



# Efficient Succinct Data Structures on Directed Acyclic Graphs

Tesi Triennale in Matematica

**Luca Lombardo**

9 Maggio 2025





# Why Succinct Data Structures?

Massive Data and Structure Overhead

## The Challenge: Massive Data & Query Needs

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.**  $\implies$  Fitting everything in RAM is a major bottleneck.



# Why Succinct Data Structures?

Massive Data and Structure Overhead

## The Challenge: Massive Data & Query Needs

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.**  $\implies$  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?

Massive Data and Structure Overhead

## The Challenge: Massive Data & Query Needs

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.**  $\implies$  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.



# Shannon Entropy

Fundamental Limits of Lossless Compression

**Goal:** Determine the minimum average number of bits per symbol required for a lossless representation of data from a source  $X$ .



# Shannon Entropy

Fundamental Limits of Lossless Compression

**Goal:** Determine the minimum average number of bits per symbol required for a lossless representation of data from a source  $X$ .

## Definition (Shannon Entropy $H(X)$ )

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

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



# Shannon Entropy

Fundamental Limits of Lossless Compression

**Goal:** Determine the minimum average number of bits per symbol required for a lossless representation of data from a source  $X$ .

## Definition (Shannon Entropy $H(X)$ )

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

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

## Source Coding Theorem (Lower Bound)

Shannon proved that  $H(X)$  constitutes the **theoretical lower bound** on the average number of bits per symbol required to represent the output of source  $X$  without loss of information.



# Zero-Order Empirical Entropy

A Practical Bound Based on Data

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



## Zero-Order Empirical Entropy

A Practical Bound Based on Data

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  $P_X(x)$ .



## Zero-Order Empirical Entropy

A Practical Bound Based on Data

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  $P_X(x)$ .

### Relevance for Succinct Structures

$n \cdot \mathcal{H}_0(S)$  is a practical space benchmark. Succinct data structures often target space close to this value (or higher-order versions  $\mathcal{H}_k$ ) for the **given sequence S**.



# Bitvectors and Fundamental Queries

The Simplest Sequence

Consider the most basic sequence: a **bitvector**  $B[1..n]$ , a sequence of  $n$  bits from  $\{0, 1\}$ .

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 and Fundamental Queries

The Simplest Sequence

Consider the most basic sequence: a **bitvector**  $B[1..n]$ , a sequence of  $n$  bits from  $\{0, 1\}$ .

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

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

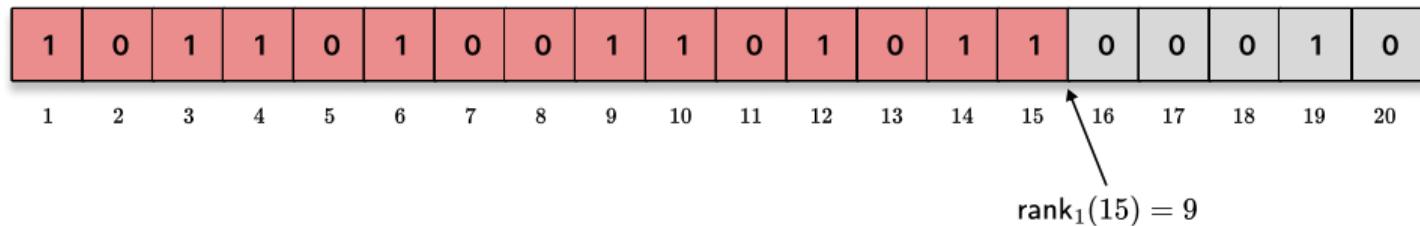


# Bitvectors and Fundamental Queries

The Simplest Sequence

Consider the most basic sequence: a **bitvector**  $B[1..n]$ , a sequence of  $n$  bits from  $\{0, 1\}$ .

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



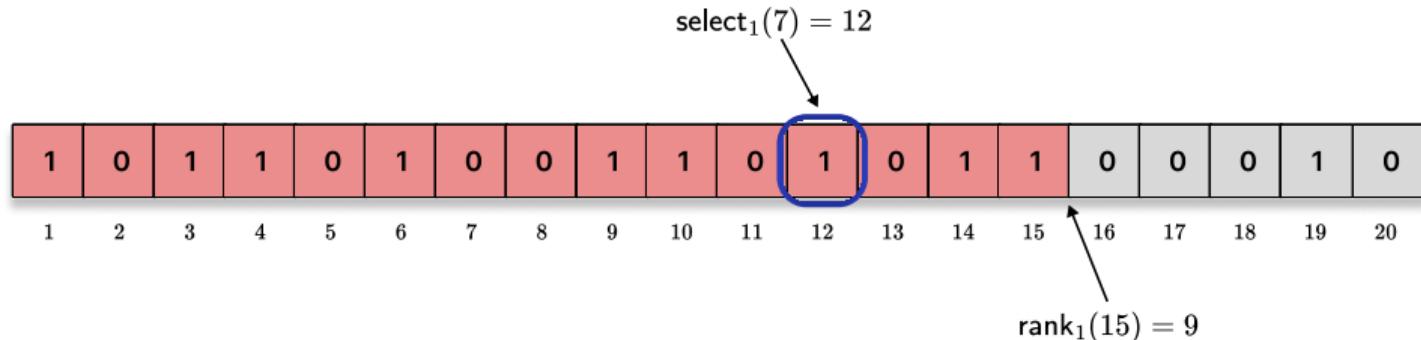


# Bitvectors and Fundamental Queries

The Simplest Sequence

Consider the most basic sequence: a **bitvector**  $B[1..n]$ , a sequence of  $n$  bits from  $\{0, 1\}$ .

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



Furthermore, rank and select are inverse operations:

$$\text{rank}_b(B, \text{select}_b(B, j)) = j \quad \text{and} \quad \text{select}_b(B, \text{rank}_b(B, i)) = i.$$



# RRR Structure: Entropy-Compressed Bitvectors

Achieving Space Close to Empirical Entropy

## 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: Entropy-Compressed Bitvectors

Achieving Space Close to Empirical Entropy

## 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

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

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

$B(n, m) \approx n\mathcal{H}_0(B)$  is the **information-theoretic minimum space** required to store an arbitrary subset of size  $m$  from a universe of size  $n$



# Beyond Bitvectors: General Alphabets

Wavelet Trees

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



# Beyond Bitvectors: General Alphabets

Wavelet Trees

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

**abracadabra**

{a, b, c, d, r}



# Beyond Bitvectors: General Alphabets

Wavelet Trees

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

**abracadabra**  
00100010010

{a, b, c | d, r}



## Beyond Bitvectors: General Alphabets

Wavelet Trees

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

**abracadabra**  
00100010010     $\{a, b, c \mid d, r\}$



**abacaaba**  
00010000     $\{a, b \mid c\}$

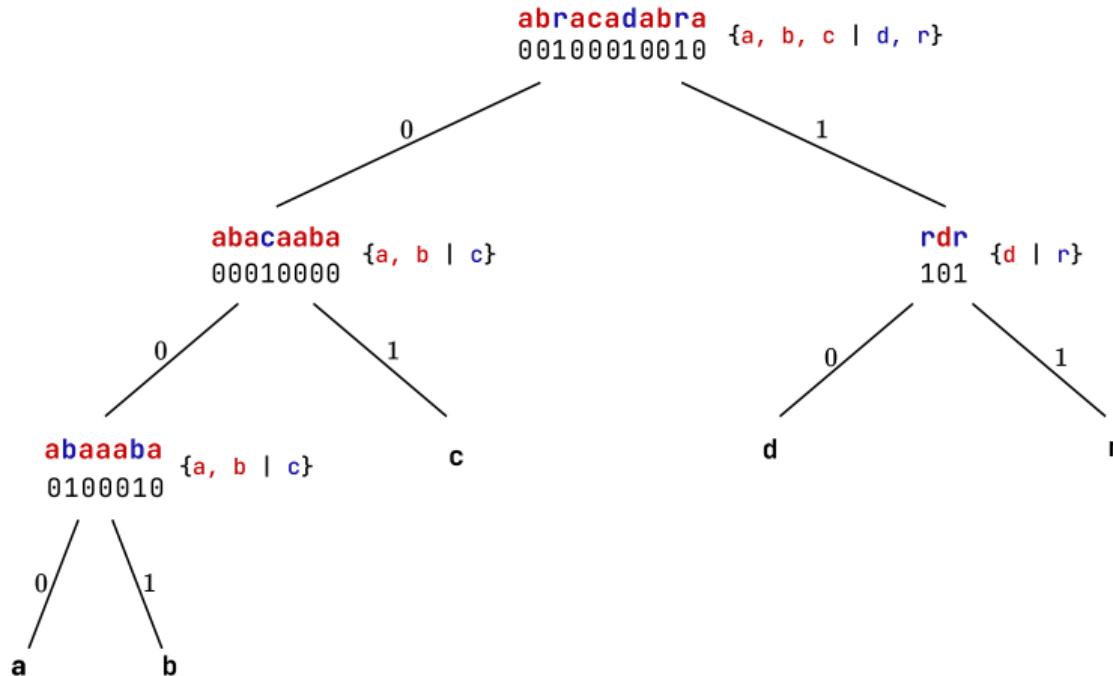
**rdr**  
101     $\{d \mid r\}$



## Beyond Bitvectors: General Alphabets

Wavelet Trees

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

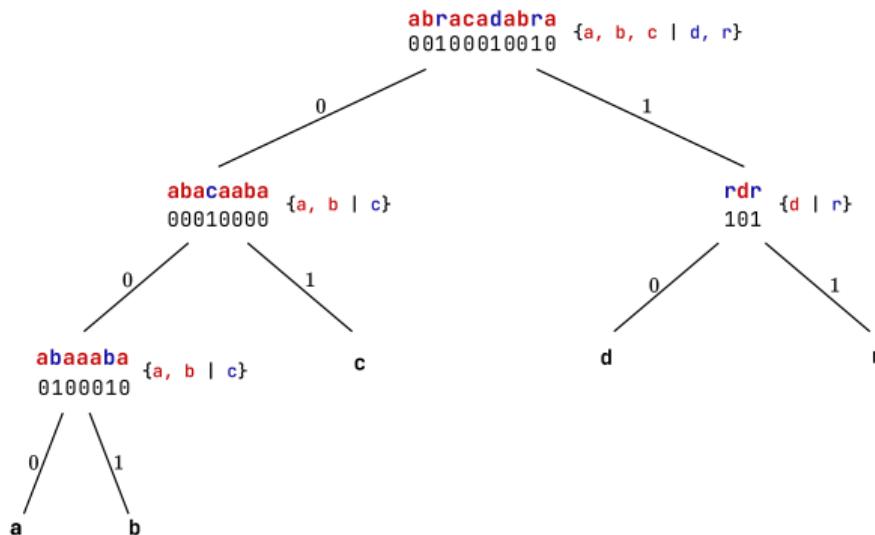




# Beyond Bitvectors: General Alphabets

## Wavelet Trees

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



## $\mathcal{H}_0$ -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,  $\text{rank}_c$ ,  $\text{select}_c$ .

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  $X = X_1X_2 \dots X_n$ , where each  $X_i$  is a subset of the alphabet  $\Sigma$  with cardinality  $\sigma$ . We can define the following operations:

- $\text{subset-rank}_X(i, c)$ : Counts sets  $X_k$  ( $k \leq i$ ) where  $c \in X_k$ .
- $\text{subset-select}_X(j, c)$ : 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  $X = X_1 X_2 \dots X_n$ , where each  $X_i$  is a subset of the alphabet  $\Sigma$  with cardinality  $\sigma$ . We can define the following operations:

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

$$X = \begin{array}{cccc} \left\{ \begin{array}{c} A \\ C \\ G \end{array} \right\} & \left\{ \begin{array}{c} A \\ T \end{array} \right\} & \left\{ \begin{array}{c} C \end{array} \right\} & \left\{ \begin{array}{c} T \\ G \end{array} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{array} \quad S = \begin{array}{ccccc} ACG & AT & C & TG \\ R = \begin{array}{ccccc} 100 & 10 & 1 & 10 & 1 \\ S_1 & S_2 & S_3 & S_4 \end{array} \end{array}$$



# Representing Sequence Variation: Degenerate Strings

Definitions and Core Operations

## Definition and Rank & Select Adaptation

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

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

$$X = \begin{array}{cccc} \left\{ \begin{array}{c} A \\ C \\ G \end{array} \right\} & \left\{ \begin{array}{c} A \\ T \end{array} \right\} & \left\{ \begin{array}{c} C \end{array} \right\} & \left\{ \begin{array}{c} T \\ G \end{array} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{array} \quad S = \begin{array}{ccccc} ACG & AT & C & TG \\ R = 100 & 10 & 1 & 10 & 1 \\ S_1 & S_2 & S_3 & S_4 \end{array}$$

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



# From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string  $X$ :

$$X = \begin{matrix} \left\{ \begin{matrix} A \\ C \\ G \end{matrix} \right\} & \left\{ \begin{matrix} A \\ T \end{matrix} \right\} & \left\{ \begin{matrix} T \\ C \\ A \end{matrix} \right\} & \left\{ \begin{matrix} A \\ G \end{matrix} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{matrix}$$



# From Degenerate Strings to Weighted DAGs

A New Perspective

Recall our degenerate string  $X$ :

$$X = \begin{matrix} \left\{ \begin{matrix} A \\ C \\ G \end{matrix} \right\} & \left\{ \begin{matrix} A \\ T \end{matrix} \right\} & \left\{ \begin{matrix} T \\ C \\ A \end{matrix} \right\} & \left\{ \begin{matrix} A \\ G \end{matrix} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{matrix}$$

## 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  $X$ :

$$X = \begin{matrix} \left\{ \begin{matrix} A \\ C \\ G \end{matrix} \right\} & \left\{ \begin{matrix} A \\ T \end{matrix} \right\} & \left\{ \begin{matrix} T \\ C \\ A \end{matrix} \right\} & \left\{ \begin{matrix} A \\ G \end{matrix} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{matrix}$$

## 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  $X$ :

$$X = \begin{matrix} \left\{ \begin{matrix} A \\ C \\ G \end{matrix} \right\} & \left\{ \begin{matrix} A \\ T \end{matrix} \right\} & \left\{ \begin{matrix} T \\ C \\ A \end{matrix} \right\} & \left\{ \begin{matrix} A \\ G \end{matrix} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{matrix}$$

## 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  $X$ :

$$X = \begin{matrix} \left\{ \begin{matrix} A \\ C \\ G \end{matrix} \right\} & \left\{ \begin{matrix} A \\ T \end{matrix} \right\} & \left\{ \begin{matrix} T \\ C \\ A \end{matrix} \right\} & \left\{ \begin{matrix} A \\ G \end{matrix} \right\} \\ X_1 & X_2 & X_3 & X_4 \end{matrix}$$

## 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  $P = (v_0 = s, \dots, v_k = v)$  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  $P = (v_0 = s, \dots, v_k = v)$  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 ( $v \neq s$ ):**

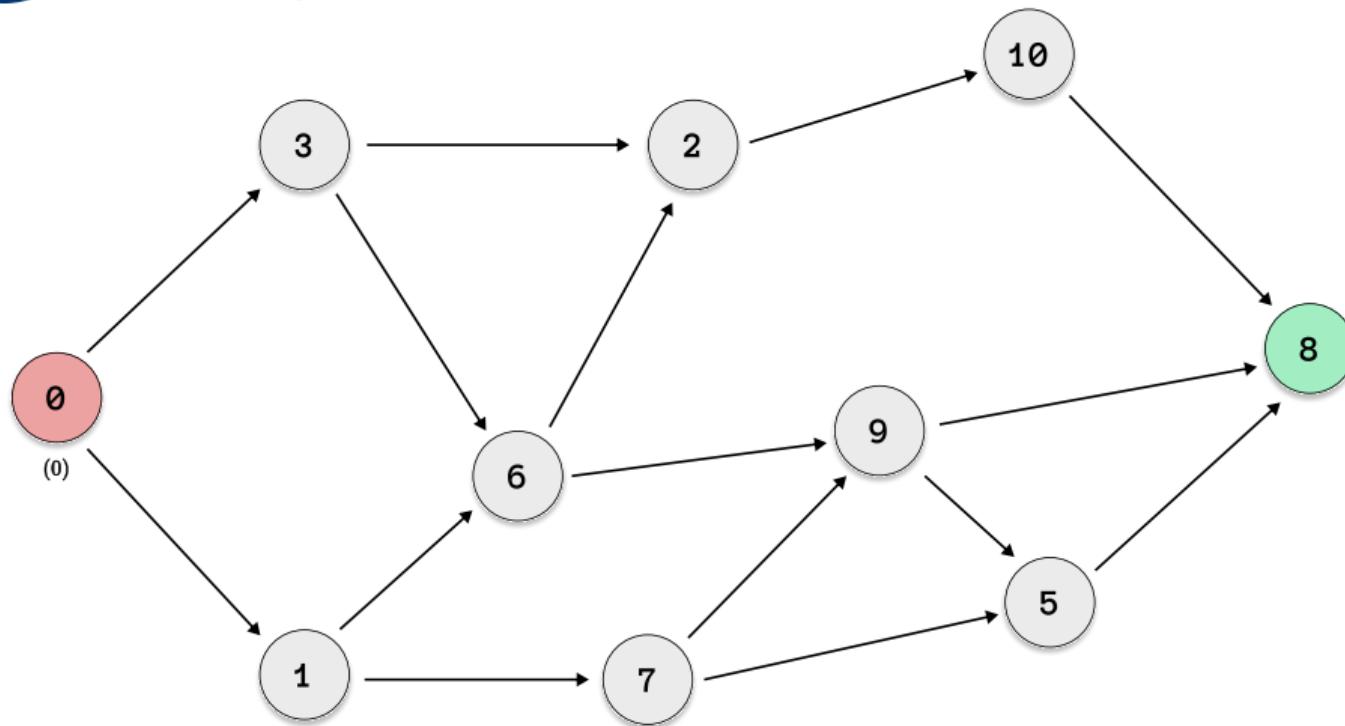
$$\mathcal{O}_v = \bigcup_{u \in \text{Pred}(v)} \{\gamma + w(v) \mid \gamma \in \mathcal{O}_u\} = \{W(P) \mid P \in \text{Path}(s, v)\}$$

Keep only **distinct** values. Store as a sorted sequence.



# O-Set Construction Example

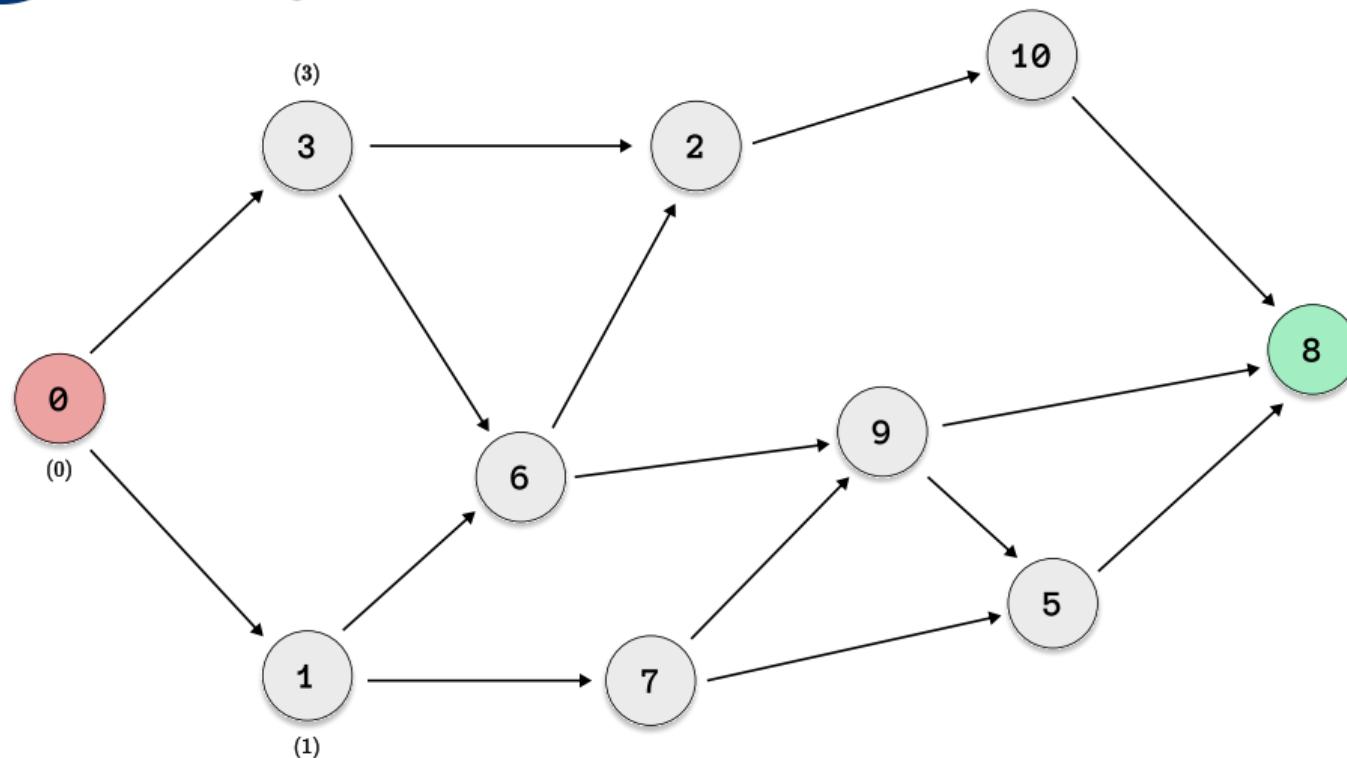
Visualizing the Recursive Definition





# O-Set Construction Example

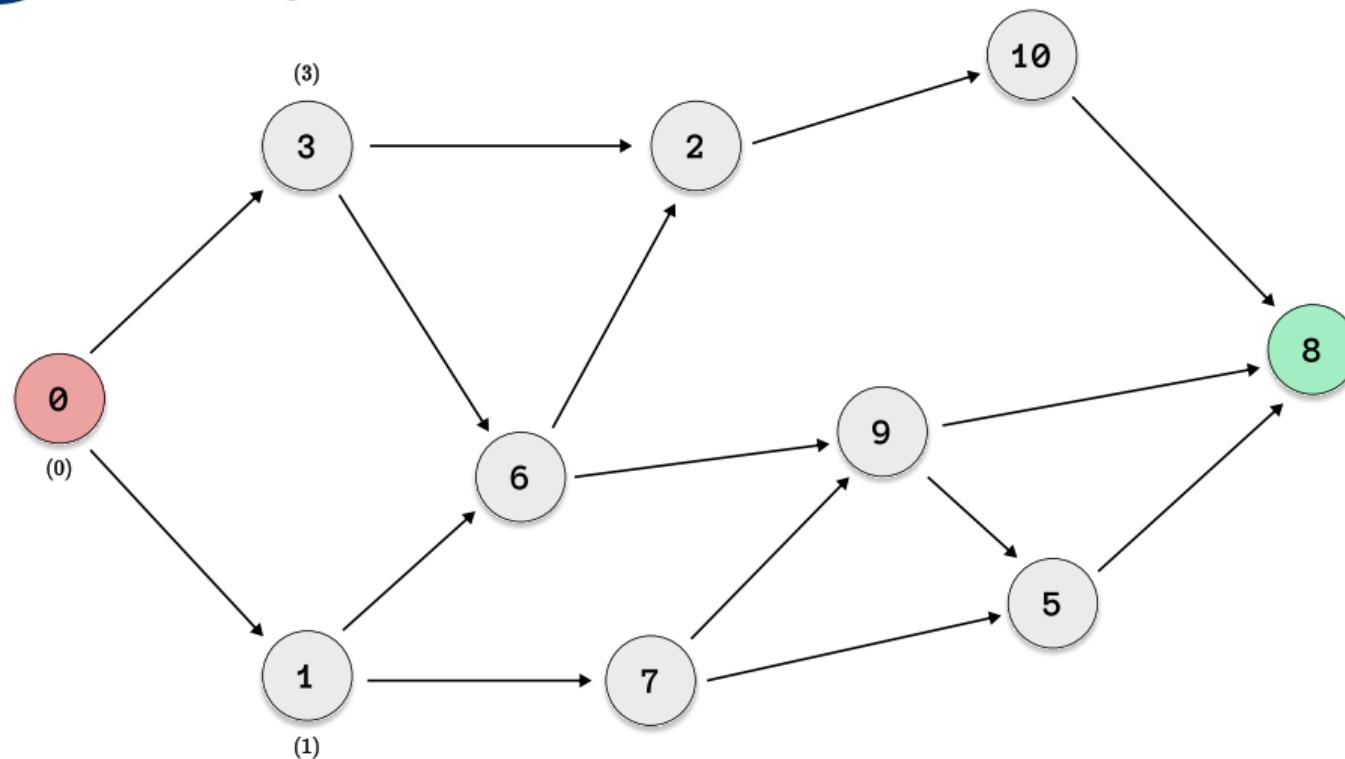
Visualizing the Recursive Definition





# O-Set Construction Example

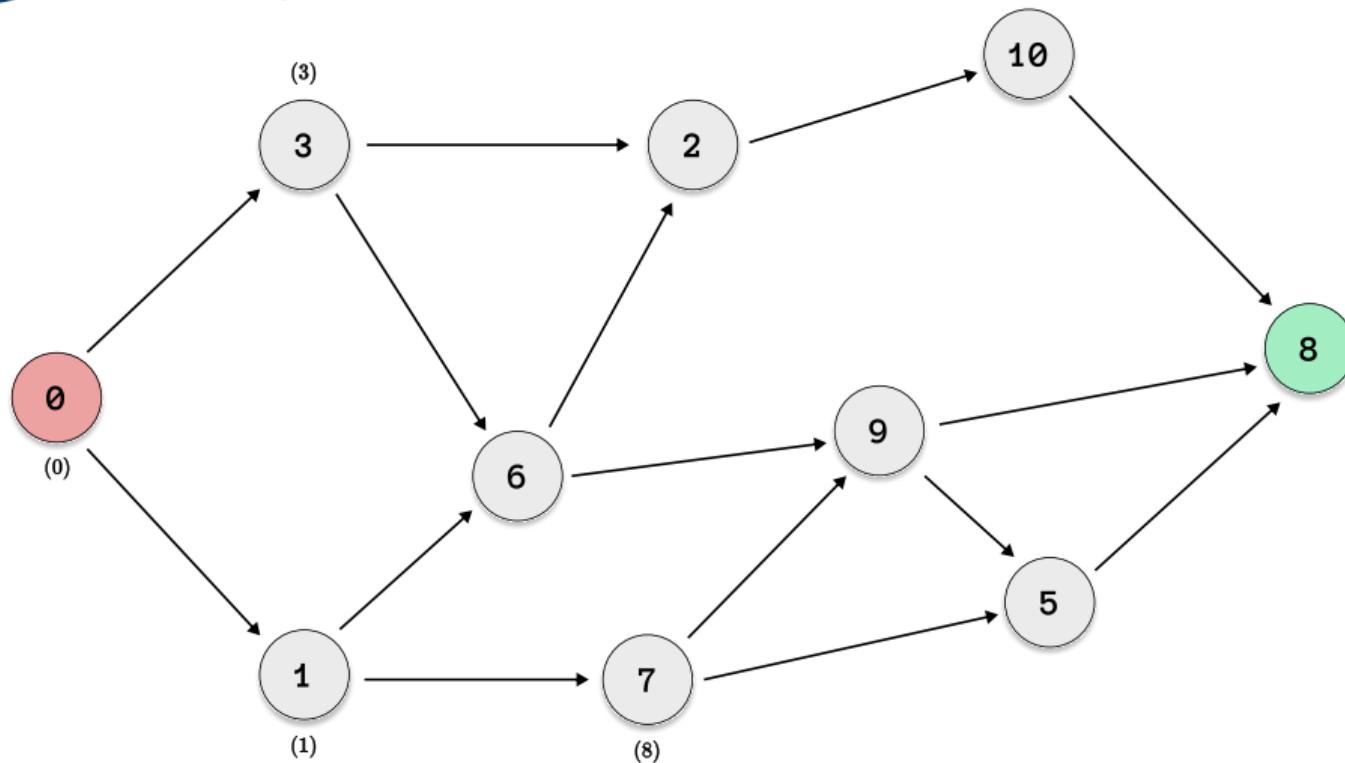
Visualizing the Recursive Definition





# O-Set Construction Example

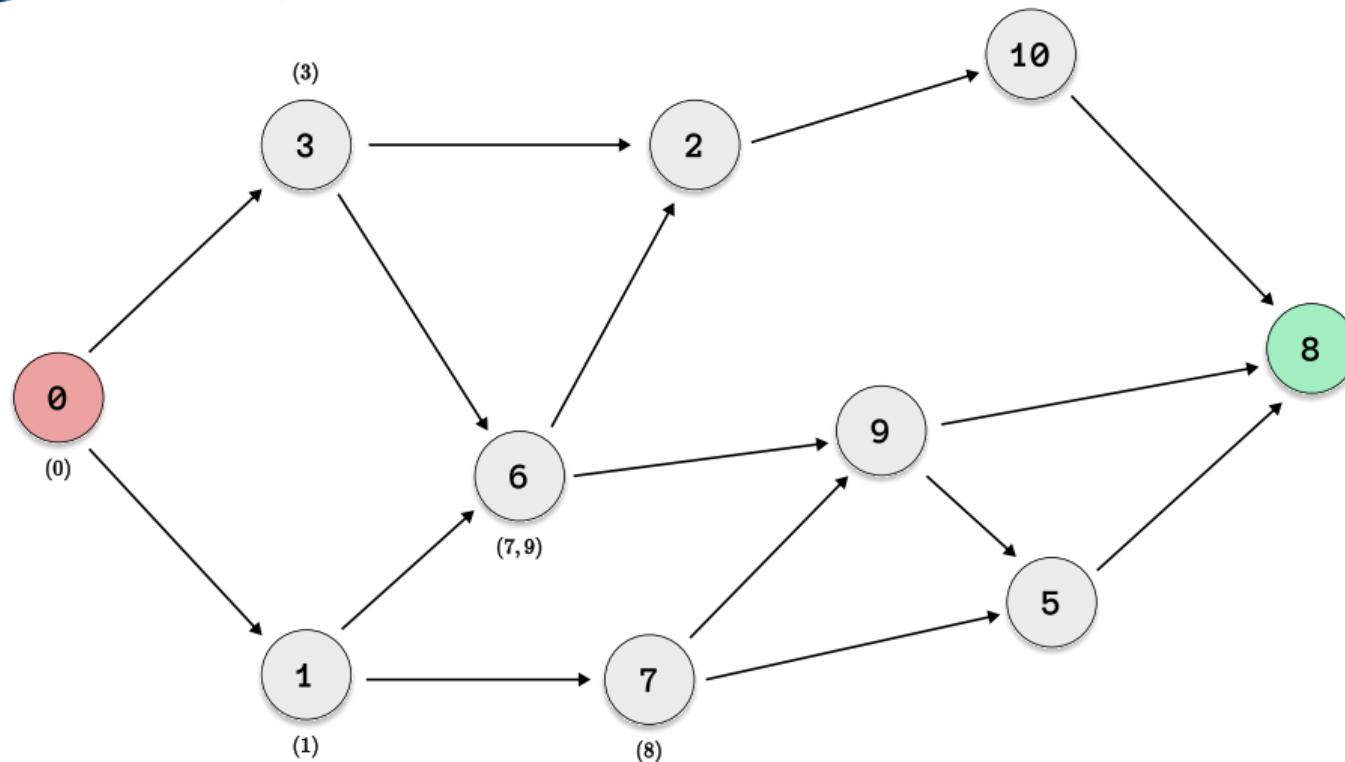
Visualizing the Recursive Definition





# O-Set Construction Example

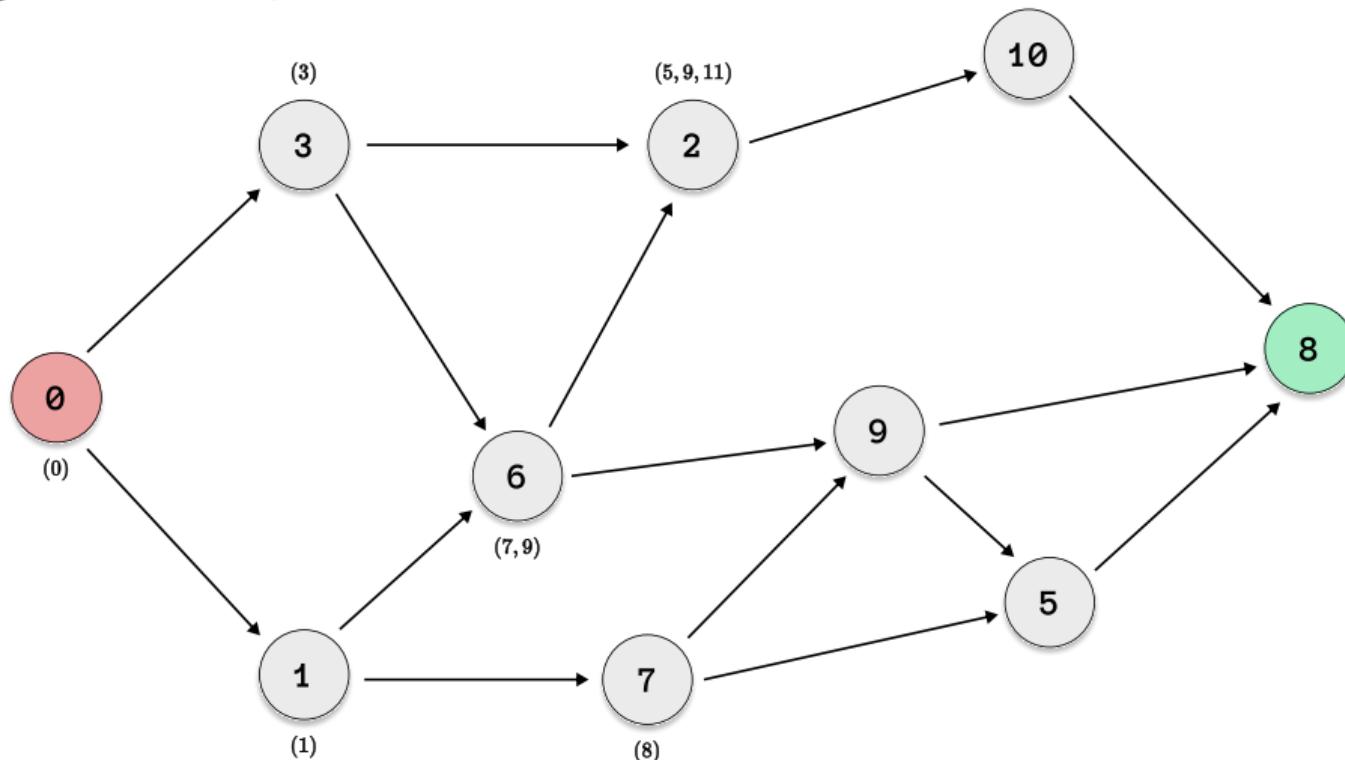
Visualizing the Recursive Definition





# O-Set Construction Example

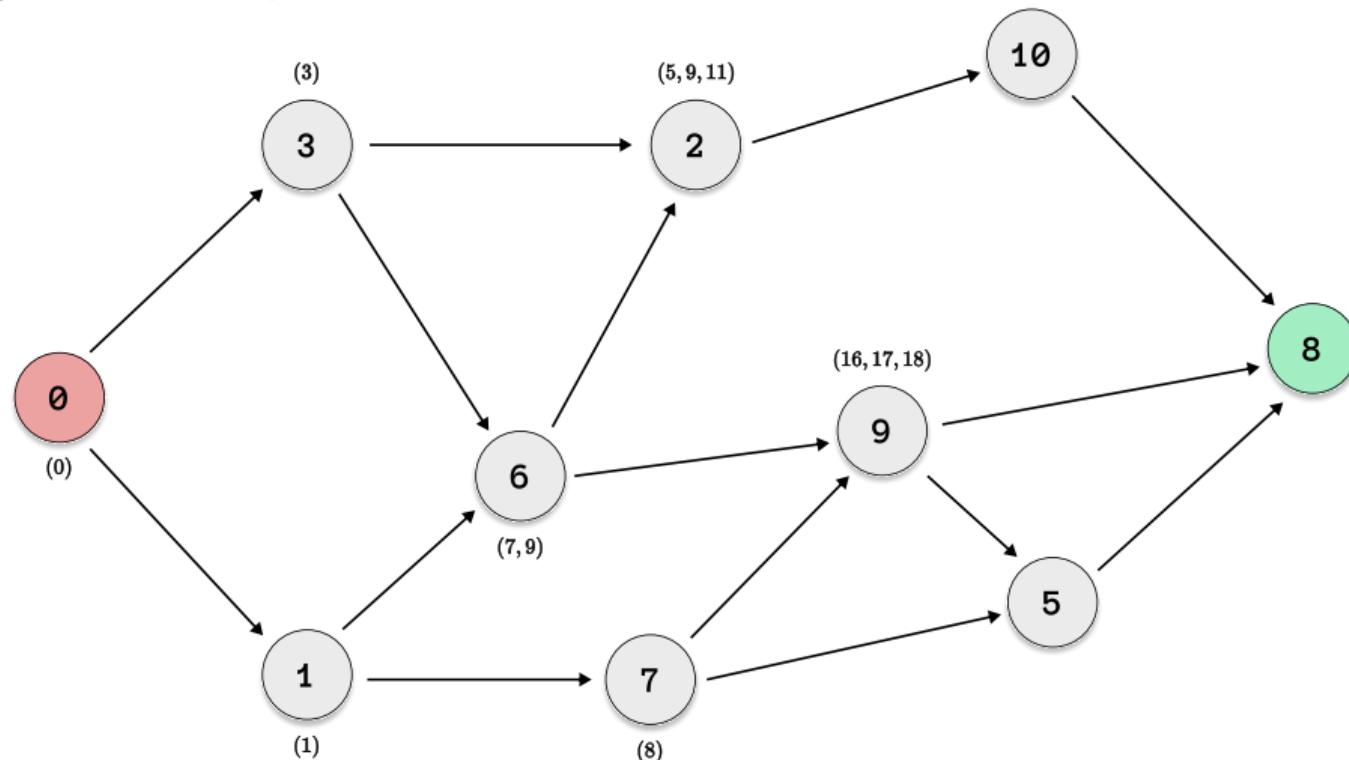
Visualizing the Recursive Definition





# O-Set Construction Example

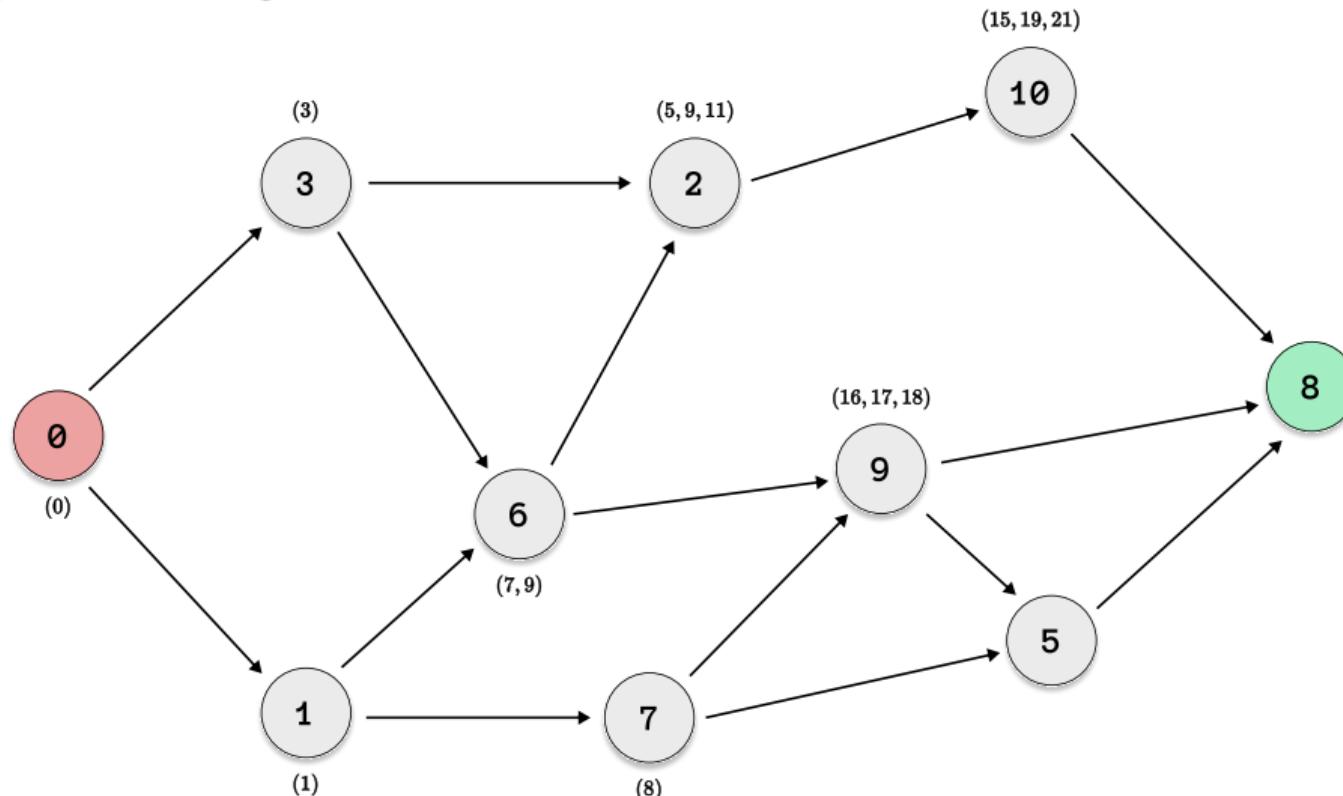
Visualizing the Recursive Definition





# O-Set Construction Example

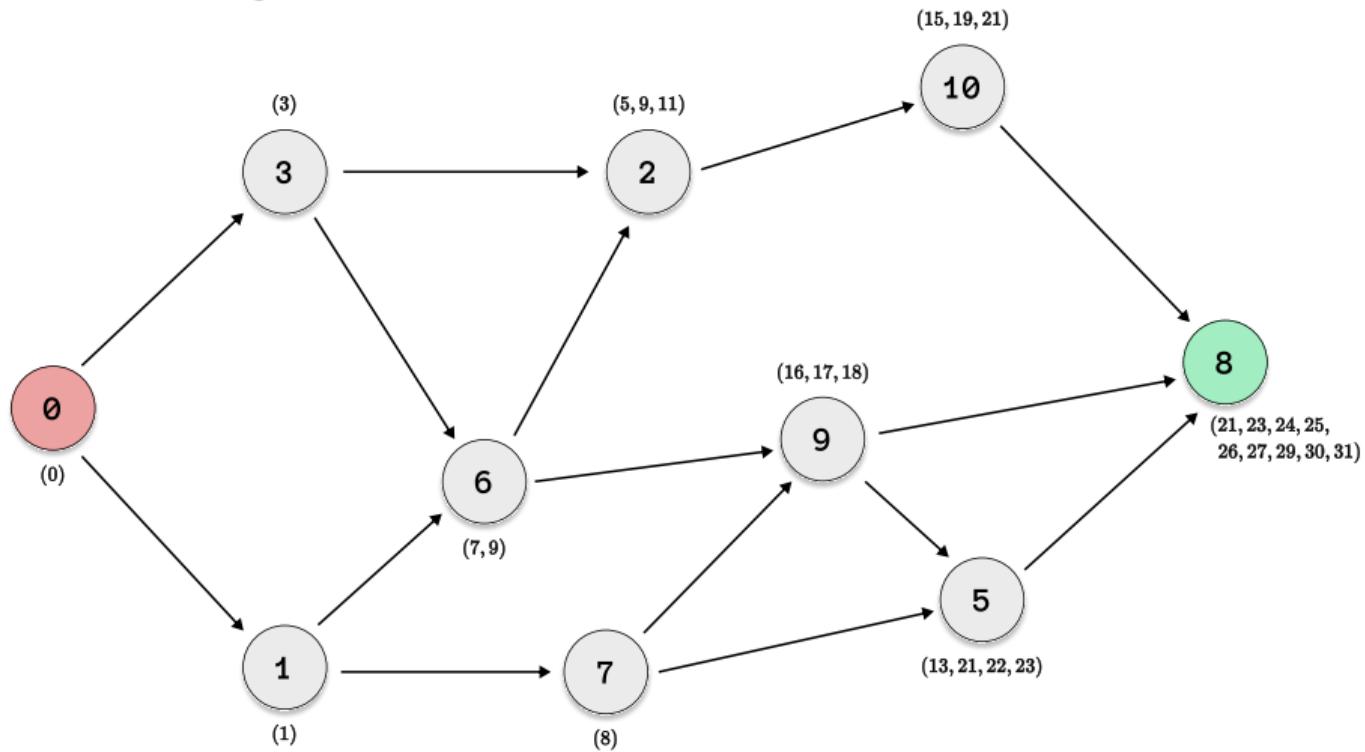
Visualizing the Recursive Definition





# 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: $\text{rank}_{\mathcal{G}}(N)$

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: $\text{rank}_{\mathcal{G}}(N)$

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  $\text{rank}_{\mathcal{G}}(N)$  returns a **minimal collection of disjoint closed integer intervals**

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

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 ( $V_E$ )

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.)

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  $|\mathcal{O}_u|$ .

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

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

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

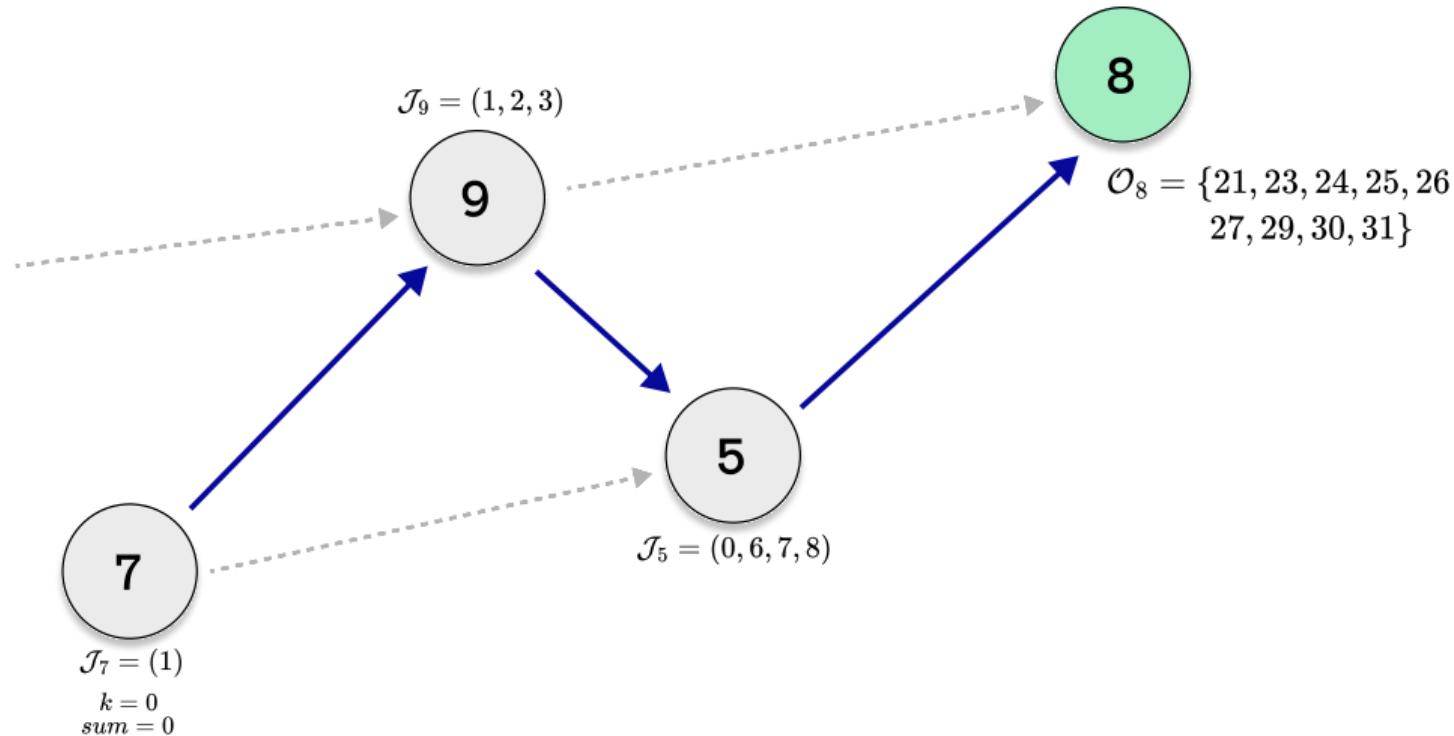
- **Relationship:** Each element  $x_k \in \mathcal{O}_v$  comes from some  $y_{j_k} \in \mathcal{O}_u$  via  $x_k = y_{j_k} - w(u)$ .
- **Offset Sequence  $\mathcal{J}_v$ :** Stores the index  $j_k$  corresponding to each  $x_k$ .

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



## Example: Computing $\mathcal{O}_7[0]$

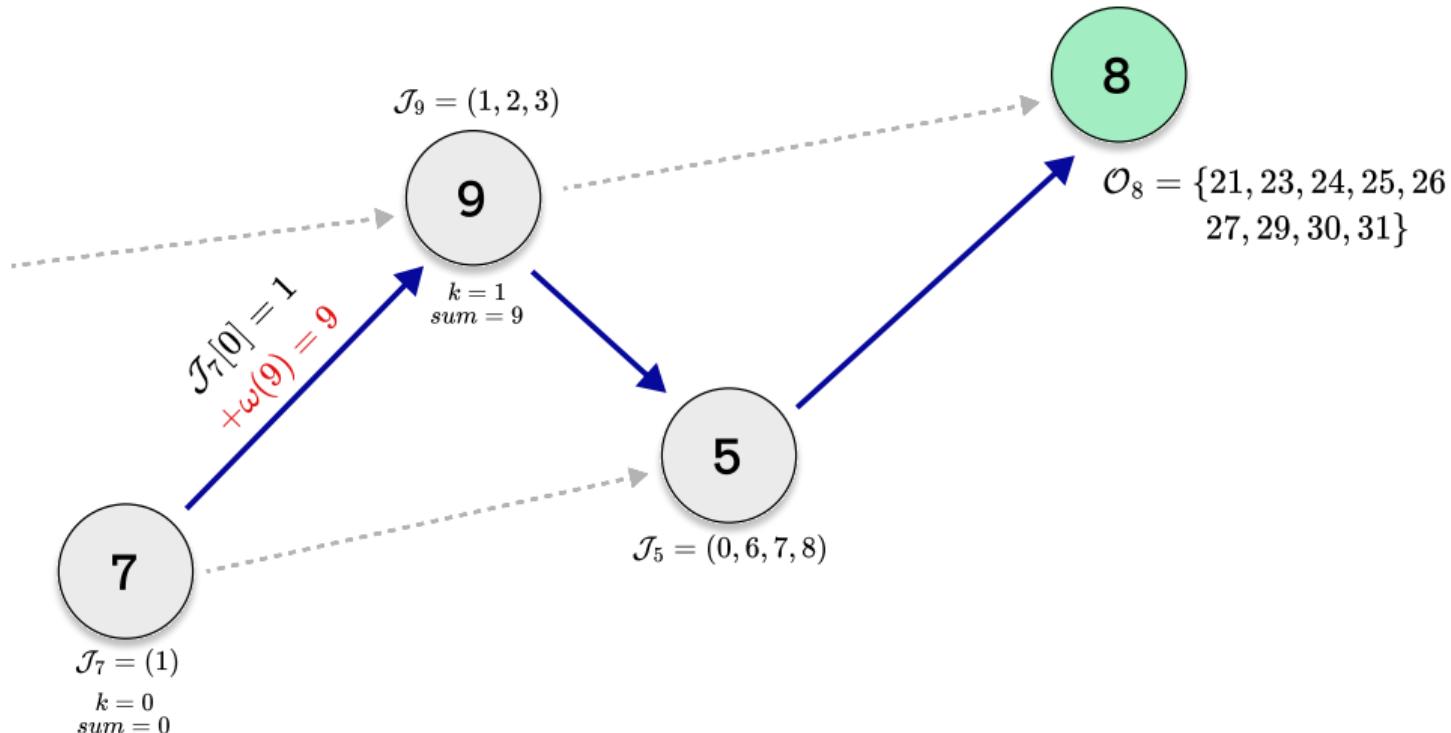
Following the Successor Path:  $7 \rightarrow 9 \rightarrow 5 \rightarrow 8$





## Example: Computing $\mathcal{O}_7[0]$

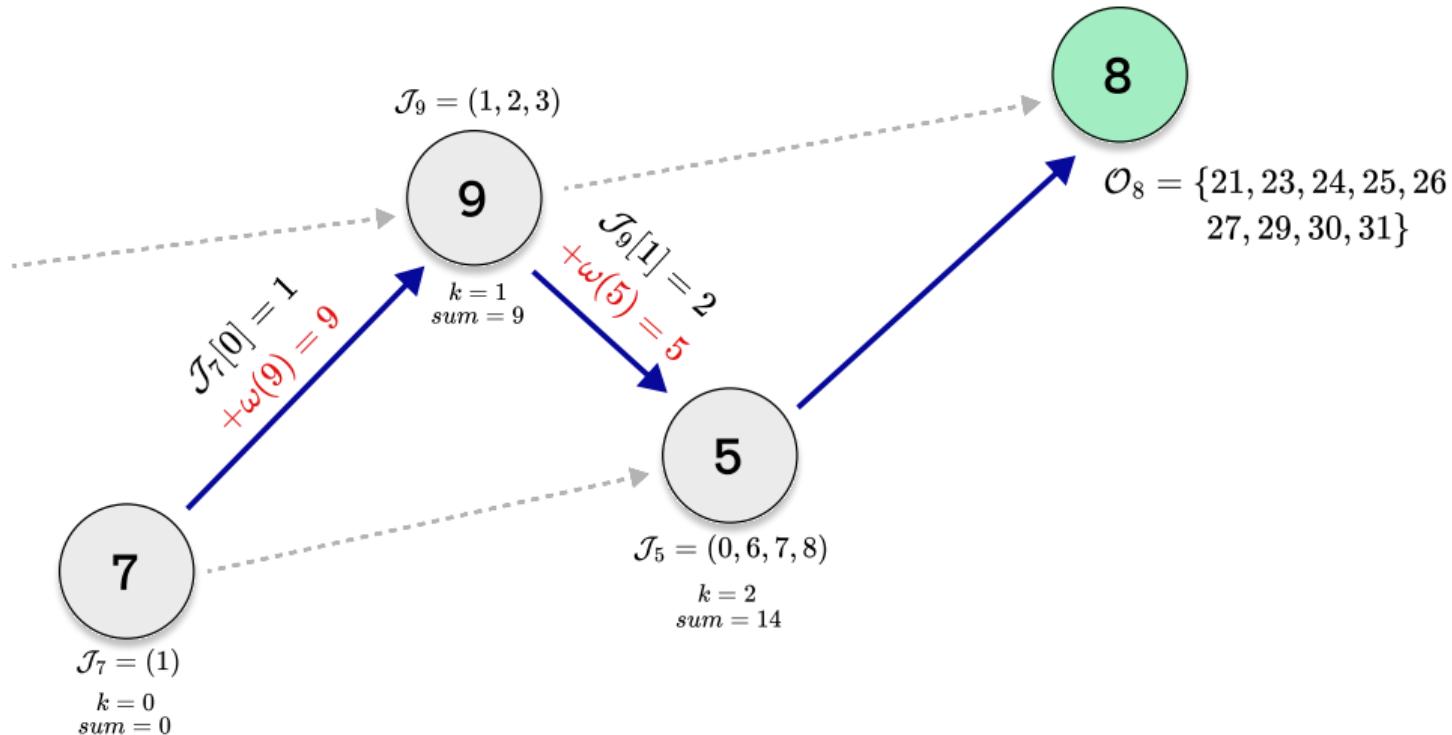
Following the Successor Path:  $7 \rightarrow 9 \rightarrow 5 \rightarrow 8$





## Example: Computing $\mathcal{O}_7[0]$

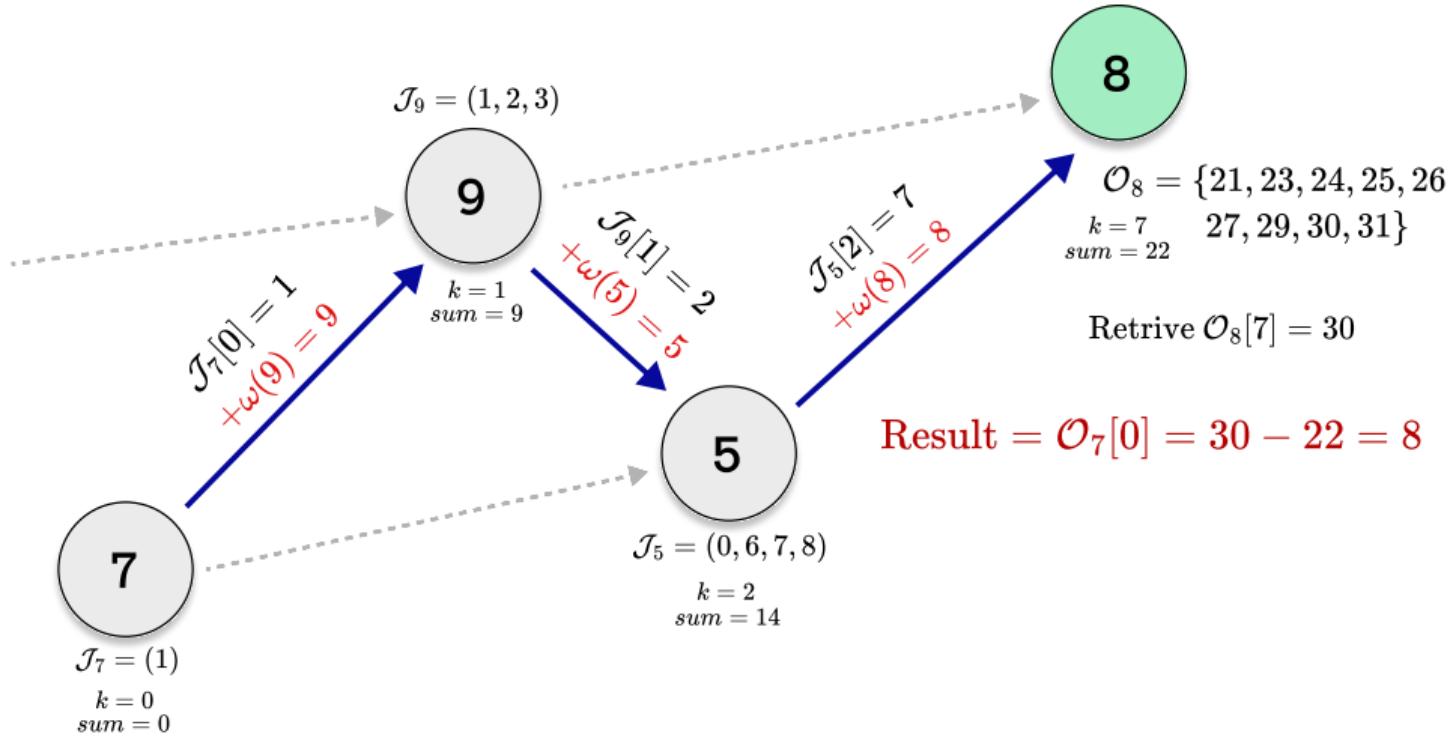
Following the Successor Path:  $7 \rightarrow 9 \rightarrow 5 \rightarrow 8$





## Example: Computing $\mathcal{O}_7[0]$

Following the Successor Path:  $7 \rightarrow 9 \rightarrow 5 \rightarrow 8$

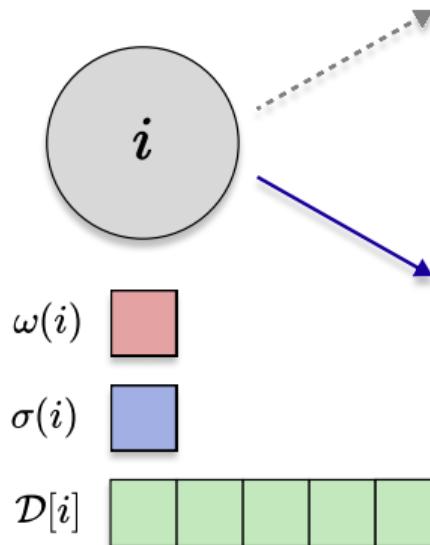




# Succinct Data Structure: Components

Arrays Indexed by Vertex ID

Each node stores 3 components

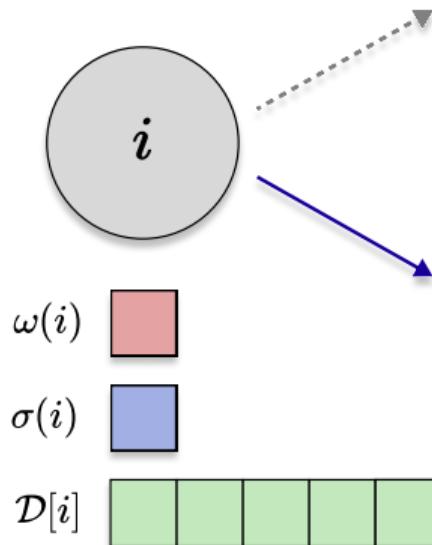




# Succinct Data Structure: Components

Arrays Indexed by Vertex ID

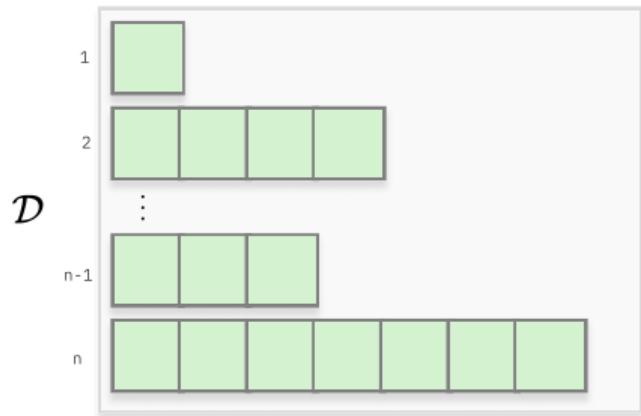
Each node stores 3 components



Succinct DAG as a Struct of Arrays

$\omega$	0	$\omega(1)$	$\omega(2)$	$\dots$	$\dots$	$\dots$	$\dots$	$\omega(n)$
----------	---	-------------	-------------	---------	---------	---------	---------	-------------

$\sigma$	0	$\sigma(1)$	$\sigma(2)$	$\dots$	$\dots$	$\dots$	$\dots$	$\sigma(n)$
----------	---	-------------	-------------	---------	---------	---------	---------	-------------





# Compression Strategies

Reducing Memory Footprint

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



# Compression Strategies

Reducing Memory Footprint

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

## Compression Options

- Variable-Length Integer Coding —→ we published a Rust library<sup>a</sup> for this!
- Wavelet Trees (*for nodes with small weight range*)

---

<sup>a</sup><https://crates.io/crates/compressed-intvec>



# Compression Strategies

Reducing Memory Footprint

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

## Compression Options

- Variable-Length Integer Coding —→ we published a Rust library<sup>a</sup> for this!
- Wavelet Trees (*for nodes with small weight range*)

---

<sup>a</sup><https://crates.io/crates/compressed-intvec>



# Compression Strategies

Reducing Memory Footprint

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

## Compression Options

- Variable-Length Integer Coding —→ we published a Rust library<sup>a</sup> for this!
- Wavelet Trees (*for nodes with small weight range*)
- Elias-Fano Encoding (*for monotonic sequences*)
- Run-Length Encoding (RLE) (*for clustered monotonic sequences*)

<sup>a</sup><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<sup>th</sup>-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<sup>th</sup>-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<sup>th</sup>-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<sup>th</sup>-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  $\Sigma$
- 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  $\Sigma$
- 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
<hr/>	
Precomputed Rank Queries:	
Explicit Binary Storage	4,854,533
Elias-Fano Compressed	2,211,849
<hr/>	
Our Succinct DAG	602,808
Weights $\mathcal{W}$	60,824
Successors $\Sigma$	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
<hr/>	
Precomputed Rank Queries:	
Explicit Binary Storage	4,854,533
Elias-Fano Compressed	2,211,849
<hr/>	
Our Succinct DAG	602,808
Weights $\mathcal{W}$	60,824
Successors $\Sigma$	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  $\Sigma$ ).

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

$$v \rightarrow \sigma(v) \rightarrow \sigma(\sigma(v)) \rightarrow \dots \rightarrow e \in V_E$$

**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

$$v \rightarrow \sigma(v) \rightarrow \sigma(\sigma(v)) \rightarrow \dots \rightarrow e \in V_E$$

**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.