
- subset-select$_X(j, c)$: Finds index k of the j-th set X_k where $c \in X_k$.

Representing Sequence Variation: Degenerate Strings

Definitions and Core Operations

Definition and Rank & Select Adaptation

A **degenerate string** is a sequence $X = X_1 X_2 \ldots X_n$, where each X_i is a *subset* of the alphabet Σ with cardinality σ. We can define the following operations:

- subset-rank$_X(i, c)$: Counts sets X_k ($k \leq i$) where $c \in X_k$.
- subset-select$_X(j, c)$: Finds index k of the j-th set X_k where $c \in X_k$.

$$X = \underbrace{\begin{Bmatrix} \texttt{A} \\ \texttt{C} \\ \texttt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \texttt{A} \\ \texttt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \texttt{C} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \texttt{T} \\ \texttt{G} \end{Bmatrix}}_{X_4}$$

$$\begin{array}{rlllll} S = & \texttt{ACG} & \texttt{AT} & \texttt{C} & \texttt{TG} & \\ R = & \texttt{100} & \texttt{10} & \texttt{1} & \texttt{10} & \texttt{1} \\ & S_1 & S_2 & S_3 & S_4 & \end{array}$$

Representing Sequence Variation: Degenerate Strings

Definitions and Core Operations

Definition and Rank & Select Adaptation

A **degenerate string** is a sequence $X = X_1 X_2 \ldots X_n$, where each X_i is a *subset* of the alphabet Σ with cardinality σ. We can define the following operations:

- subset-rank$_X(i, c)$: Counts sets X_k ($k \leq i$) where $c \in X_k$.
- subset-select$_X(j, c)$: Finds index k of the j-th set X_k where $c \in X_k$.

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{C} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

$$\begin{array}{rlllll} S = & \mathtt{ACG} & \mathtt{AT} & \mathtt{C} & \mathtt{TG} & \\ R = & \mathtt{100} & \mathtt{10} & \mathtt{1} & \mathtt{10} & \mathtt{1} \\ & S_1 & S_2 & S_3 & S_4 & \end{array}$$

To compute subset-rank$_X(i, c)$: Let p be the end position in S for prefix $X_1..X_i$, found using select$_1(R, i+1)$. The result is rank$_c(S, p)$.

From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string X:

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{C} \\ \mathtt{A} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string X:

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{C} \\ \mathtt{A} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

Key Idea

We can model queries on X by constructing a specific node-weighted Directed Acyclic Graph (DAG) G_c for a target character c.

From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string X:

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{C} \\ \mathtt{A} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

Key Idea

We can model queries on X by constructing a specific node-weighted Directed Acyclic Graph (DAG) G_c for a target character c.

- **Vertices** V_c: Source s + one node $v_{k,a}$ for each character $a \in X_k$ at each position k.

From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string X:

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{C} \\ \mathtt{A} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

Key Idea

We can model queries on X by constructing a specific node-weighted Directed Acyclic Graph (DAG) G_c for a target character c.

- **Vertices** V_c: Source s + one node $v_{k,a}$ for each character $a \in X_k$ at each position k.
- **Weights** w_c: $w_c(s) = 0$. $w_c(v_{k,a}) = 1$ if $a = c$, else 0. (Highlights occurrences of c).

From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string X:

$$X = \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{C} \\ \mathtt{G} \end{Bmatrix}}_{X_1} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{T} \end{Bmatrix}}_{X_2} \underbrace{\begin{Bmatrix} \mathtt{T} \\ \mathtt{C} \\ \mathtt{A} \end{Bmatrix}}_{X_3} \underbrace{\begin{Bmatrix} \mathtt{A} \\ \mathtt{G} \end{Bmatrix}}_{X_4}$$

Key Idea

We can model queries on X by constructing a specific node-weighted Directed Acyclic Graph (DAG) G_c for a target character c.

- **Vertices** V_c: Source s + one node $v_{k,a}$ for each character $a \in X_k$ at each position k.
- **Weights** w_c: $w_c(s) = 0$. $w_c(v_{k,a}) = 1$ if $a = c$, else 0. (Highlights occurrences of c).
- **Edges** E_c: Connect s to $k = 1$. Connect all $v_{k,a}$ to all $v_{k+1,b}$. (Represents sequence adjacency).

Path Weight Aggregation: The $\mathcal{O}$-Set

Capturing All Path Weights

Given a path $P = (v_0 = s, \ldots, v_k = v)$ we define $W(P) = \sum_{j=1}^{k} w(v_j)$

Goal

Characterize the set of all possible distinct cumulative path weights arriving at each node.

Path Weight Aggregation: The $\mathcal{O}$-Set

Capturing All Path Weights

Given a path $P = (v_0 = s, \ldots, v_k = v)$ we define $W(P) = \sum_{j=1}^{k} w(v_j)$

Goal

Characterize the set of all possible distinct cumulative path weights arriving at each node.

$\mathcal{O}$-Set Definition (Recursive)

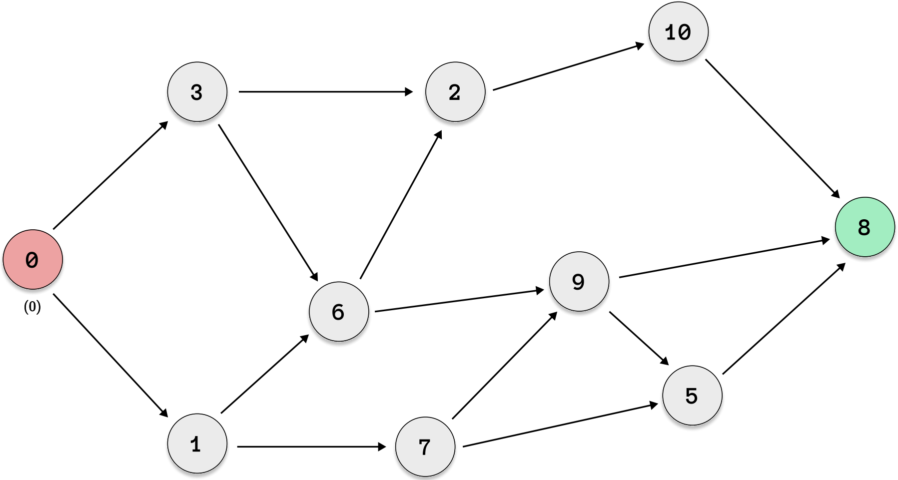
- **Base Case (Source):** $\mathcal{O}_s = \{0\}$
- **Recursive Step ($v \neq s$):**

$$\mathcal{O}_v = \bigcup_{u \in Pred(v)} \{\gamma + w(v) \mid \gamma \in \mathcal{O}_u\} = \{W(P) \mid P \in Path(s, v)\}$$

Keep only distinct values. Store as a sorted sequence.

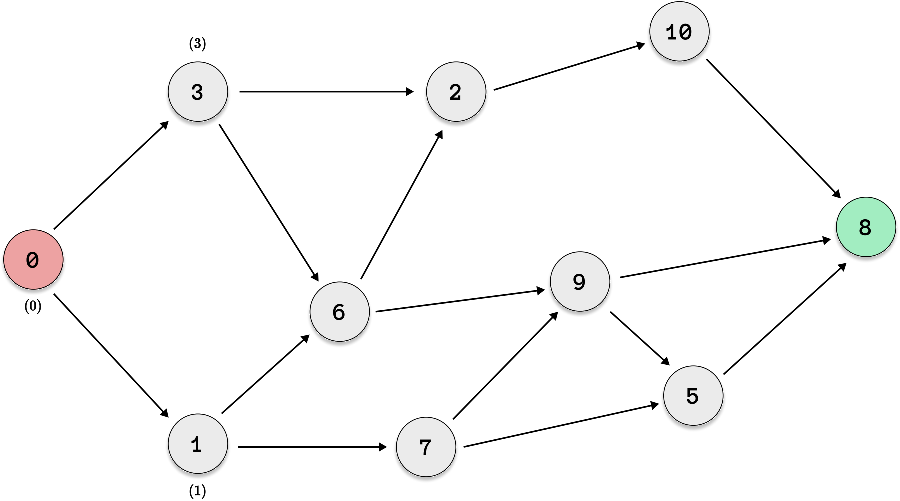
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



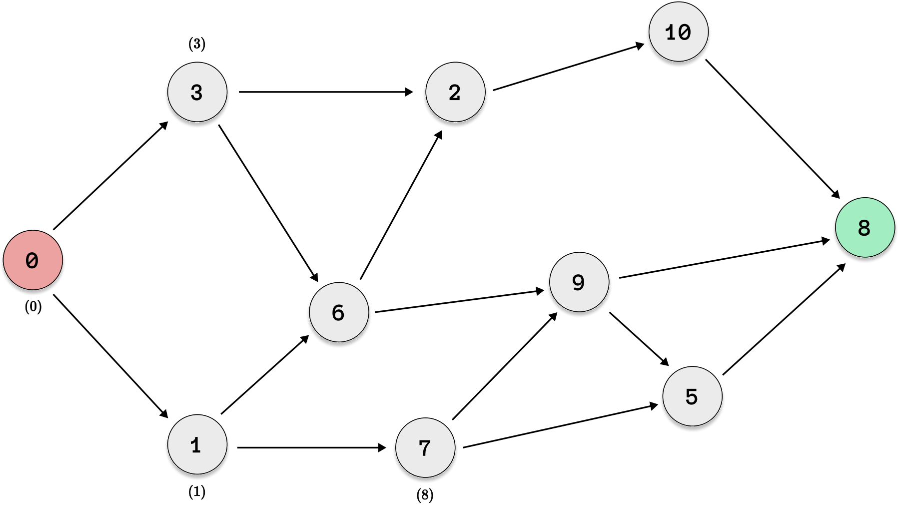
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



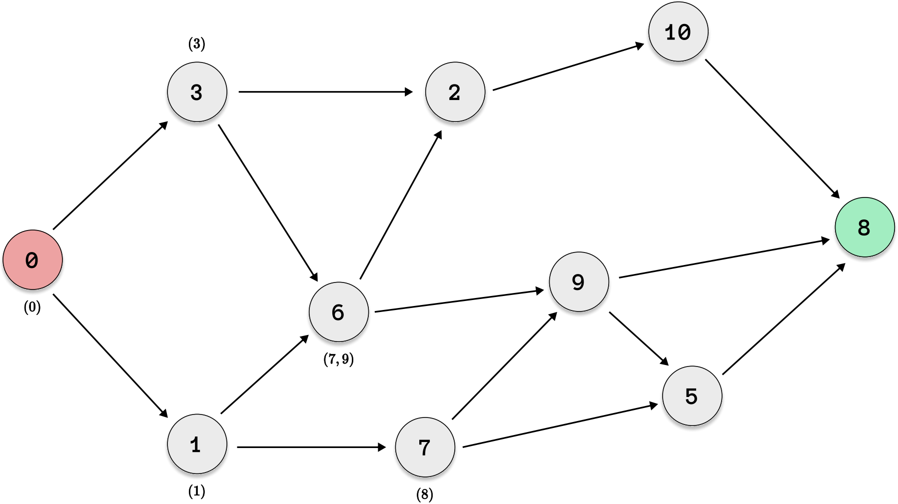
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



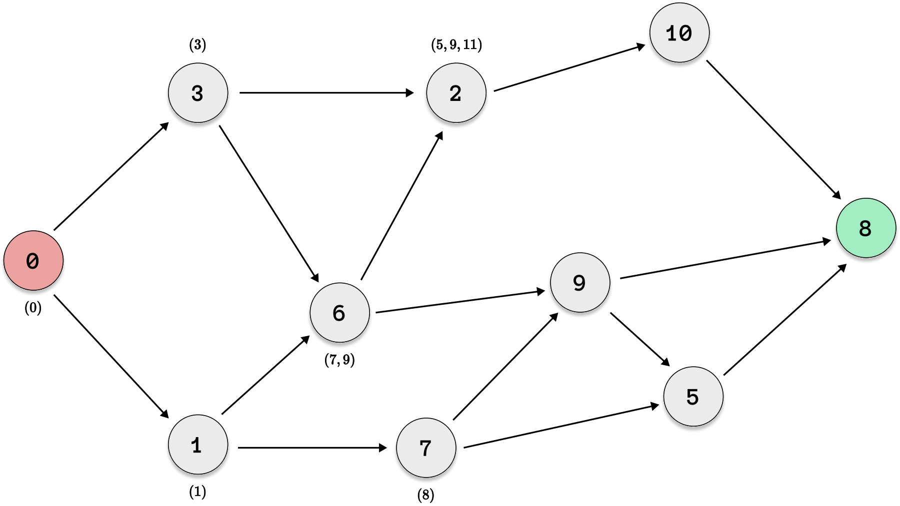
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



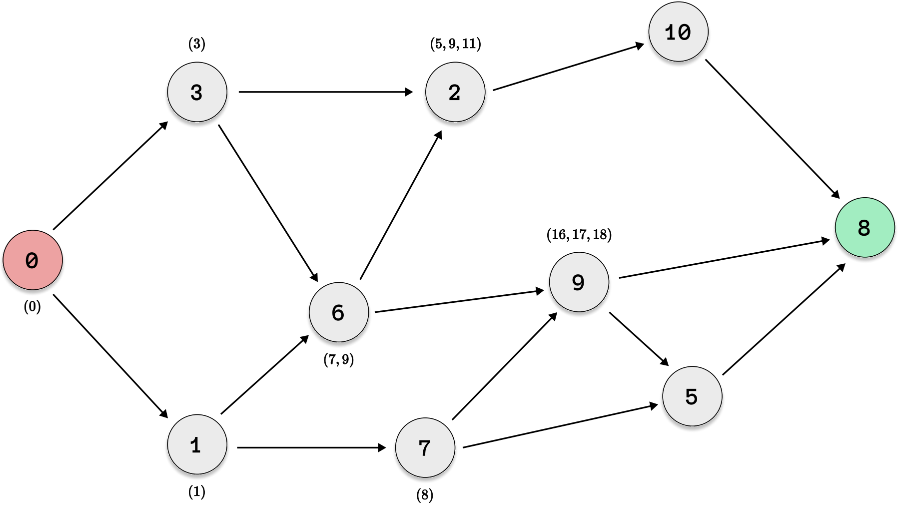
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



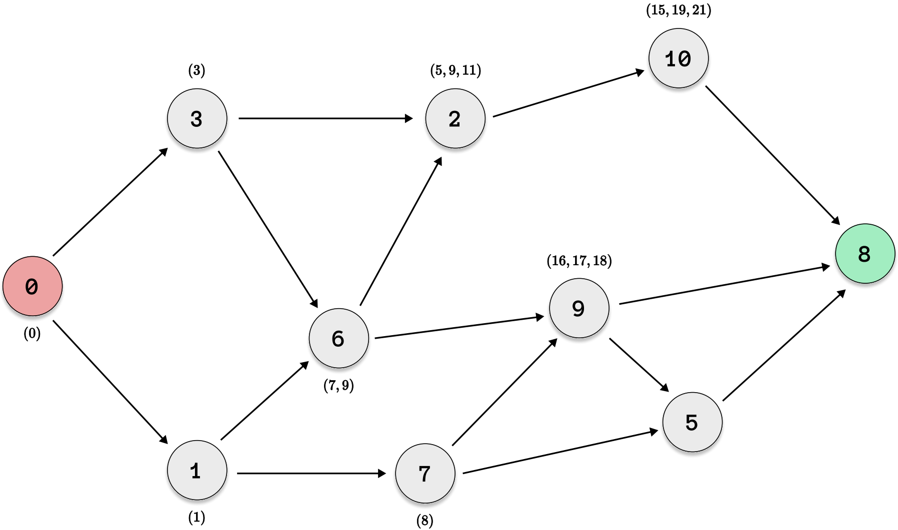
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



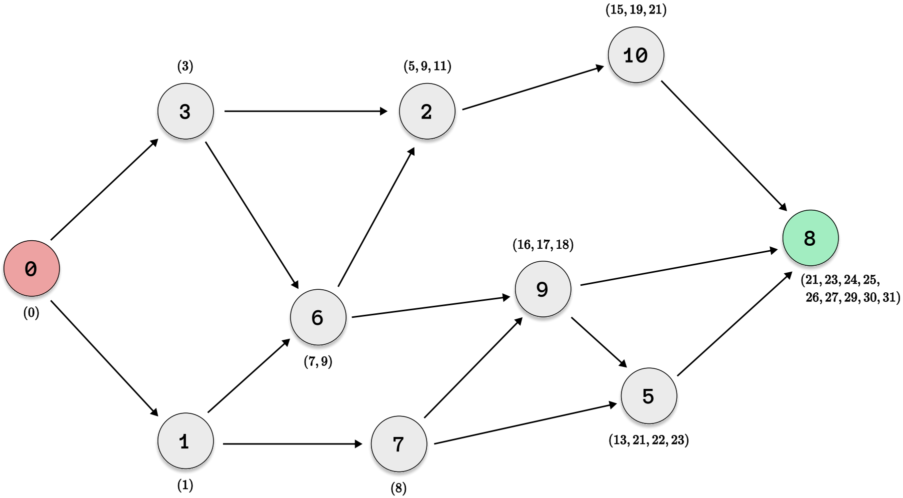
$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



$\mathcal{O}$-Set Construction Example

Visualizing the Recursive Definition



The Rank Query on Weighted DAGs

What Values are ”Active” at Node N?

Rank Query on a Node N: $\mathrm{rank}_G(N)$

1. Returns a representation of a set of integers derived from the $\mathcal{O}$-set $\mathcal{O}_N$.

$$S_N = \bigcup_{x \in \mathcal{O}_N} \{z \in \mathbb{N}_0 \mid \max(0, x - w(N) + 1) \leq z \leq x\}.$$

The Rank Query on Weighted DAGs

What Values are ”Active” at Node N?

Rank Query on a Node N: $\mathrm{rank}_G(N)$

1. Returns a representation of a set of integers derived from the $\mathcal{O}$-set $\mathcal{O}_N$.

$$S_N = \bigcup_{x \in \mathcal{O}_N} \{z \in \mathbb{N}_0 \mid \max(0, x - w(N) + 1) \leq z \leq x\}.$$

2. These intervals are then maximally merged. The query $\mathrm{rank}_G(N)$ returns a minimal collection of disjoint closed integer intervals

$$\mathcal{R}_N = \{[l_1, r_1], [l_2, r_2], \ldots, [l_p, r_p]\}$$

such that their union exactly covers S_N.

$\mathcal{R}_N$ captures the range of possible cumulative sums during the *activity* at node N

The Challenge: Storing Path Information

$\mathcal{O}$-Sets Can Be Huge!

- **Problem**: The size $|\mathcal{O}_v|$ can grow very large!
- **Question:** Can we represent the necessary information more compactly?

The Challenge: Storing Path Information

$\mathcal{O}$-Sets Can Be Huge!

- **Problem**: The size $|\mathcal{O}_v|$ can grow very large!
- **Question:** Can we represent the necessary information more compactly?

Core Idea: Partitioning + Indirection

Partition vertices V into two types:

1. Explicit Vertices (V_E)

Store $\mathcal{O}_v$ directly.
(*Simple, but potentially large*)

2. Implicit Vertices (V_I)

Do *not* store $\mathcal{O}_v$ explicitly
(*Reconstruct on-the-fly.*)

The Challenge: Storing Path Information

$\mathcal{O}$-Sets Can Be Huge!

- **Problem**: The size $|\mathcal{O}_v|$ can grow very large!
- **Question:** Can we represent the necessary information more compactly?

Core Idea: Partitioning + Indirection

Partition vertices V into two types:

1. Explicit Vertices (V_E)	2. Implicit Vertices (V_I)
Store $\mathcal{O}_v$ directly. (*Simple, but potentially large*)	Do *not* store $\mathcal{O}_v$ explicitly (*Reconstruct on-the-fly.*)

Reconstruction for $v \in V_I$ using:

- Designated Successor $\sigma(v)$
- Offset Sequence $\mathcal{J}_v$ (at v)

Implicit Reconstruction: Successor & Offset

How V_I Nodes Refer to Others

1. Designated Successor $\sigma(v)$ (for $v \in V_I$)

Which successor should v point to? **Heuristic**: Choose $u = \sigma(v)$ that minimizes $|\mathcal{O}_u|$.

$$\sigma(v) \in \operatorname*{argmin}_{u \in Succ(v)} \{|\mathcal{O}_u|\}.$$

2. Offset Sequence $\mathcal{J}_v$ (for $v \in V_I$)

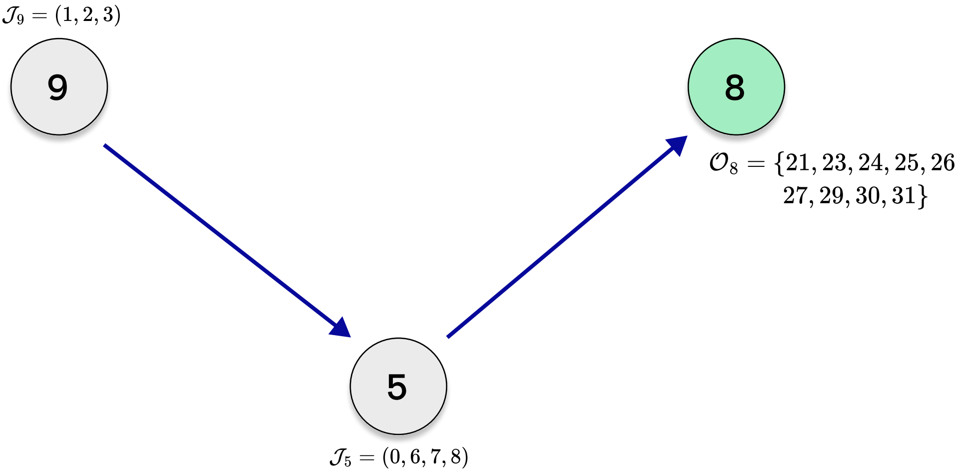
How to get $\mathcal{O}_v$ from $\mathcal{O}_{\sigma(v)}$? Let $u = \sigma(v)$.

- **Relationship**: Each element $x_k \in \mathcal{O}_v$ comes from some $y_{j_k} \in \mathcal{O}_u$ via $x_k = y_{j_k} - w(u)$.
- **Offset Sequence $\mathcal{J}_v$**: Stores the index j_k corresponding to each x_k.

$$\mathcal{J}_v = (j_0, j_1, \ldots, j_{m-1}), \quad \text{where } m = |\mathcal{O}_v|$$

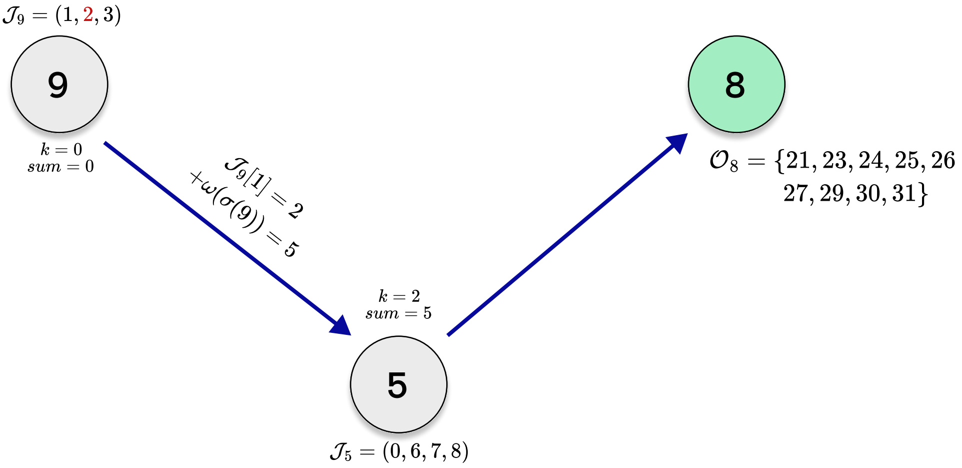
Example: Computing $\mathcal{O}_9[1]$

Following the Successor Path: $9 \to 5 \to 8$



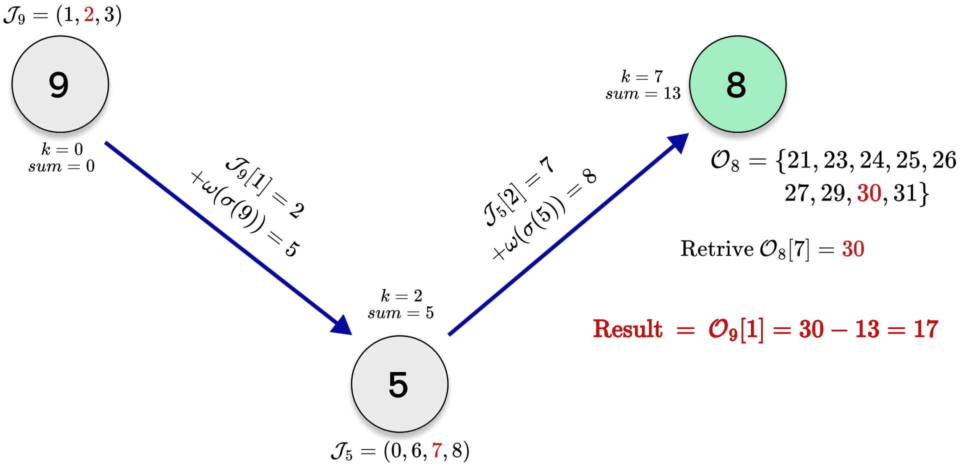
Example: Computing $\mathcal{O}_9[1]$

Following the Successor Path: $9 \to 5 \to 8$



Example: Computing $\mathcal{O}_9[1]$

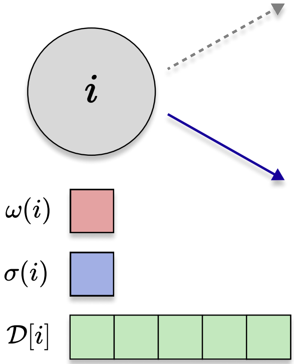
Following the Successor Path: $9 \rightarrow 5 \rightarrow 8$



Succinct Data Structure: Components

Arrays Indexed by Vertex ID

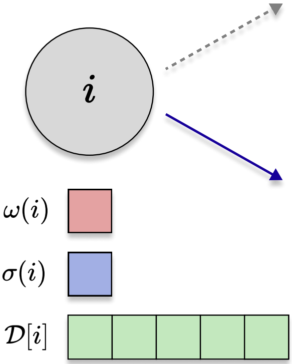
Each node stores 3 components



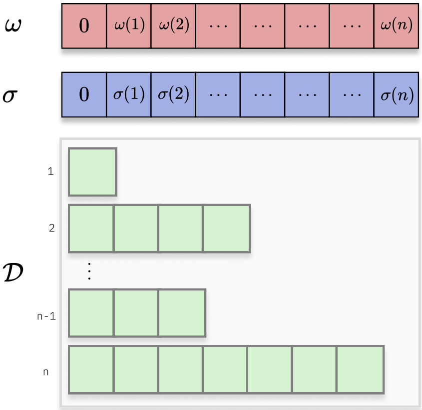
Succinct Data Structure: Components

Arrays Indexed by Vertex ID

Each node stores 3 components



Succinct DAG as a Struct of Arrays



Compression Strategies

Reducing Memory Footprint

Component	Description
$\mathcal{W}$ (Node weights)	Array of positive integers.
Σ (Successor IDs)	Array of positive integers.

Compression Strategies

Reducing Memory Footprint

Component	Description
$\mathcal{W}$ (Node weights)	Array of positive integers.
Σ (Successor IDs)	Array of positive integers.

Compression Options

- Variable-Length Integer Coding $\longrightarrow$ we published a Rust library[a] for this!
- Wavelet Trees *(for nodes with small weight range)*

[a] `https://crates.io/crates/compressed-intvec`

Compression Strategies

Reducing Memory Footprint

Component	Description
$\mathcal{W}$ (Node weights)	Array of positive integers.
Σ (Successor IDs)	Array of positive integers.
$\mathcal{D}$ (Associated Data)	Array of arrays of increasing positive integers.

Compression Options

- Variable-Length Integer Coding $\longrightarrow$ we published a Rust library[a] for this!
- Wavelet Trees *(for nodes with small weight range)*

[a] `https://crates.io/crates/compressed-intvec`

Compression Strategies

Reducing Memory Footprint

Component	Description
$\mathcal{W}$ (Node weights)	Array of positive integers.
Σ (Successor IDs)	Array of positive integers.
$\mathcal{D}$ (Associated Data)	Array of arrays of increasing positive integers.

Compression Options

- Variable-Length Integer Coding $\longrightarrow$ we published a Rust library[a] for this!
- Wavelet Trees *(for nodes with small weight range)*
- Elias-Fano Encoding *(for monotonic sequences)*
- Run-Length Encoding (RLE) *(for clustered monotonic sequences)*

[a] `https://crates.io/crates/compressed-intvec`

Space Efficiency: Baseline Comparison

How Much Information is in the Graph?

To evaluate our structure's space, we need a baseline.

0^{th}-Order Graph Entropy $H_0(G)$

A theoretical lower bound for storing the *entire* weighted DAG (V, E, w) losslessly.

$$H_0(G) = \underbrace{H_W(G)}_{\text{Cost for Weights}} + \underbrace{H_E(G)}_{\text{Cost for Topology (Edges)}}$$

Space Efficiency: Baseline Comparison

How Much Information is in the Graph?

To evaluate our structure's space, we need a baseline.

0^{th}-Order Graph Entropy $H_0(G)$

A theoretical lower bound for storing the *entire* weighted DAG (V, E, w) losslessly.

$$H_0(G) = \underbrace{H_W(G)}_{\text{Cost for Weights}} + \underbrace{H_E(G)}_{\text{Cost for Topology (Edges)}}$$

- $H_W(G) \approx \sum_{v \in V} \log(w(v) + 1)$ bits (*Minimal binary encoding for weights*).

Space Efficiency: Baseline Comparison

How Much Information is in the Graph?

To evaluate our structure's space, we need a baseline.

0^{th}-Order Graph Entropy $H_0(G)$

A theoretical lower bound for storing the *entire* weighted DAG (V, E, w) losslessly.

$$H_0(G) = \underbrace{H_W(G)}_{\text{Cost for Weights}} + \underbrace{H_E(G)}_{\text{Cost for Topology (Edges)}}$$

- $H_W(G) \approx \sum_{v \in V} \log(w(v) + 1)$ bits (*Minimal binary encoding for weights*).
- $H_E(G) \approx \log \binom{n(n-1)}{m}$ bits (*Cost to choose $m = |E|$ edges out of all possible $n(n-1)$*).

Space Efficiency: Baseline Comparison

How Much Information is in the Graph?

To evaluate our structure's space, we need a baseline.

0^{th}-Order Graph Entropy $H_0(G)$

A theoretical lower bound for storing the *entire* weighted DAG (V, E, w) losslessly.

$$H_0(G) = \underbrace{H_W(G)}_{\text{Cost for Weights}} + \underbrace{H_E(G)}_{\text{Cost for Topology (Edges)}}$$

- $H_W(G) \approx \sum_{v \in V} \log(w(v) + 1)$ bits (*Minimal binary encoding for weights*).
- $H_E(G) \approx \log \binom{n(n-1)}{m}$ bits (*Cost to choose* $m = |E|$ *edges out of all possible* $n(n-1)$).

Any method saving the *full* graph structure needs at least $H_0(G)$ bits!

Space Comparison: Succinct Structure vs. Baselines

Bitcoin DAG Example ($n \approx 22k, m \approx 50k$)

Method	Estimated Bits
Theoretical Lower Bound	**1,525,730**
Weights $H_W(G)$	60,824
Topology $H_E(G)$	1,464,906
Precomputed Rank Queries:	
Explicit Binary Storage	
Elias-Fano Compressed	
Our Succinct DAG	
Weights $\mathcal{W}$	
Successors Σ	
Assoc. Data $\mathcal{D}$ (RLE)	

Space Comparison: Succinct Structure vs. Baselines

Bitcoin DAG Example ($n \approx 22k, m \approx 50k$)

Method	Estimated Bits
Theoretical Lower Bound	**1,525,730**
Weights $H_W(G)$	60,824
Topology $H_E(G)$	1,464,906
Precomputed Rank Queries:	
Explicit Binary Storage	4,854,533
Elias-Fano Compressed	2,211,849
Our Succinct DAG	
Weights $\mathcal{W}$	
Successors Σ	
Assoc. Data $\mathcal{D}$ (RLE)	

Space Comparison: Succinct Structure vs. Baselines

Bitcoin DAG Example ($n \approx 22k$, $m \approx 50k$)

Method	Estimated Bits
Theoretical Lower Bound	**1,525,730**
Weights $H_W(G)$	60,824
Topology $H_E(G)$	1,464,906
Precomputed Rank Queries:	
Explicit Binary Storage	4,854,533
Elias-Fano Compressed	2,211,849
Our Succinct DAG	**602,808**
Weights $\mathcal{W}$	60,824
Successors Σ	297,700
Assoc. Data $\mathcal{D}$ (RLE)	244,284

Space Comparison: Succinct Structure vs. Baselines

Bitcoin DAG Example ($n \approx 22k, m \approx 50k$)

Method	Estimated Bits
Theoretical Lower Bound	**1,525,730**
Weights $H_W(G)$	60,824
Topology $H_E(G)$	1,464,906
Precomputed Rank Queries:	
Explicit Binary Storage	4,854,533
Elias-Fano Compressed	2,211,849
Our Succinct DAG	**602,808**
Weights $\mathcal{W}$	60,824
Successors Σ	297,700
Assoc. Data $\mathcal{D}$ (RLE)	244,284

Achieving Sub-Entropy Space: How?

Our structure is **lossy** regarding the full graph topology:

- It does not store the complete edge set.
- It only stores the chosen successor $\sigma(v)$ for each implicit node (in Σ).

However, it is **lossless** for computing the specific Rank Query.

Future Direction: Bounded Query Time

Guaranteeing Predictable Performance

Performance Consideration

Query time for implicit node v depends on the length of the successor path

$$v \to \sigma(v) \to \sigma(\sigma(v)) \to \cdots \to e \in V_E$$

Problem: Can be large/variable in deep DAGs $\implies$ slow worst-case query time.

Future Direction: Bounded Query Time

Guaranteeing Predictable Performance

Performance Consideration

Query time for implicit node v depends on the length of the successor path

$$v \to \sigma(v) \to \sigma(\sigma(v)) \to \cdots \to e \in V_E$$

Problem: Can be large/variable in deep DAGs $\implies$ slow worst-case query time.

Solution, Challenges & Trade-offs

- **Solution:** Ensure every implicit node can reach an explicit node within k steps.
- **Challenges:** Finding the smallest possible V_E' that satisfies this condition is NP-hard (*minimum distance-k dominating set*).
- **Trade-off:** More explicit nodes $\implies$ faster queries, but larger space.

Efficient Succinct Data Structures on Directed Acyclic Graphs

Thank you for listening!

Worst-Case $\mathcal{O}$-Set Size: Is Exponential Growth Possible?

Understanding the $\mathcal{O}$-set Size

Exponential Growth Can Occur

The cardinality of an $\mathcal{O}$-set, $|\mathcal{O}_v|$, is not generally bounded by a polynomial in the number of vertices $|V|$. It can grow exponentially.

Underlying Reason: Path Count

The number of distinct paths from a source s to a vertex v, denoted $|Path(s, v)|$, can itself be exponential in certain DAG structures. Since $|\mathcal{O}_v| \leq |Path(s, v)|$, the potential for exponential size exists.

Worst-Case $\mathcal{O}$-Set Size: Is Exponential Growth Possible?

Understanding the $\mathcal{O}$-set Size

Exponential Growth Can Occur

The cardinality of an $\mathcal{O}$-set, $|\mathcal{O}_v|$, is not generally bounded by a polynomial in the number of vertices $|V|$. It can grow exponentially.

Underlying Reason: Path Count

The number of distinct paths from a source s to a vertex v, denoted $|Path(s, v)|$, can itself be exponential in certain DAG structures. Since $|\mathcal{O}_v| \leq |Path(s, v)|$, the potential for exponential size exists.

Key Condition for Exponential Growth

The exponential potential is realized if the vertex weights $w(v)$ are assigned such that distinct paths $P_1 \neq P_2$ almost always lead to distinct cumulative weights $W(P_1) \neq W(P_2)$.

Achieving Exponential $\mathcal{O}$-Set Size

A Strategy for Path Weight Uniqueness

Start with a DAG structure that naturally admits an exponential number of paths between two nodes. An example is a layered graph with multiple choices at each layer transition.

Strategic Weight Assignment

Assign vertex weights $w(v)$ carefully to ensure path weight uniqueness.

$$w(v) = 2^k \quad \text{(using a unique exponent } k \text{ for each node)}$$

Achieving Exponential $\mathcal{O}$-Set Size

A Strategy for Path Weight Uniqueness

Start with a DAG structure that naturally admits an exponential number of paths between two nodes. An example is a layered graph with multiple choices at each layer transition.

Strategic Weight Assignment

Assign vertex weights $w(v)$ carefully to ensure path weight uniqueness.

$$w(v) = 2^k \quad \text{(using a unique exponent } k \text{ for each node)}$$

Mechanism: Unique Binary Representation

With power-of-2 weights, the cumulative path weight $W(P) = \sum_{v \in P \setminus \{s\}} w(v)$ becomes a sum of distinct powers of 2. Due to the uniqueness of binary representation, different sets of nodes (i.e., different paths) produce different sums. Therefore, $|Path(s, v)|$ distinct paths yield $|\mathcal{O}_v| = |Path(s, v)|$ distinct weights.