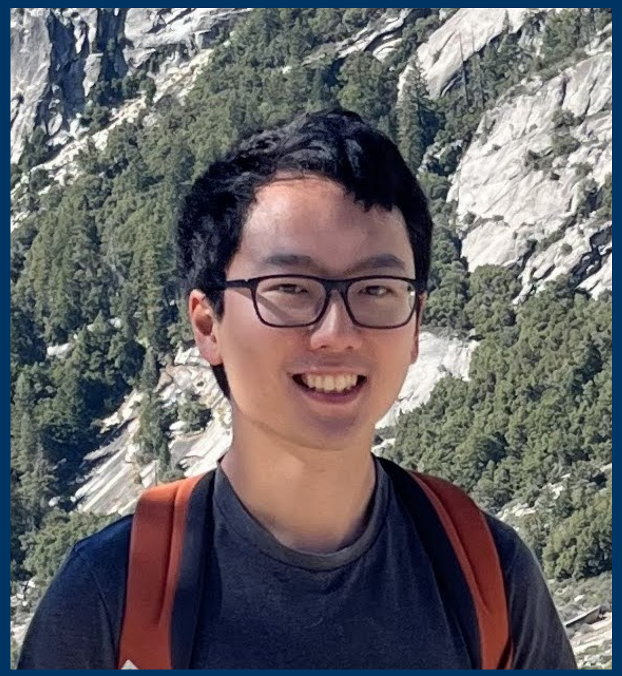
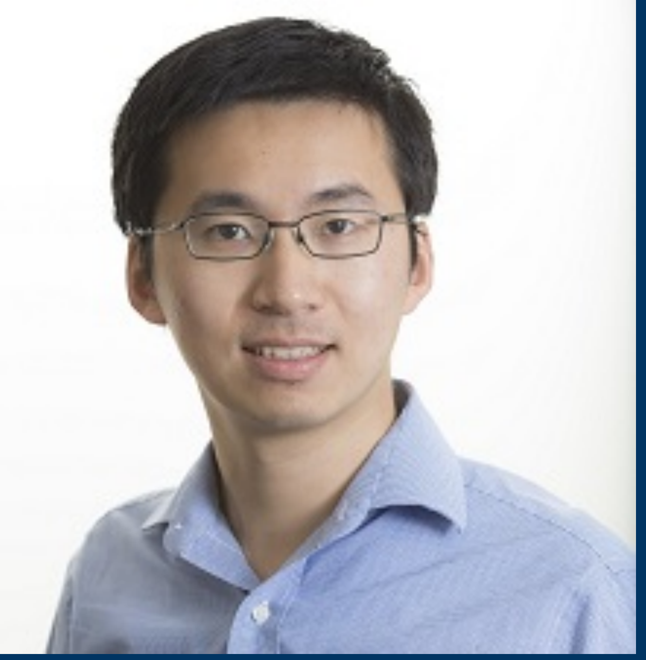
Fast algorithms for separable linear programs

Sally Dong
University of Washington

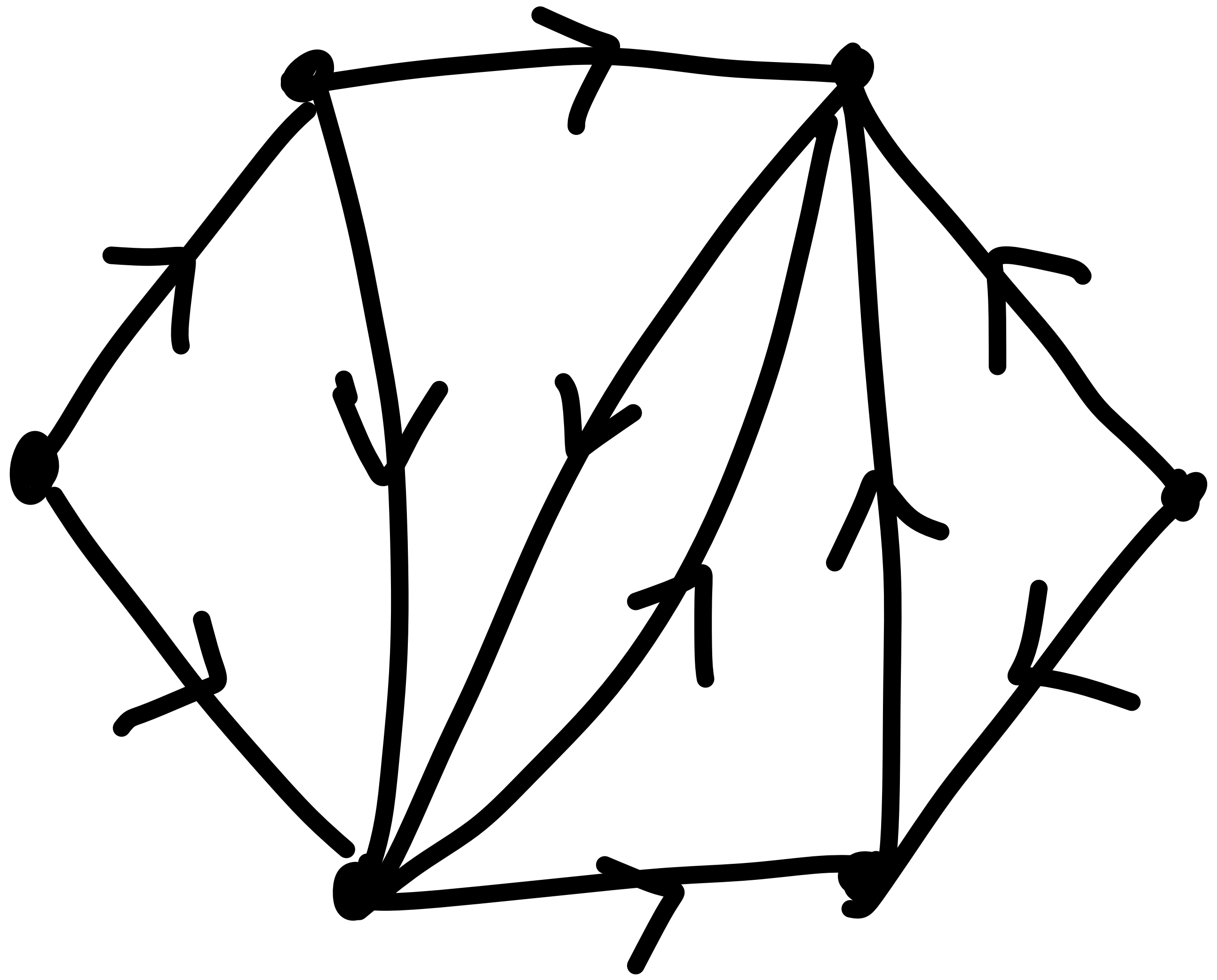
Joint work with:
Yu Gao, Gramoz Goranci, Yin Tat Lee, Lawrence Li, Richard Peng, Sushant Sachdeva, Guanghao Ye

Min-cost flow

Input:

- graph $G = (V, E)$ with n vertices and m directed edges,

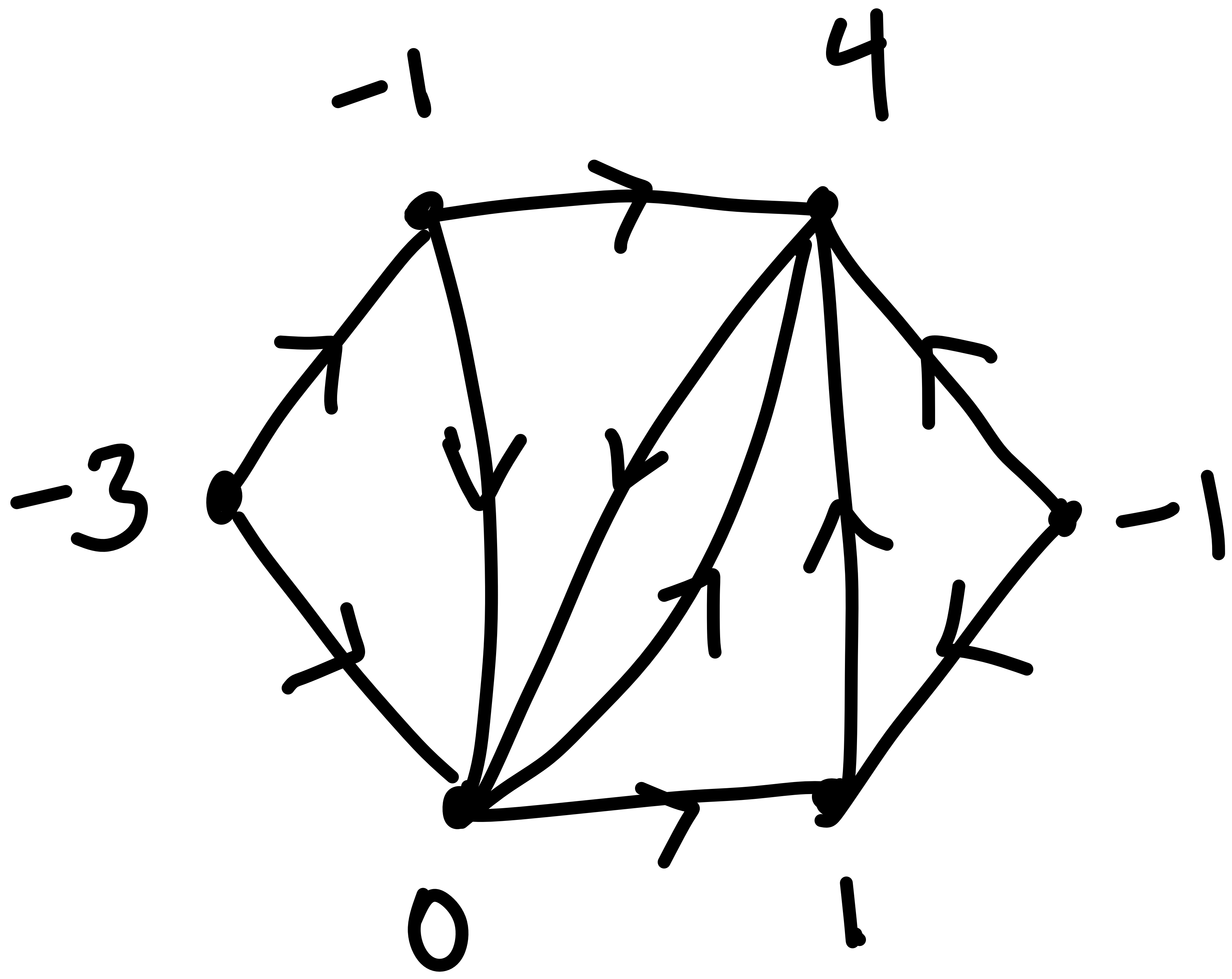


Min-cost flow

Input:

- graph $G = (V, E)$ with n vertices and m directed edges,
- vertex demands $\boldsymbol{d}$, such that

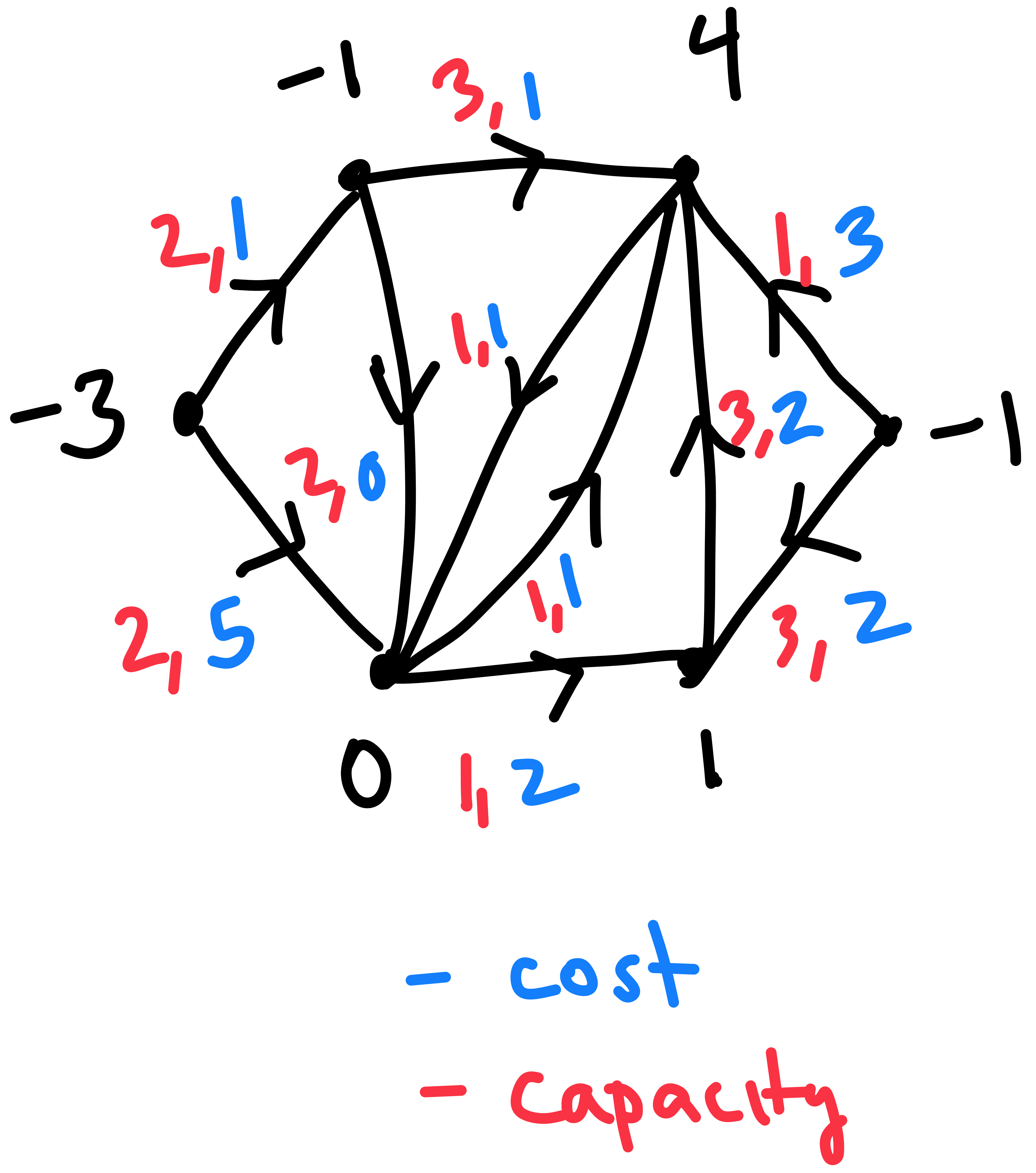
$$\sum_{v} d_v = 0.$$



Min-cost flow

Input:

- graph $G = (V, E)$ with n vertices and m directed edges,
- vertex demands $\boldsymbol{d}$, such that
$$\sum_v d_v = 0.$$
- edge capacities $\boldsymbol{u} \geq 0$ and costs $\boldsymbol{c}$.



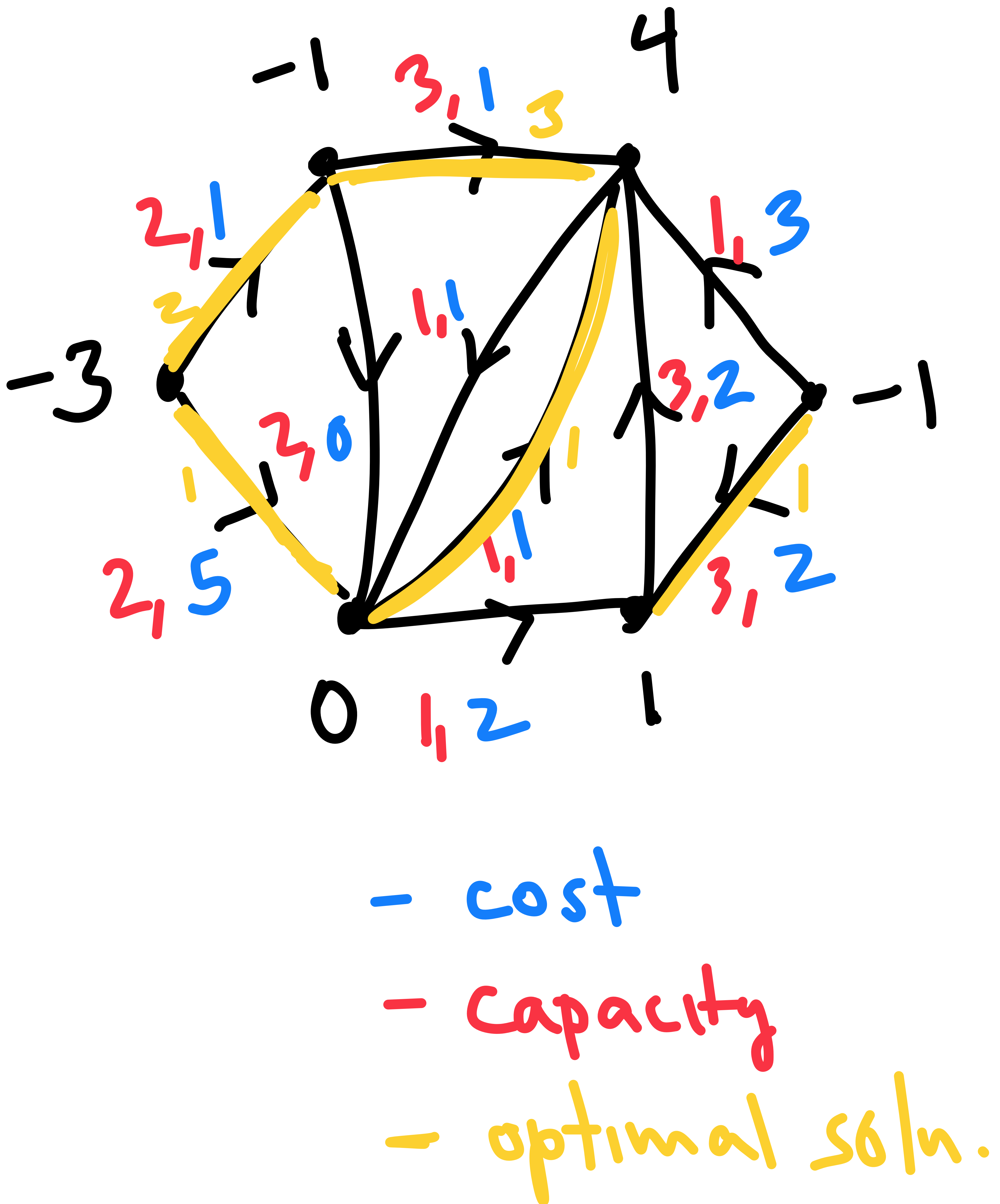
Min-cost flow

Input:

- graph $G = (V, E)$ with n vertices and m directed edges,
- vertex demands $\boldsymbol{d}$, such that
$$\sum_v \boldsymbol{d}_v = 0.$$
- edge capacities $\boldsymbol{u} \geq 0$ and costs $\boldsymbol{c}$.

Output:

- Flow $\boldsymbol{f}$ minimizing $\boldsymbol{c}^\top \boldsymbol{f}$, and satisfying capacity constraints and demands.



General LP

$$
\begin{aligned}
\min\ & \boldsymbol{c}^\top \boldsymbol{x} \\
\text{s.t.}\quad & \mathbf{A}\boldsymbol{x} = \boldsymbol{b} \\
& \boldsymbol{x} \leq \boldsymbol{u} \\
& \boldsymbol{x} \geq \boldsymbol{\ell}
\end{aligned}
$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$.

Dual graph of general LP

$$
\begin{aligned}
& \min \boldsymbol{c}^\top \boldsymbol{x} \\
\text{s.t.} \quad & \mathbf{A}\boldsymbol{x} = \boldsymbol{b} \\
& \boldsymbol{x} \le \boldsymbol{u} \\
& \boldsymbol{x} \ge \boldsymbol{\ell}
\end{aligned}
$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$.

Dual graph $G_{\mathbf{A}}$: n vertices, and each column of $\mathbf{A}$ is a hyper-edge (equiv. clique) on the set of vertices corresponding to rows with non-zero entries

Treewidth of LP

$$
\begin{aligned}
& \min \boldsymbol{c}^\top \boldsymbol{x} \\
\text{s.t.} \quad & \mathbf{A}\boldsymbol{x} = \boldsymbol{b} \\
& \boldsymbol{x} \le \boldsymbol{u} \\
& \boldsymbol{x} \ge \boldsymbol{\ell}
\end{aligned}
$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$.

Dual graph $G_{\mathbf{A}}$: n vertices, and each column of $\mathbf{A}$ is a hyper-edge (equiv. clique) on the set of vertices corresponding to rows with non-zero entries

Define treewidth of the LP to be the treewidth of $G_{\mathbf{A}}$.

Separable LP

$$\begin{aligned} & \min \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \mathbf{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \leq \boldsymbol{u} \\ & \boldsymbol{x} \geq \boldsymbol{\ell} \end{aligned}$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$.

Dual graph $G_{\mathbf{A}}$: n vertices, and each column of $\mathbf{A}$ is a hyper-edge (equiv. clique) on the set of vertices corresponding to rows with non-zero entries

Say LP is separable if $G_{\mathbf{A}}$ is separable.

Min-cost flow LP

$$
\begin{aligned}
& \min \boldsymbol{c}^\top \boldsymbol{f} \\
\text{s.t.} \quad & \mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d} \\
& \boldsymbol{f} \le \boldsymbol{u} \\
& \boldsymbol{f} \ge \mathbf{0}
\end{aligned}
$$

where $\mathbf{B}^\top \in \mathbb{R}^{n \times m}$ is the transpose of the adjacency matrix of input graph G.

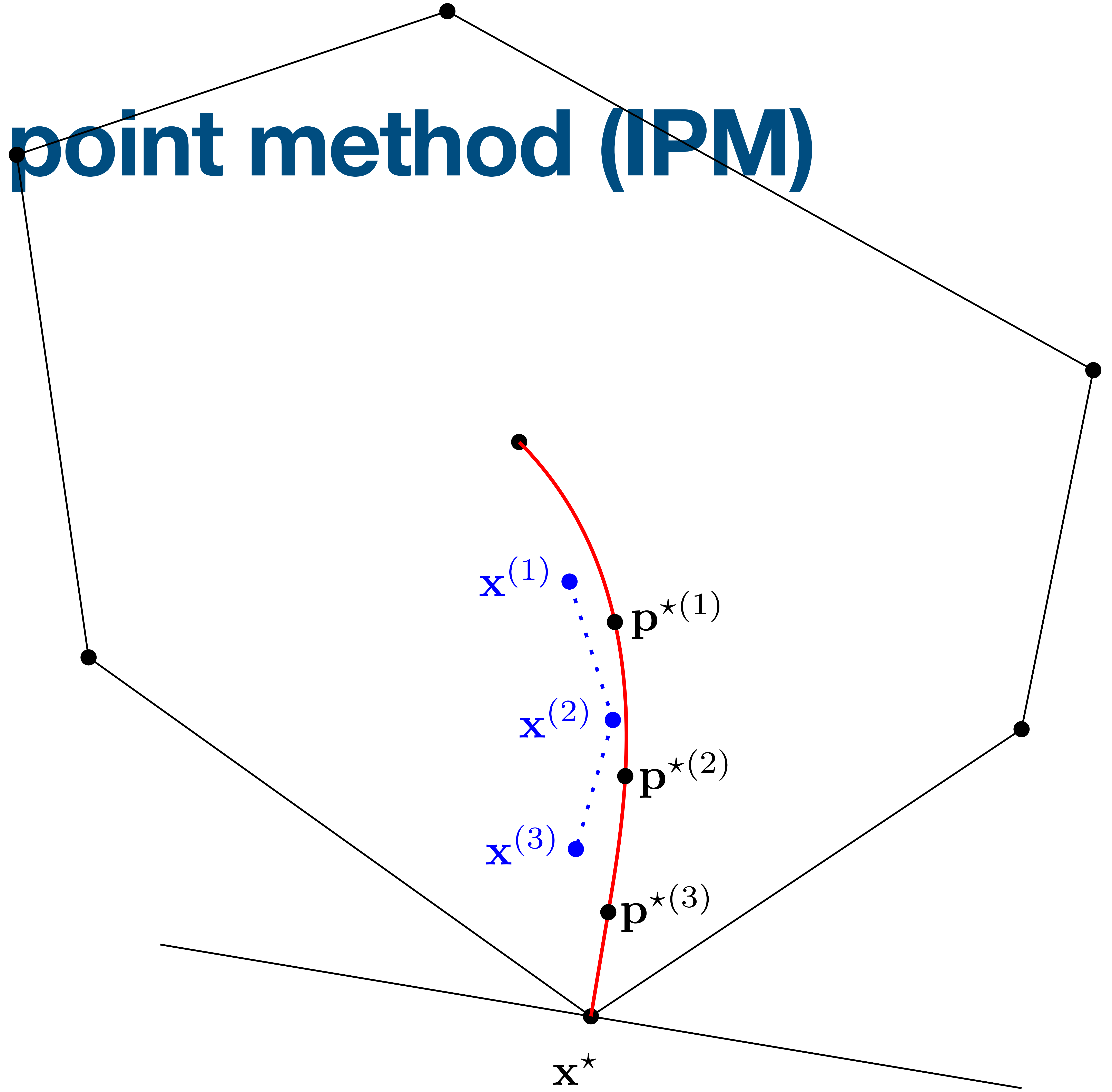
Current state of the art

Problem setting	Time	Reference
min-cost flow, strongly polytime	$O(mn + m^{31/16})$	[Orlin13]
min-cost flow, weakly polytime	$O(m^{1+o(1)})$	[CLKPPS22]
min-cost flow, planar graphs	$\tilde{O}(m)$	[**D**GGLPSY22]
min-cost flow, treewidth *t* graphs	$\tilde{O}(m\sqrt{t})$	[**D**Y23+]
k-commodity flow	$\tilde{O}(k^{2.5}\sqrt{m}n^{\omega-1/2})$	[BZ23]
k-commodity flow, planar graphs	$\tilde{O}(k^{2.5}n^{1.5})$	[**D**GLSY24]
general LPs	$\tilde{O}(m^{\omega})$	[CLS19]
LPs with treewidth *t*	$\tilde{O}(mt^{(\omega+1)/2})$	[GS22,**D**GLSY24]
α -separable LPs	$\tilde{O}(m^{1/2+2\alpha})$	[**D**GLSY24]

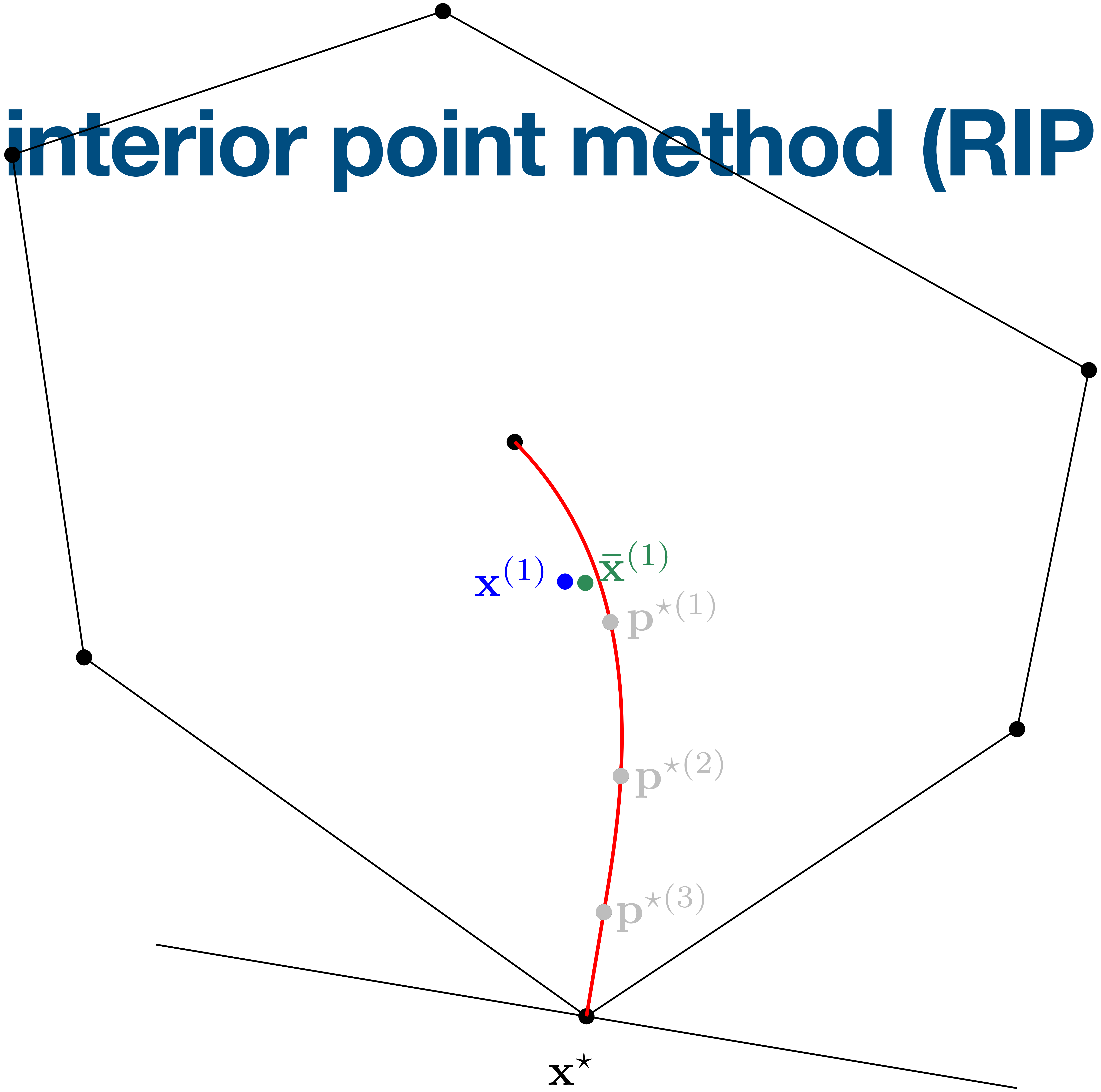
m variables; polynomially bounded entries and relative error

Interior point method for LPs

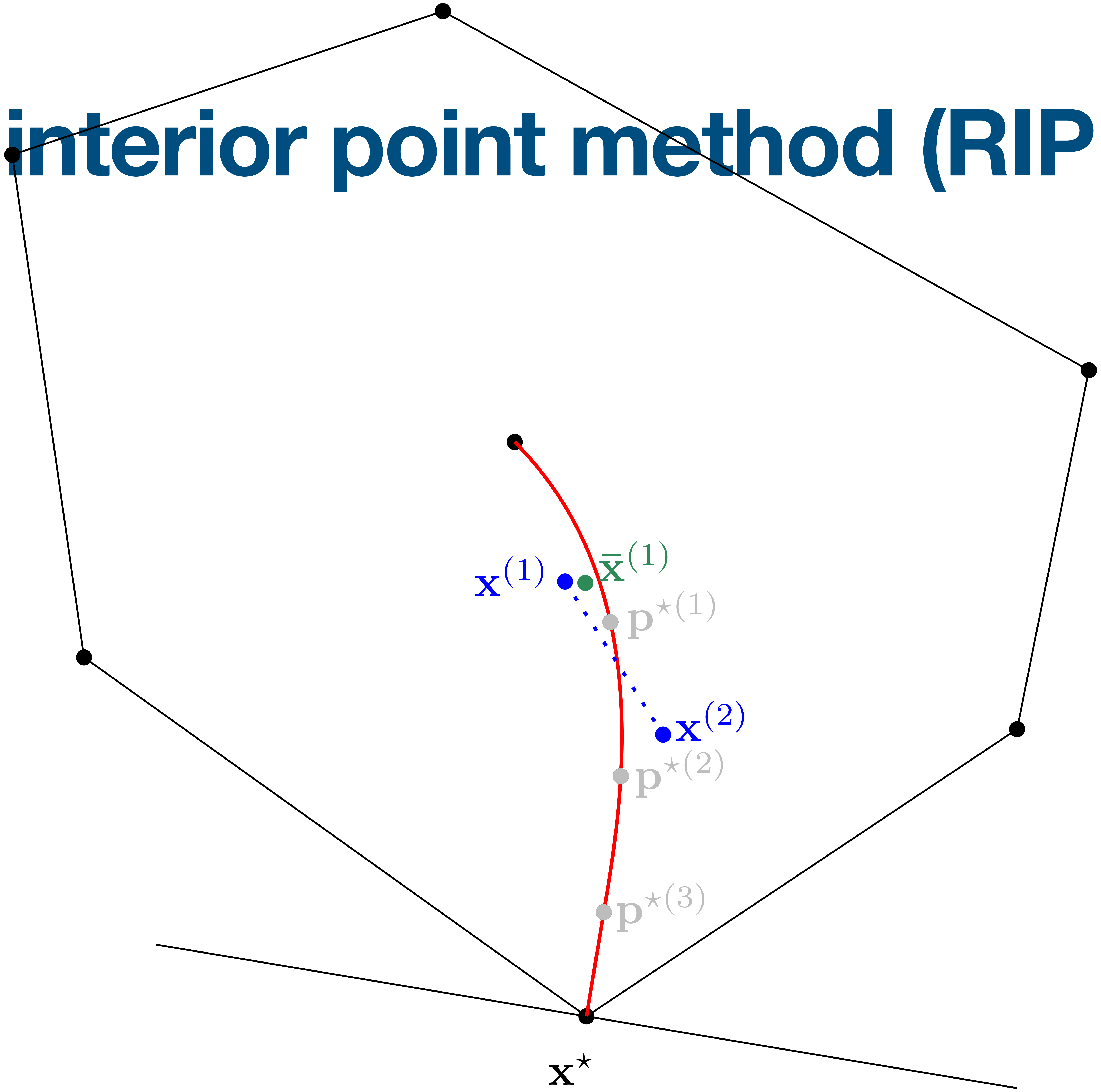
Interior point method (IPM)



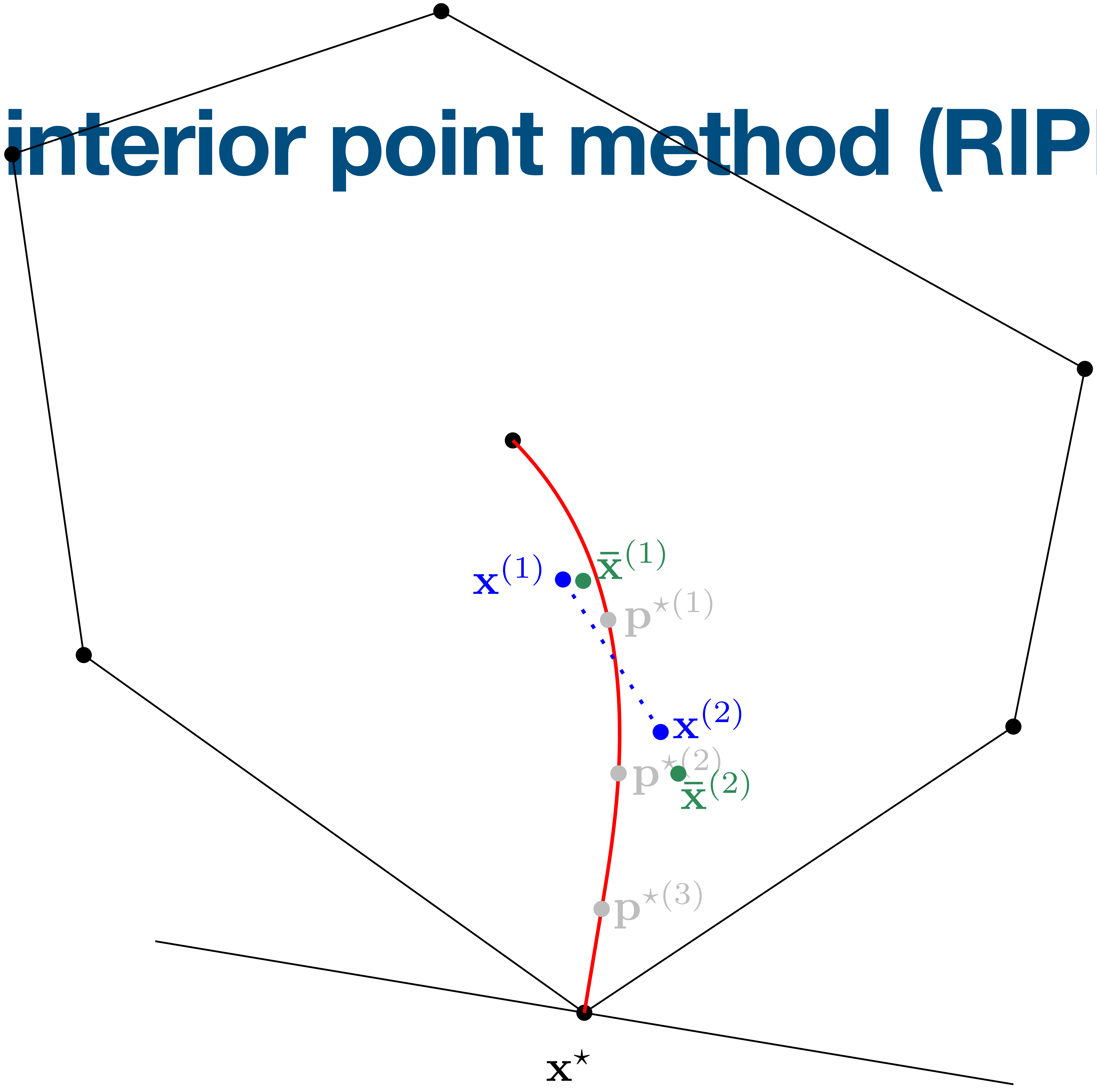
Robust interior point method (RIPM)



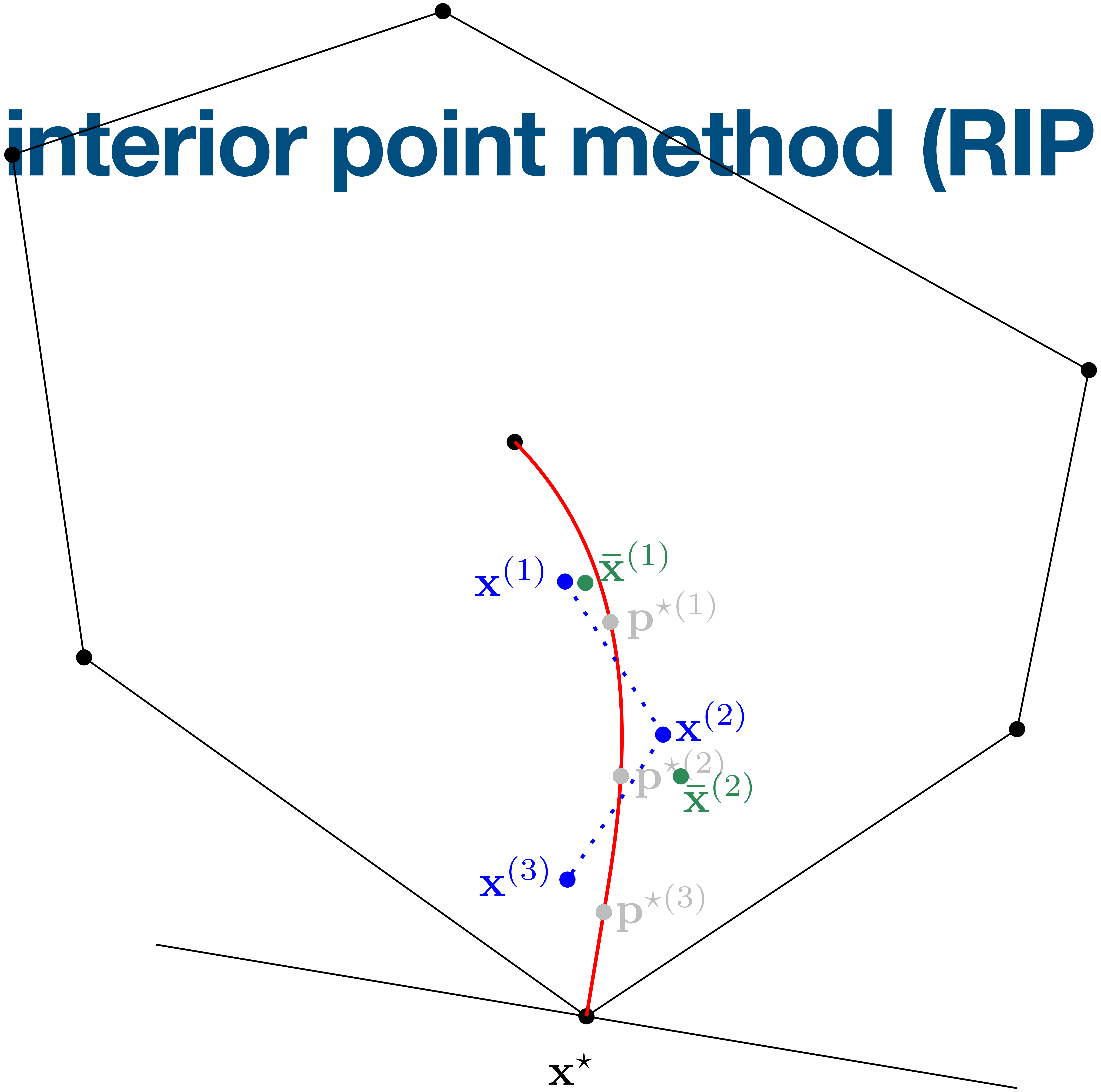
Robust interior point method (RIPM)



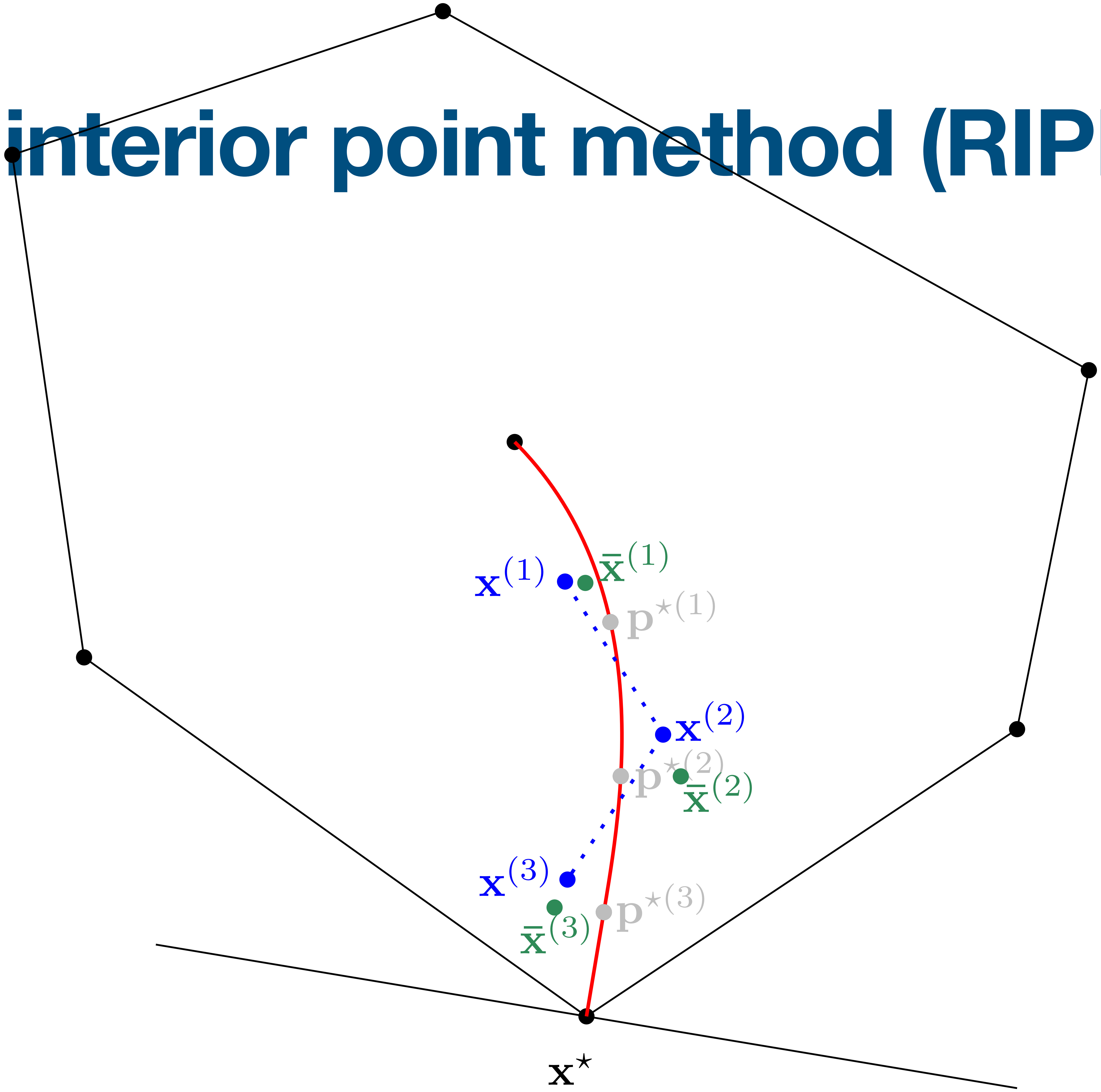
Robust interior point method (RIPM)



Robust interior point method (RIPM)

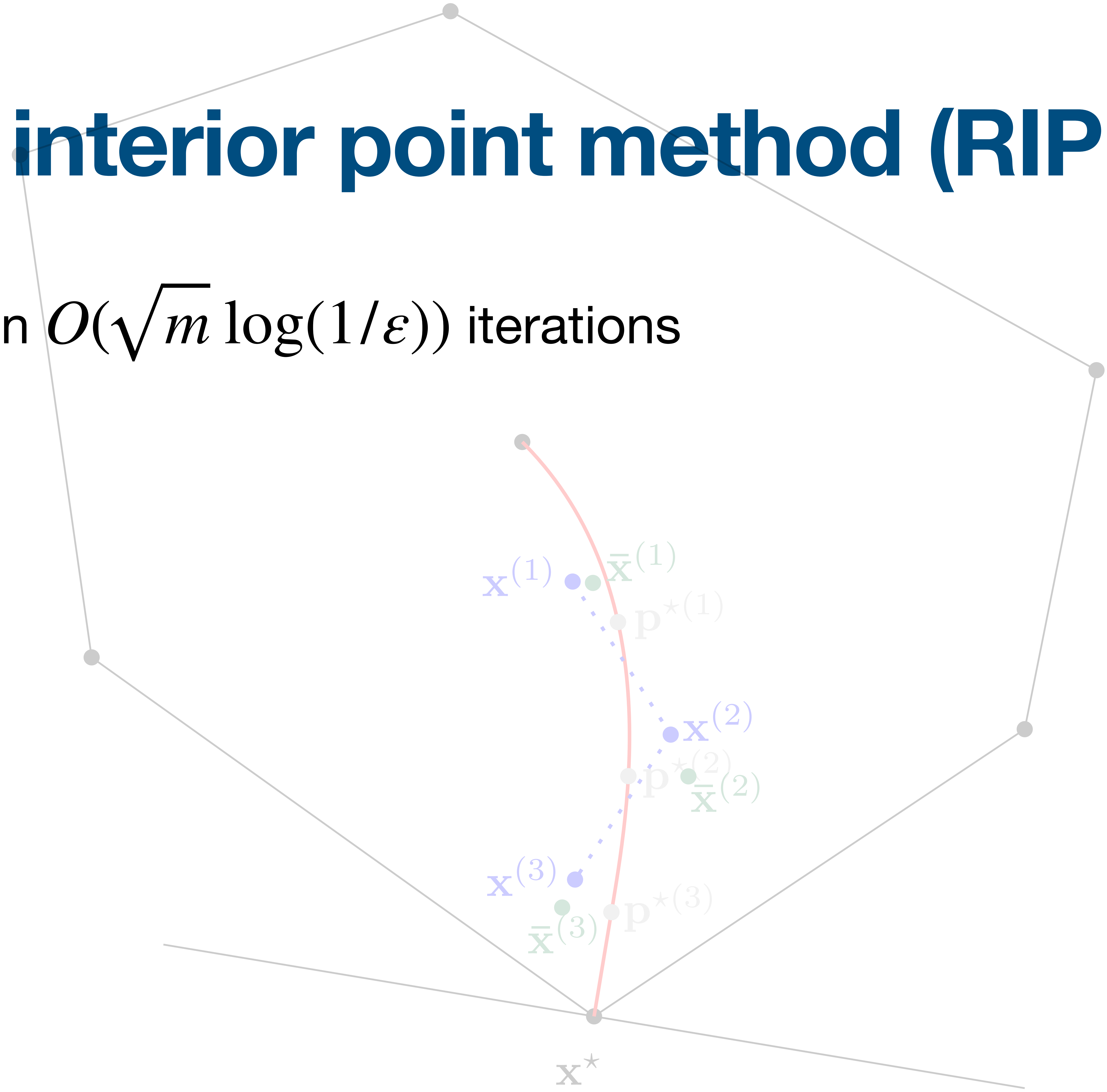


Robust interior point method (RIPM)



Robust interior point method (RIPM)

- converges in $O(\sqrt{m}\log(1/\varepsilon))$ iterations



Robust interior point method (RIPM)

- converges in $O(\sqrt{m}\log(1/\varepsilon))$ iterations
- guarantee: steps have bounded 2-norm

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$
   - $\delta_{\boldsymbol{x}}$ is of the form $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are functions of $\overline{\boldsymbol{x}}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$
   - $\delta_{\boldsymbol{x}}$ is of the form $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are functions of $\overline{\boldsymbol{x}}$

$$\mathbf{P}_{\boldsymbol{w}} \approx \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

is an (approximate) ℓ_2-projection onto feasible subspace

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$
   - $\delta_{\boldsymbol{x}}$ is of the form $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are functions of $\overline{\boldsymbol{x}}$

     $$\mathbf{P}_{\boldsymbol{w}} \approx \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

     is an (approximate) ℓ_2-projection onto feasible subspace
   - technique: matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$
   - $\delta_{\boldsymbol{x}}$ is of the form $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are functions of $\overline{\boldsymbol{x}}$

     $$\mathbf{P}_{\boldsymbol{w}} \approx \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

     is an (approximate) ℓ_2-projection onto feasible subspace
   - technique: matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - technique: find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - at every step, update $\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta_{\boldsymbol{x}}$
   - $\delta_{\boldsymbol{x}}$ is of the form $\mathbf{P}_{\boldsymbol{w}}\boldsymbol{v}$, where $\boldsymbol{v}$ and $\boldsymbol{w}$ are functions of $\overline{\boldsymbol{x}}$

     $$\mathbf{P}_{\boldsymbol{w}} \approx \mathbf{W}^{1/2}\mathbf{A}^{\top}(\mathbf{A}\mathbf{W}\mathbf{A}^{\top})^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

     is an (approximate) ℓ_2-projection onto feasible subspace
   - technique: matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - technique: find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$
2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$

"multiscale representation" [DLY21]

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$
2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

Theorem: If the update to $\boldsymbol{x}$ at every step is of the form

$$\delta_{\boldsymbol{x}} := \mathbf{P}_w \boldsymbol{v} = \Delta \nabla \boldsymbol{v},$$

then we have efficient data structures for everything.

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$

2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

tree operator, inverse tree operator (function of $\boldsymbol{w}$)

Theorem: If the update to $\boldsymbol{x}$ at every step is of the form

$$\delta_{\boldsymbol{x}} := \mathbf{P}_w \boldsymbol{v} = \overbrace{\Delta\,\nabla}\ \boldsymbol{v},$$

then we have efficient data structures for everything.

RIPM for LPs reduces to 2 problems…

1) Dynamic algorithm to maintain the current solution $\boldsymbol{x}$
   - matrix/inverse maintenance + implicit representation of $\boldsymbol{x}$
2) Dynamic algorithm to maintain coordinate-wise approximation $\overline{\boldsymbol{x}}$
   - find heavy-hitters in $\boldsymbol{x}$ *(which is represented implicitly)*

tree operator, inverse tree operator (function of $\boldsymbol{w}$)

Theorem: If the update to $\boldsymbol{x}$ at every step is of the form

$$\delta_{\boldsymbol{x}} := \mathbf{P}_{w}\boldsymbol{v} = \overbrace{\Delta\,\nabla}\ \boldsymbol{v},$$

then we have efficient data structures for everything.

efficiency depends on Δ, ∇

Defining the tree operators

Balanced separators

Given graph $G = (V, E)$, $b \in (0,1)$, and a weight assignment $\boldsymbol{p}$ to the vertices.

A vertex set S is a (b-)balanced separator of G (with respect to $\boldsymbol{p}$) if $G \backslash S$ gives disconnected components A, B, both containing at most b-fraction of the total weight.

Balanced separators

Given graph $G = (V, E)$, $b \in (0,1)$, and a weight assignment $\boldsymbol{p}$ to the vertices.

A vertex set S is a (b-)balanced separator of G (with respect to $\boldsymbol{p}$) if $G \backslash S$ gives disconnected components A, B, both containing at most b-fraction of the total weight.

Theorems:

- (Lipton-Tarjan, 79) Planar graphs have size $O(\sqrt{n})$ separators

Balanced separators

Given graph $G = (V, E)$, $b \in (0,1)$, and a weight assignment $\boldsymbol{p}$ to the vertices.

A vertex set S is a (b-)balanced separator of G (with respect to $\boldsymbol{p}$) if $G \backslash S$ gives disconnected components A, B, both containing at most b-fraction of the total weight.

Theorems:

- (Lipton-Tarjan, 79) Planar graphs have size $O(\sqrt{n})$ separators
- For $0 \leq \alpha \leq 1$, n^α-separable graphs have size n^α separators

Balanced separators

Given graph $G = (V, E)$, $b \in (0,1)$, and a weight assignment $\boldsymbol{p}$ to the vertices.

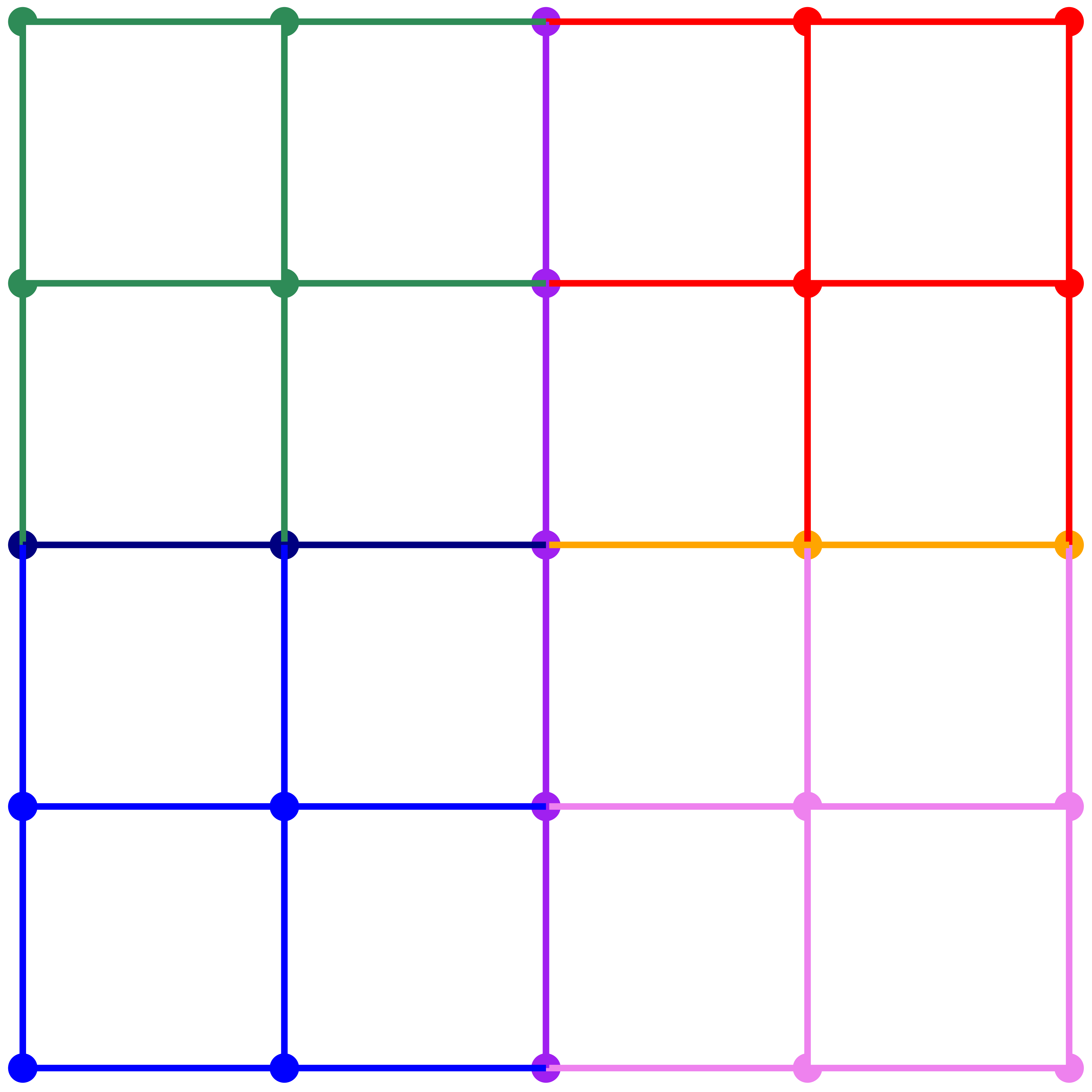
A vertex set S is a (b-)balanced separator of G (with respect to $\boldsymbol{p}$) if $G \backslash S$ gives disconnected components A, B, both containing at most b-fraction of the total weight.

Theorems:

- (Lipton-Tarjan, 79) Planar graphs have size $O(\sqrt{n})$ separators
- For $0 \leq \alpha \leq 1$, n^α-separable graphs have size n^α separators
- Treewidth t graphs have a size t separators

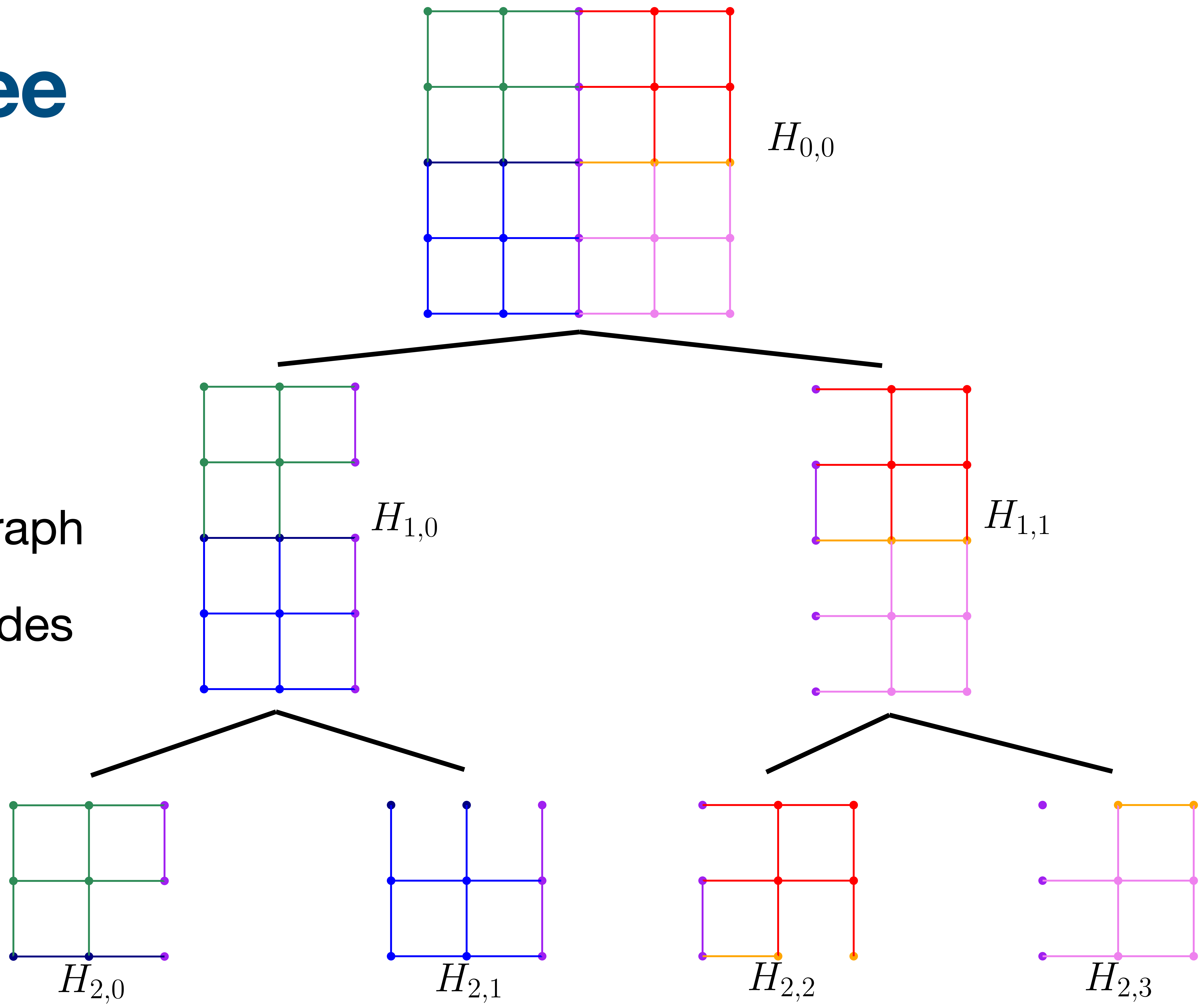
Use balanced separators to decompose graph

- planar graph G on n vertices
- recursively use balanced separator
- decompose until there is no more non-trivial balanced separator



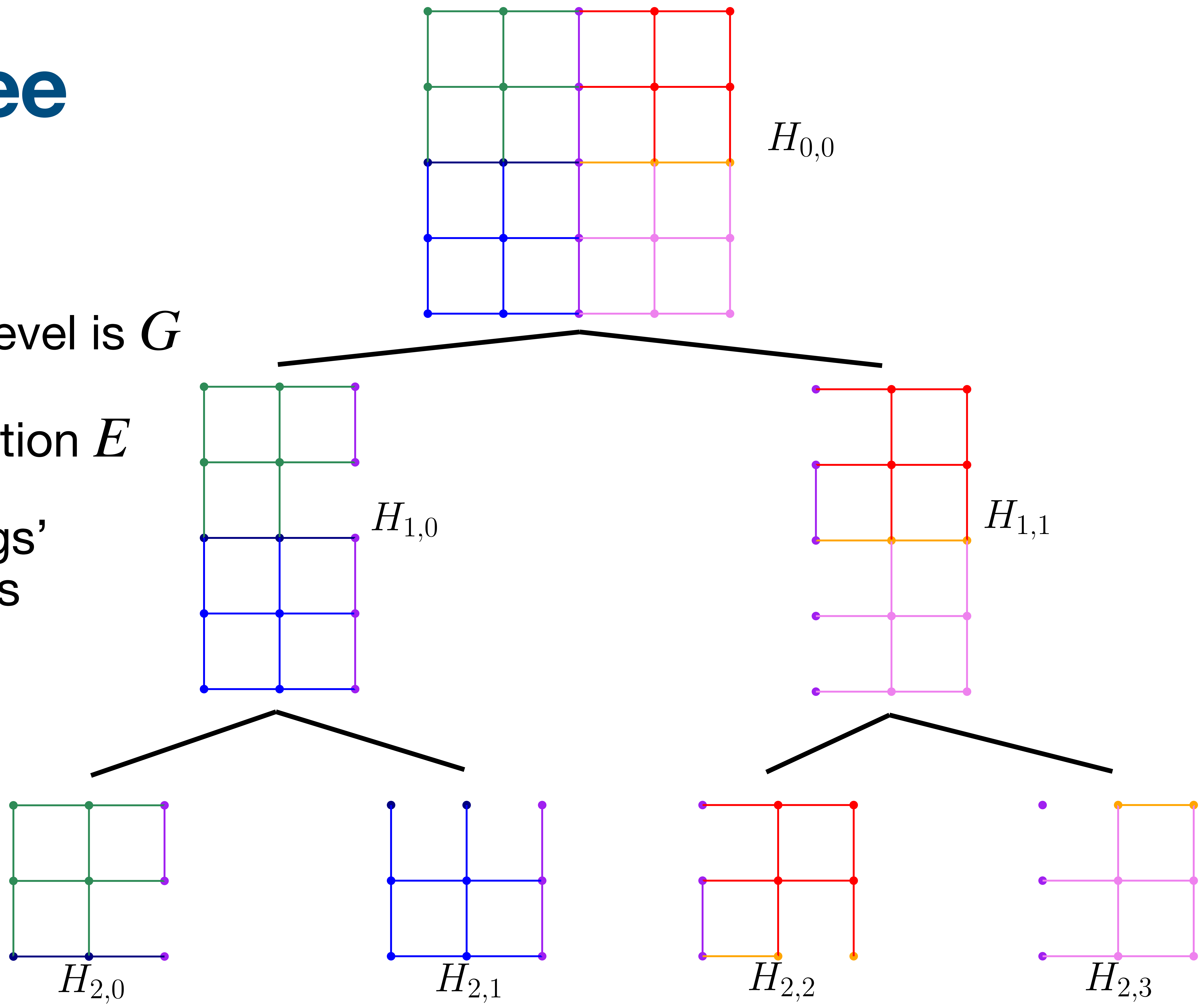
Separator tree

- height $\eta = O(\log n)$
- constant degree
- each node is a subgraph
- constant size leaf nodes



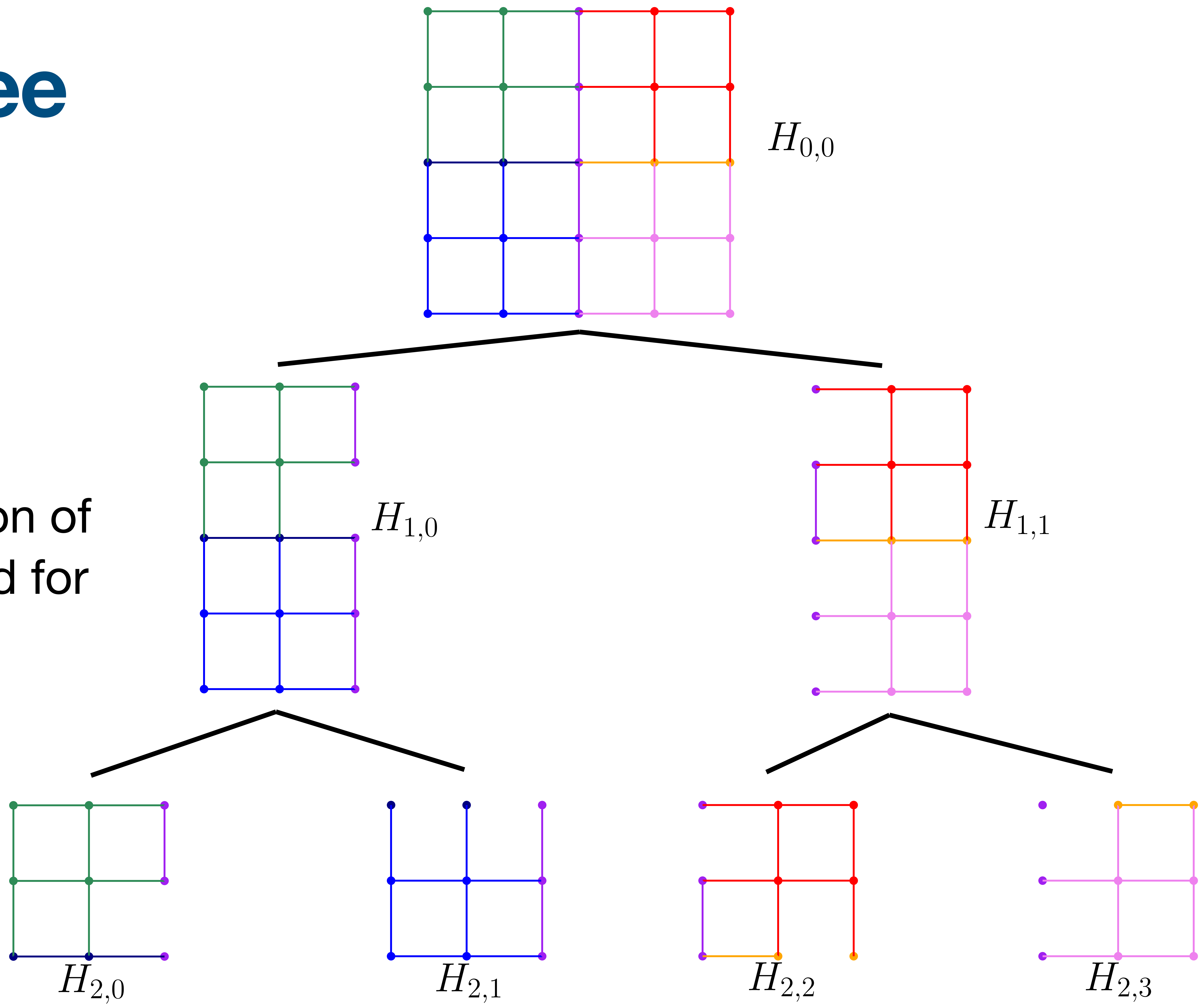
Separator tree

- union of nodes at a level is G
- nodes at a level partition E
- intersection of siblings' vertex sets is parent's separator



Separator tree

- boundary set
- separator
- gives natural definition of $V_{i,j}$'s and E_i's needed for the tree operator



Recursive Cholesky decomposition

- Recursively factor the symmetric matrix $\mathbf{L} = \mathbf{A}\mathbf{W}\mathbf{A}^\top$ based on separator tree

Recursive Cholesky decomposition

- Recursively factor the symmetric matrix $\mathbf{L} = \mathbf{A}\mathbf{W}\mathbf{A}^\top$ based on separator tree
- Rows and columns of $\mathbf{L}$ are indexed by vertices of $G_{\mathbf{A}}$

Recursive Cholesky decomposition

- Recursively factor the symmetric matrix $\mathbf{L} = \mathbf{A}\mathbf{W}\mathbf{A}^\top$ based on separator tree
- Rows and columns of $\mathbf{L}$ are indexed by vertices of $G_{\mathbf{A}}$
- Partition vertex set into F, C, then

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{L}_{FC} \\ \mathbf{L}_{CF} & \mathbf{L}_{CC} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & 0 \\ \mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C) \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ 0 & \mathbf{I} \end{pmatrix}.$$

Recursive Cholesky decomposition

- Recursively factor the symmetric matrix $\mathbf{L} = \mathbf{A}\mathbf{W}\mathbf{A}^\top$ based on separator tree
- Rows and columns of $\mathbf{L}$ are indexed by vertices of $G_{\mathbf{A}}$
- Partition vertex set into F, C, then

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{L}_{FC} \\ \mathbf{L}_{CF} & \mathbf{L}_{CC} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & 0 \\ \mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C) \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ 0 & \mathbf{I} \end{pmatrix}.$$

Schur complement of $\mathbf{L}$ onto C
is supported on C

Recursive Cholesky decomposition

- Recursively factor the symmetric matrix $\mathbf{L} = \mathbf{AWA}^\top$ based on separator tree
- Rows and columns of $\mathbf{L}$ are indexed by vertices of $G_{\mathbf{A}}$
- Partition vertex set into F, C, then

Recursive partitions defined using separator tree

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{L}_{FC} \\ \mathbf{L}_{CF} & \mathbf{L}_{CC} \end{pmatrix} = \begin{pmatrix} \mathbf{I} & 0 \\ \mathbf{L}_{CF}\mathbf{L}_{FF}^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{L}_{FF} & \mathbf{0} \\ \mathbf{0} & \mathbf{Sc}(\mathbf{L}, C) \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{L}_{FF}^{-1}\mathbf{L}_{FC} \\ 0 & \mathbf{I} \end{pmatrix}.$$

Schur complement of $\mathbf{L}$ onto C is supported on C

Useful Schur complement properties (for efficient data structure updates)

Transitivity:

If $X \subseteq Y \subseteq V(G)$, then

$$\mathbf{Sc}(\mathbf{Sc}(\mathbf{L}, Y), X) = \mathbf{Sc}(\mathbf{L}, X)\,.$$

Decomposability:

If $\mathbf{L} = \mathbf{L}_1 + \ldots + \mathbf{L}_k$, and the $\mathbf{L}_i$'s supports intersect on C and are otherwise pairwise disjoint, then

$$\mathbf{Sc}(\mathbf{L}, C) = \mathbf{Sc}(\mathbf{L}_1, C) + \ldots + \mathbf{Sc}(\mathbf{L}_k, C)\,.$$

Decomposition of $\mathbf{P}_w$

$$\mathbf{P}_w = \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

Decomposition of $\mathbf{P}_w$

$$
\begin{aligned}
\mathbf{P}_w &= \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2} \\
&= \mathbf{W}^{1/2}\mathbf{A}^\top\mathbf{\Pi}^{(\eta)\top}\ldots\mathbf{\Pi}^{(1)\top}\mathbf{\Gamma}\mathbf{\Pi}^{(1)}\ldots\mathbf{\Pi}^{(\eta)}\mathbf{A}\mathbf{W}^{1/2}
\end{aligned}
$$

where

$$
\mathbf{\Pi}^{(i)} = \boldsymbol{I} - \sum_{H \in \mathcal{T}(i)} \mathbf{L}^{(H)}_{\partial H, F_H}\left(\mathbf{L}^{(H)}_{F_H, F_H}\right)^{-1}.
$$

Decomposition of $\mathbf{P}_w$

$$\mathbf{P}_w = \mathbf{W}^{1/2}\mathbf{A}^\top(\mathbf{A}\mathbf{W}\mathbf{A}^\top)^{-1}\mathbf{A}\mathbf{W}^{1/2}$$

$$= \underbrace{\mathbf{W}^{1/2}\mathbf{A}^\top\mathbf{\Pi}^{(\eta)\top}\ldots\mathbf{\Pi}^{(1)\top}}_{\Delta}\mathbf{\Gamma}\underbrace{\mathbf{\Pi}^{(1)}\ldots\mathbf{\Pi}^{(\eta)}\mathbf{A}\mathbf{W}^{1/2}}_{\nabla}$$

where

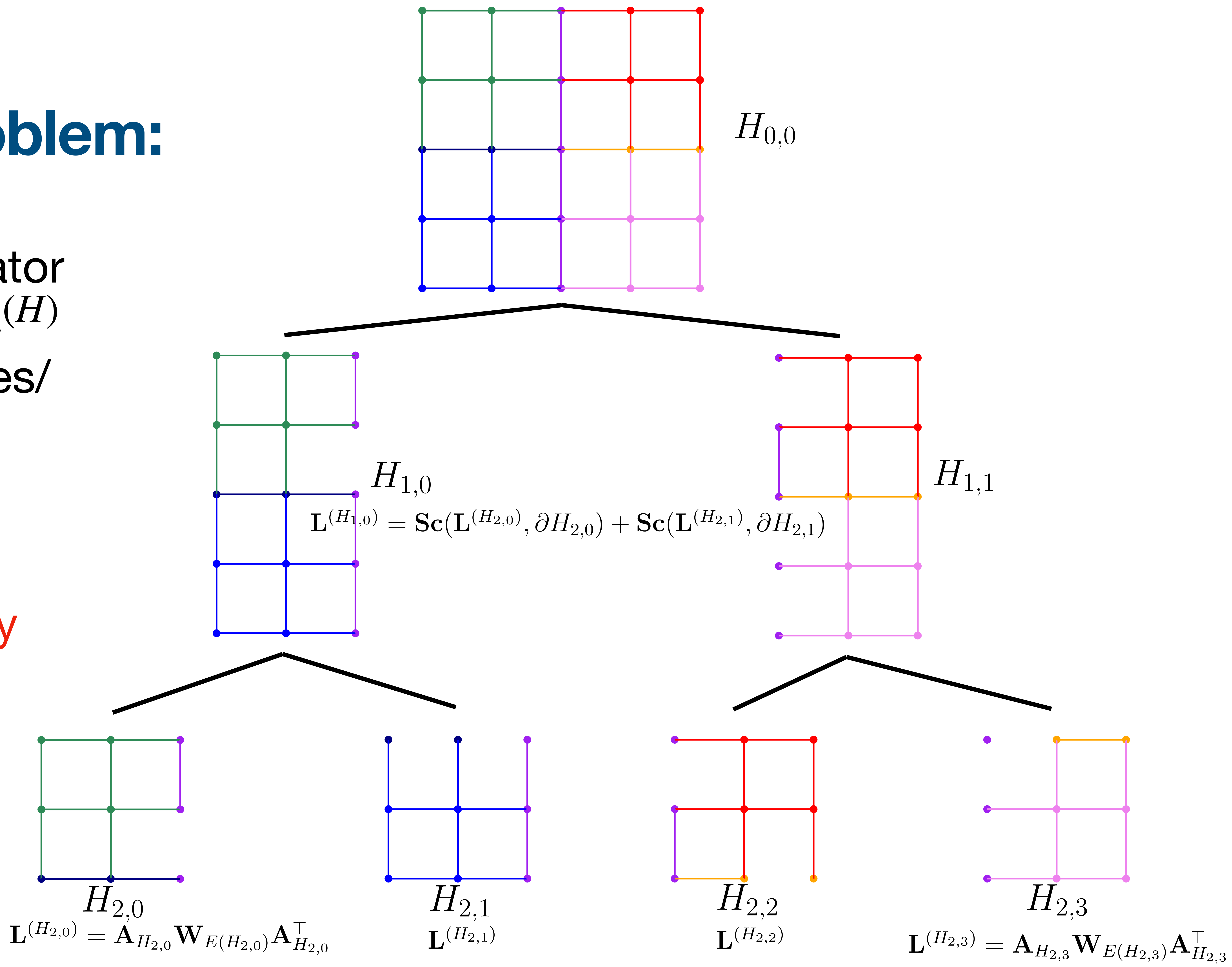
$$\mathbf{\Pi}^{(i)} = \boldsymbol{I} - \sum_{H\in\mathcal{T}(i)} \mathbf{L}^{(H)}_{\partial H,F_H}\left(\mathbf{L}^{(H)}_{F_H,F_H}\right)^{-1}.$$

Can be further decomposed based on edges of separator tree.

Matrix/inverse maintenance problem:

Every node H in separator tree maintains matrix $\mathbf{L}^{(H)}$ and some other matrices/inverses

$\mathbf{L}^{(H)}$ is supported on separator and boundary of H.



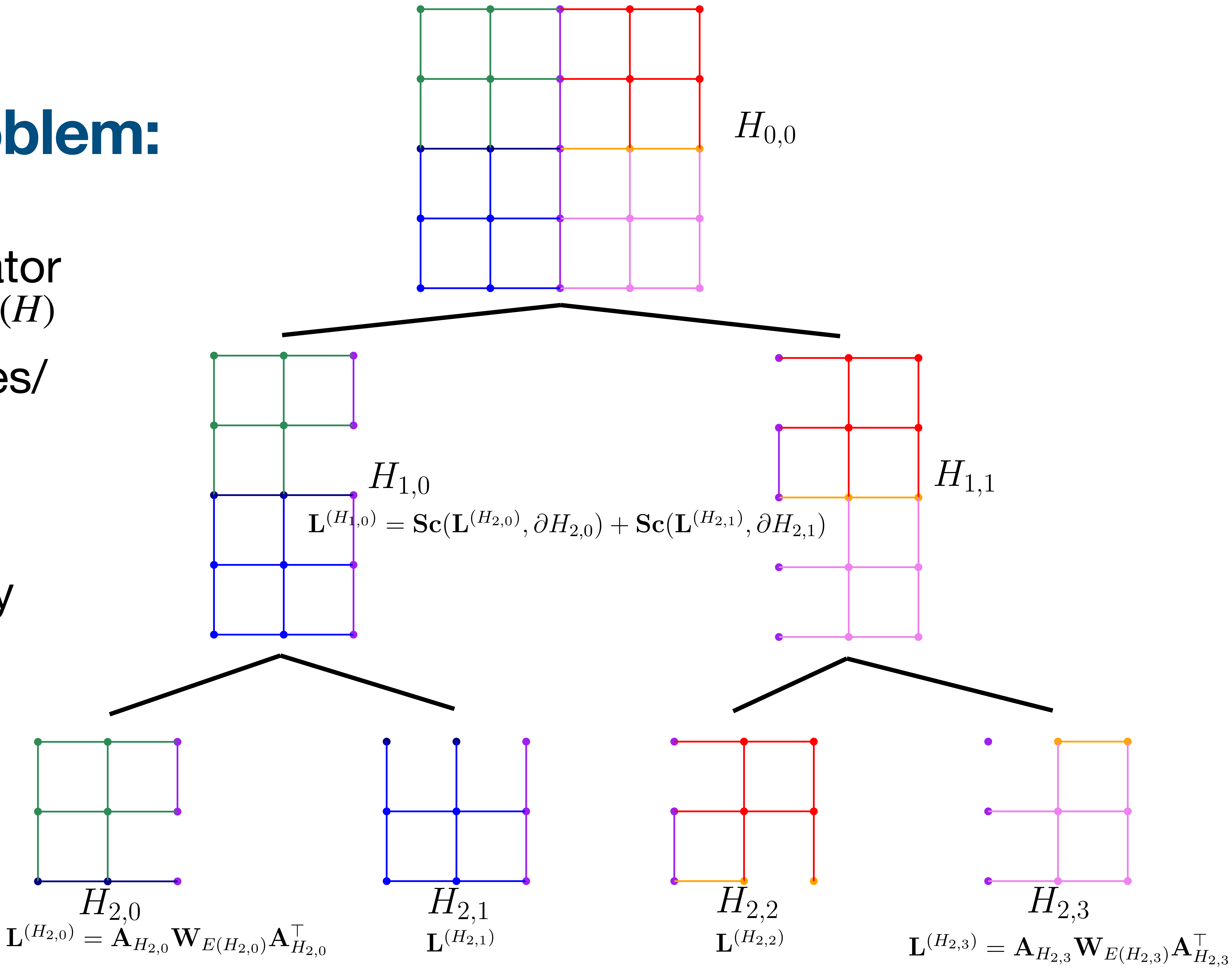
Matrix/inverse maintenance problem:

Every node H in separator tree maintains matrix $\mathbf{L}^{(H)}$ and some other matrices/inverses

$\mathbf{L}^{(H)}$ is supported on separator and boundary of H.

Theorem (DGLSY24):

Efficient algorithm for separable graphs.



Flow problems:
Use approximations to improve runtime

If the LP is a flow problem, then $\mathbf{AWA}^\top$ is a weighted Laplacian.

- [Spielman-Tang, 04] Laplacian solvers in nearly-linear time
- [Kyng-Sachdeva, 16], [Goranci-Henzinger-Peng, 18] sparse, approximate Schur complements in nearly-linear time

Flow problems:
Use approximations to improve runtime

If the LP is a flow problem, then $\mathbf{AWA}^\top$ is a weighted Laplacian.

- [Spielman-Tang, 04] Laplacian solvers in nearly-linear time
- [Kyng-Sachdeva, 16], [Goranci-Henzinger-Peng, 18] sparse, approximate Schur complements in nearly-linear time

Theorem (**D**GGPSY22): Nearly-linear time min-cost flow on planar graphs.

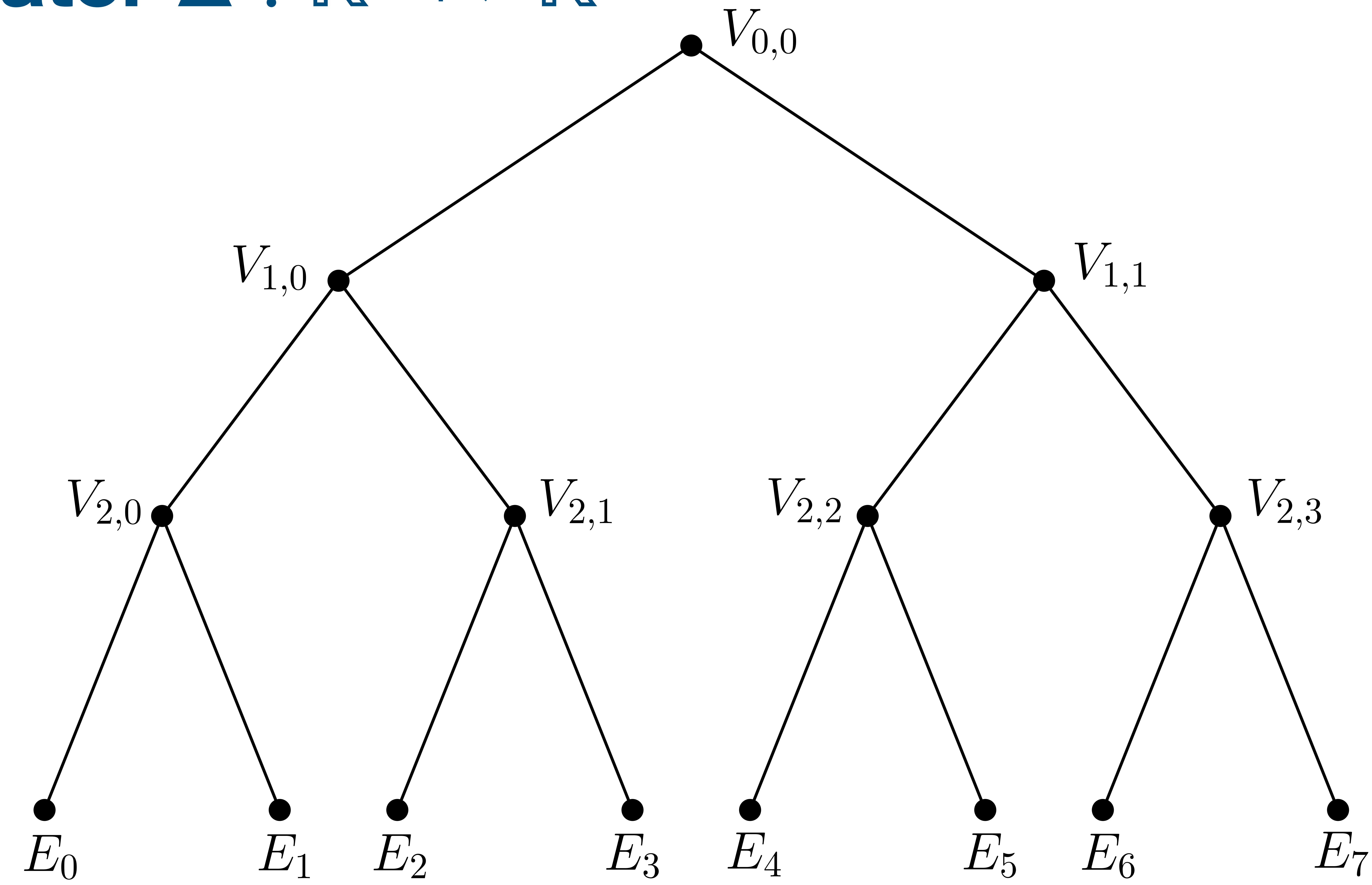
Theorem (**D**Y23+): $\tilde{O}(m\sqrt{t})$ time min-cost flow on treewidth t graphs.

Tree operator

Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

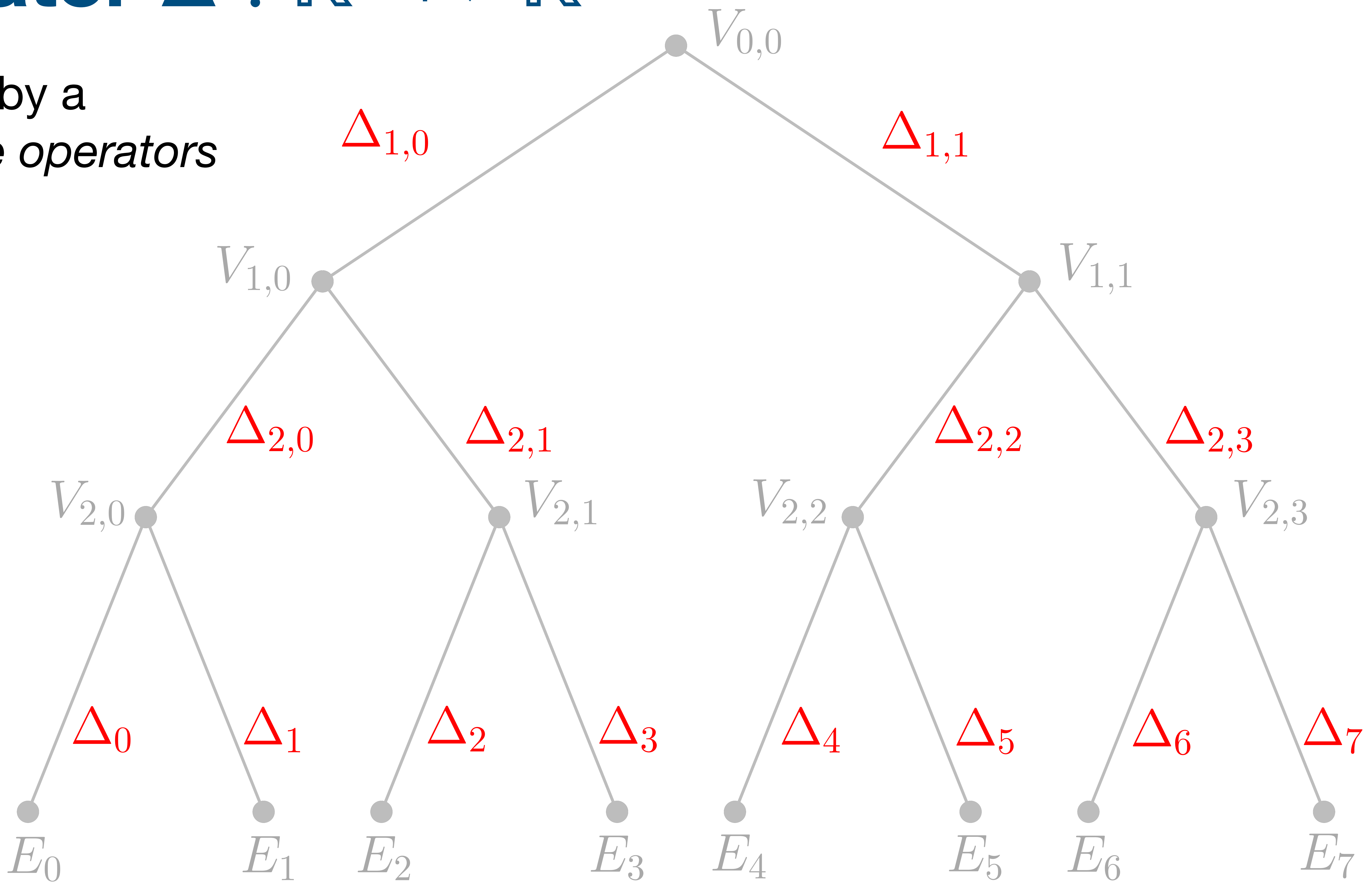
$$V := \dot{\bigcup} V_{i,j}$$

$$E := \dot{\bigcup} E_i$$



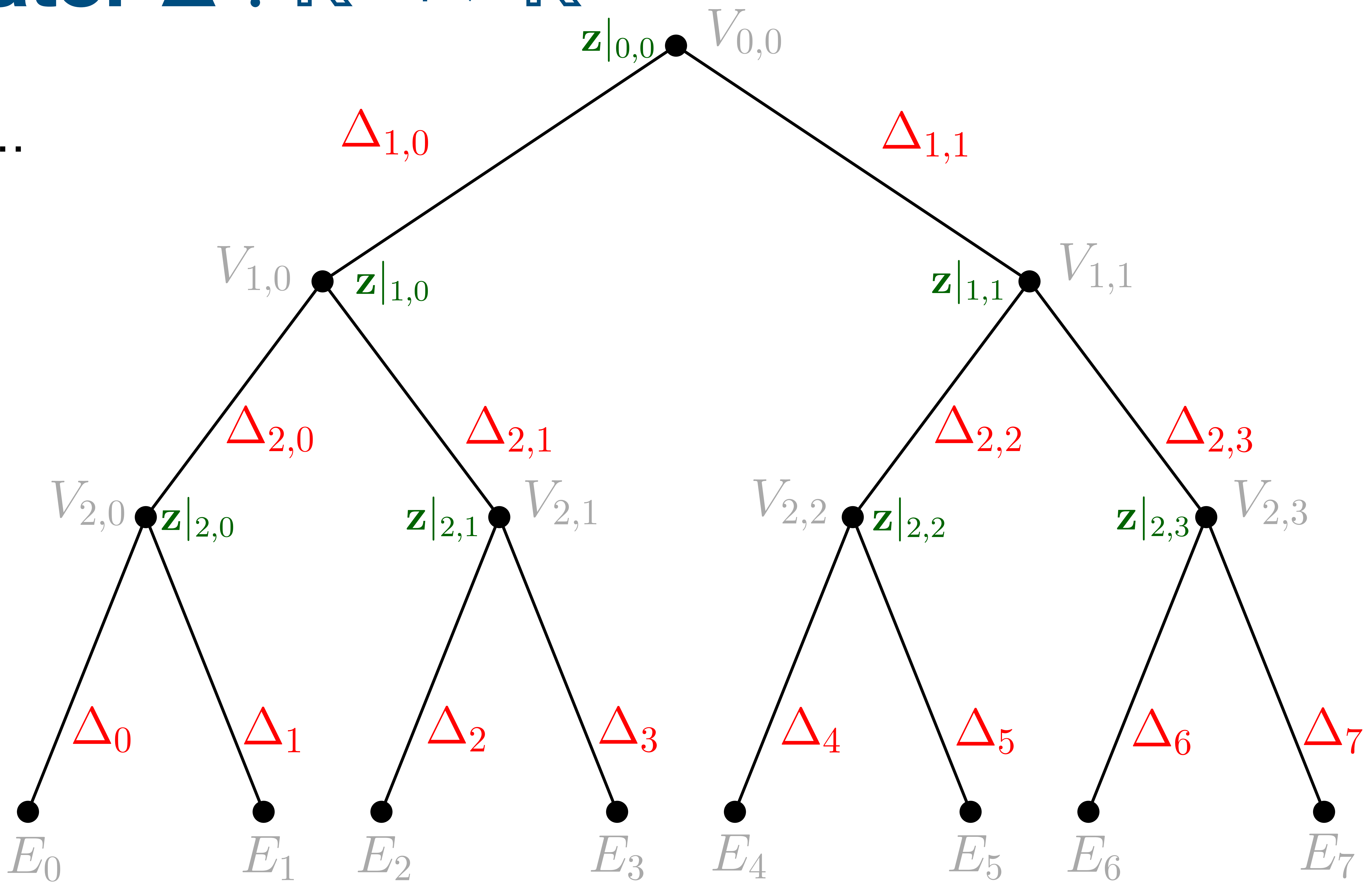
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

Δ is represented by a collection of *edge operators*



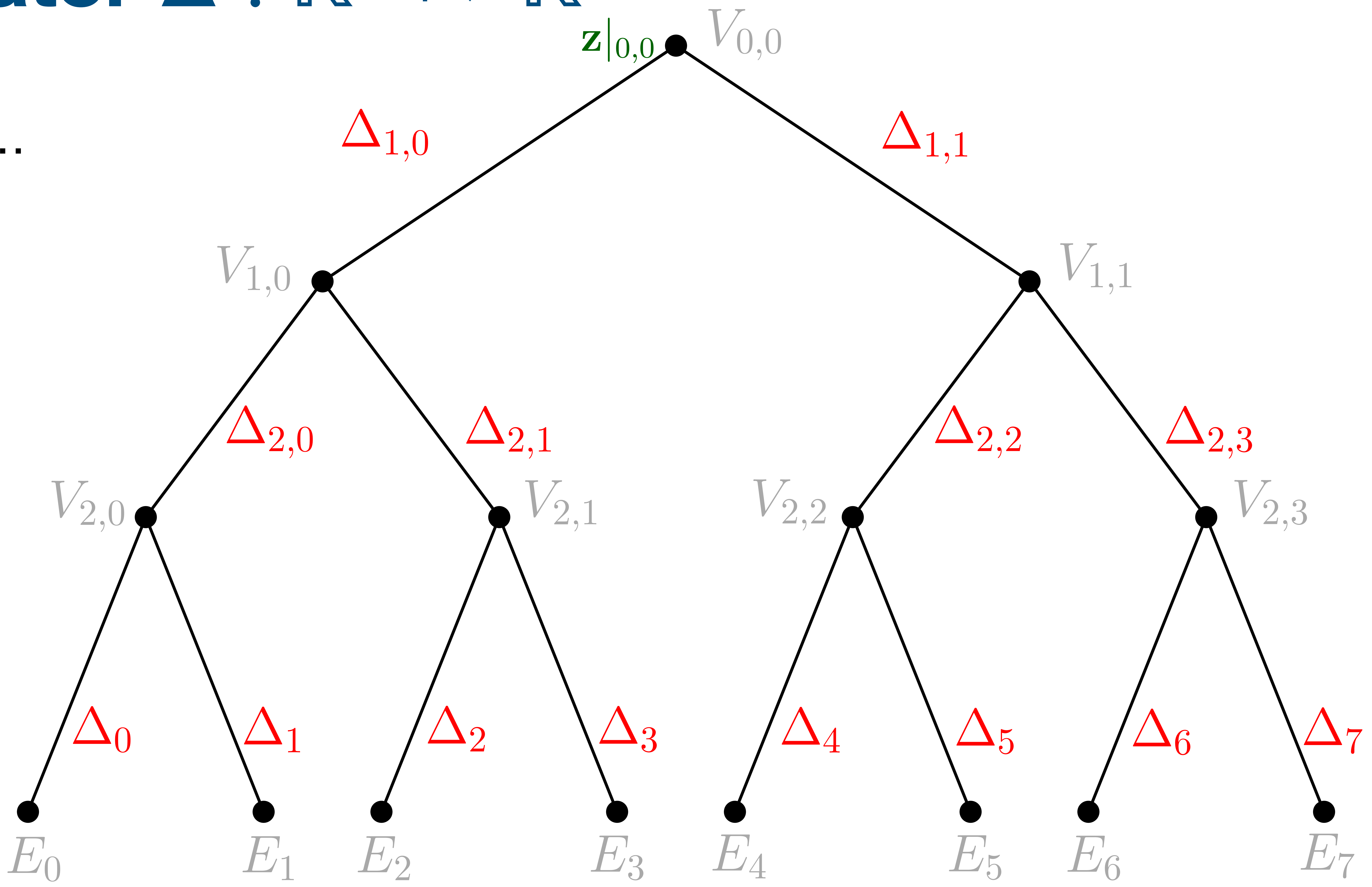
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



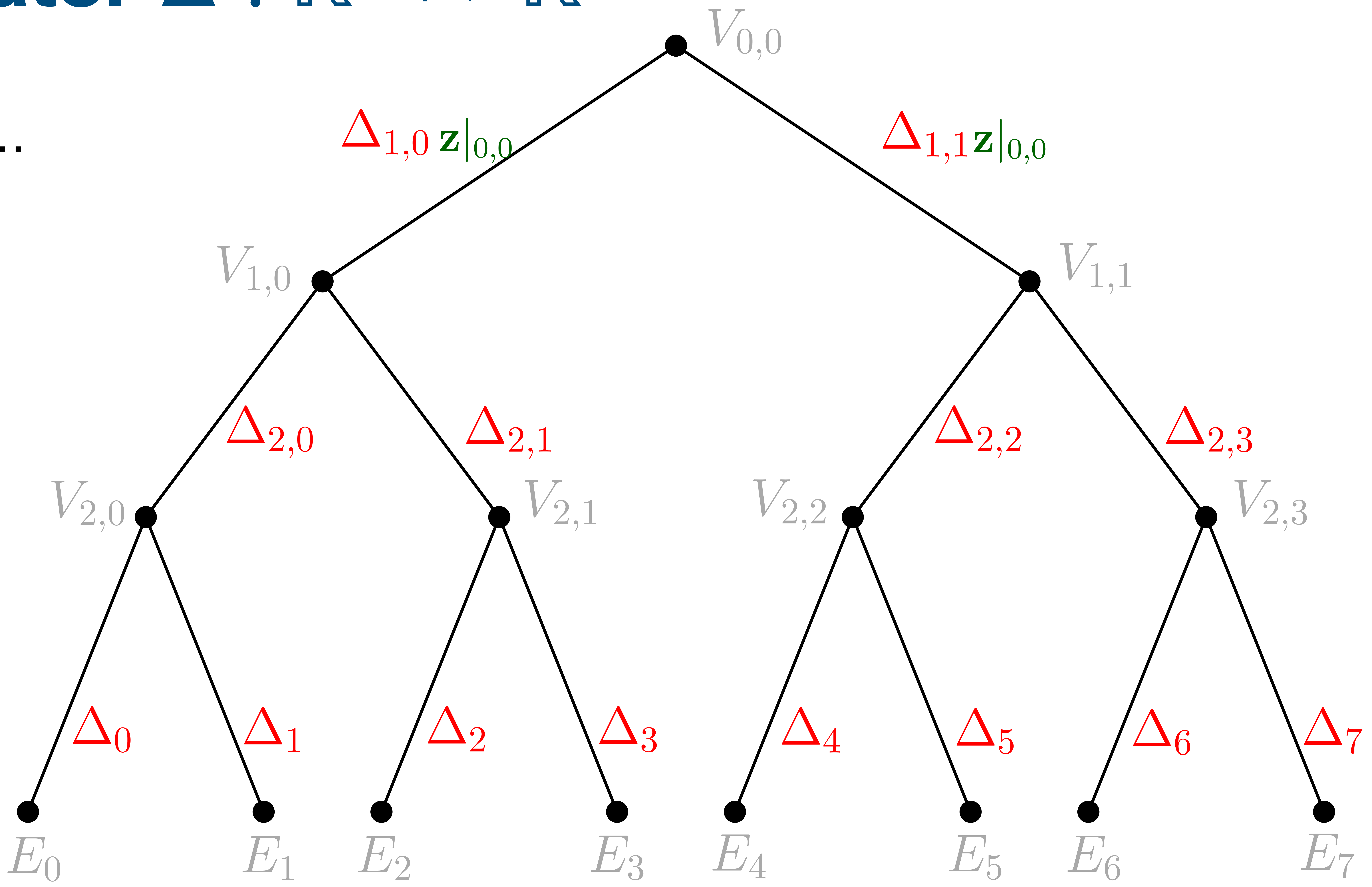
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



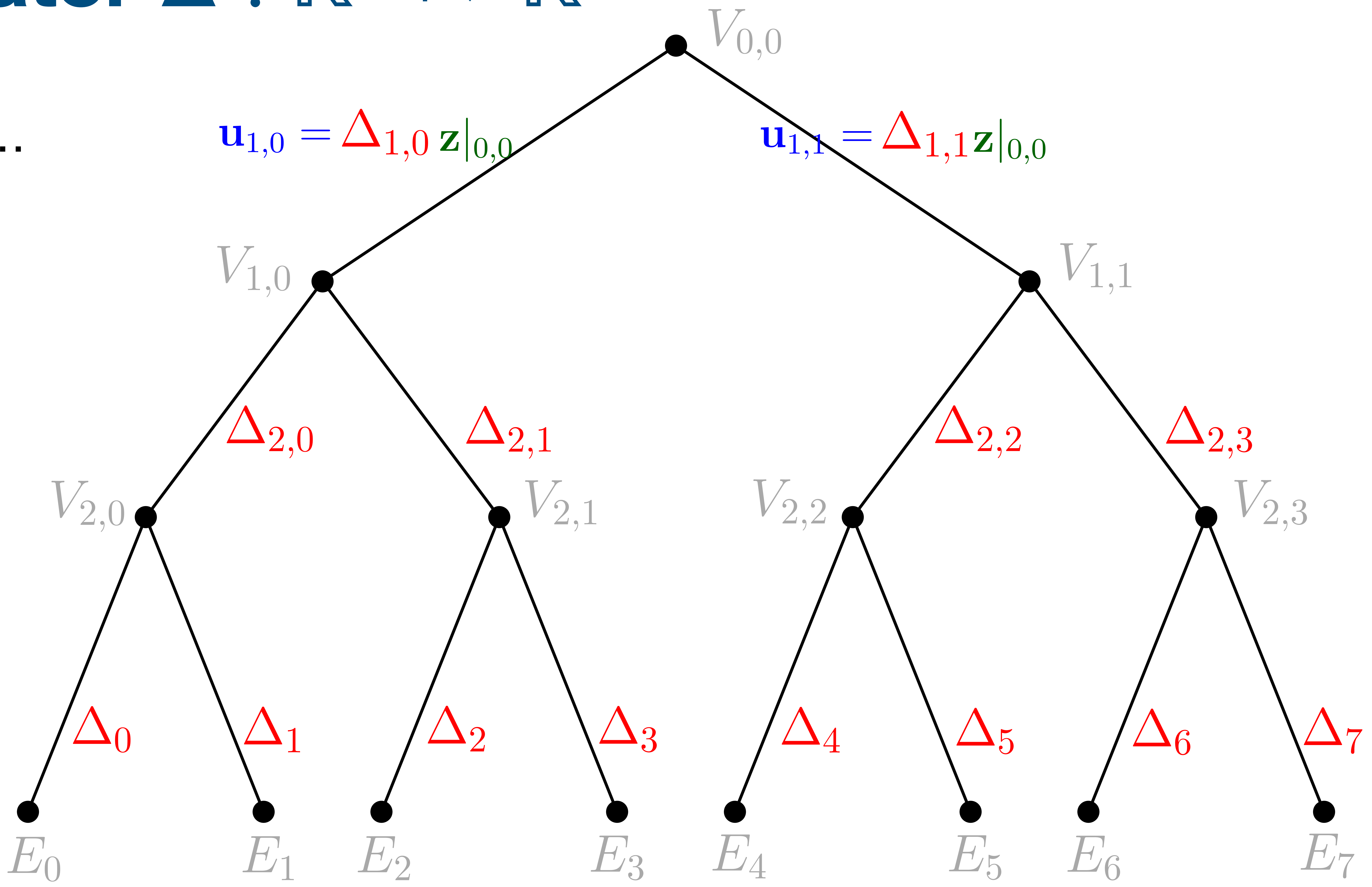
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



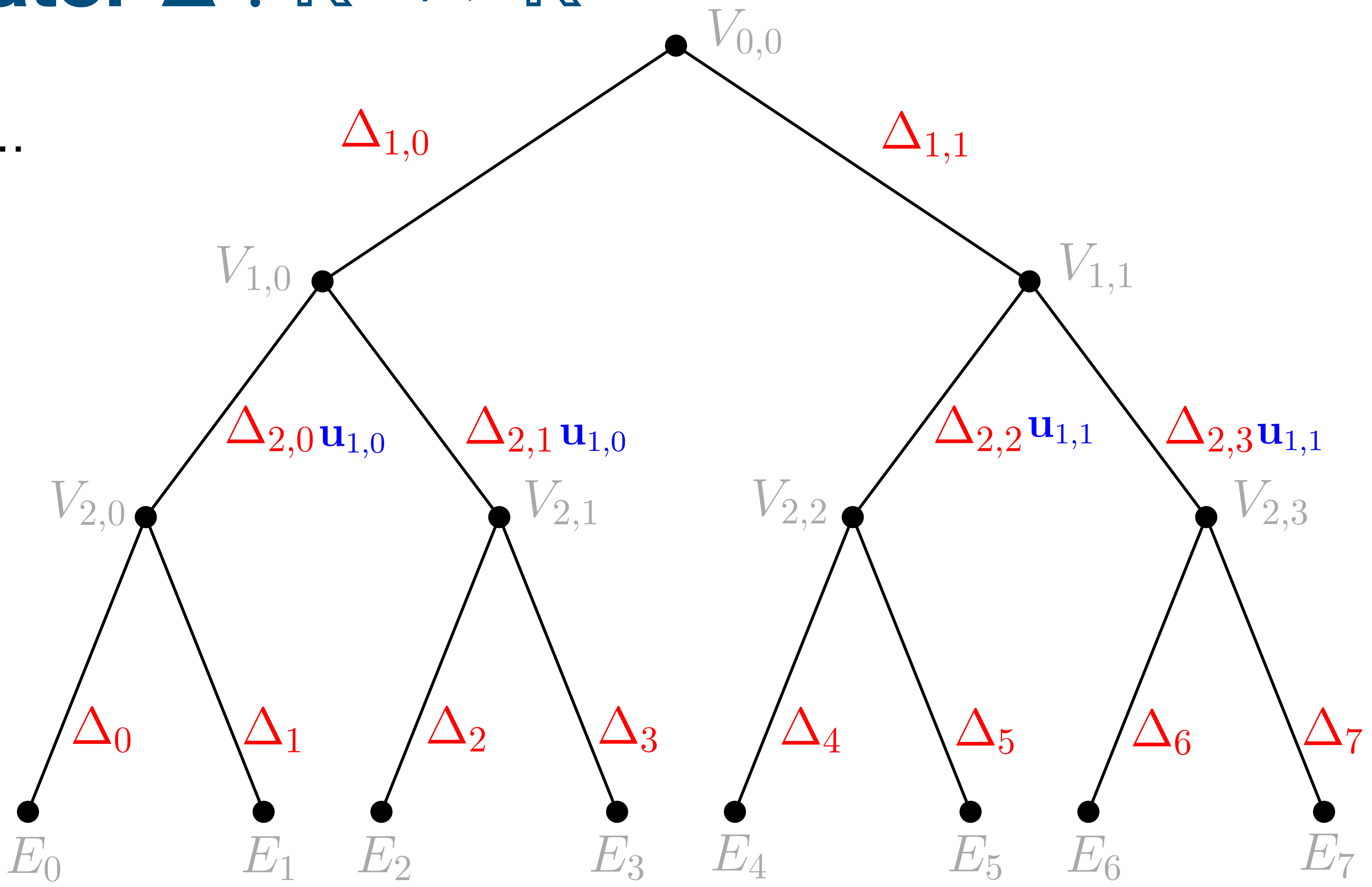
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



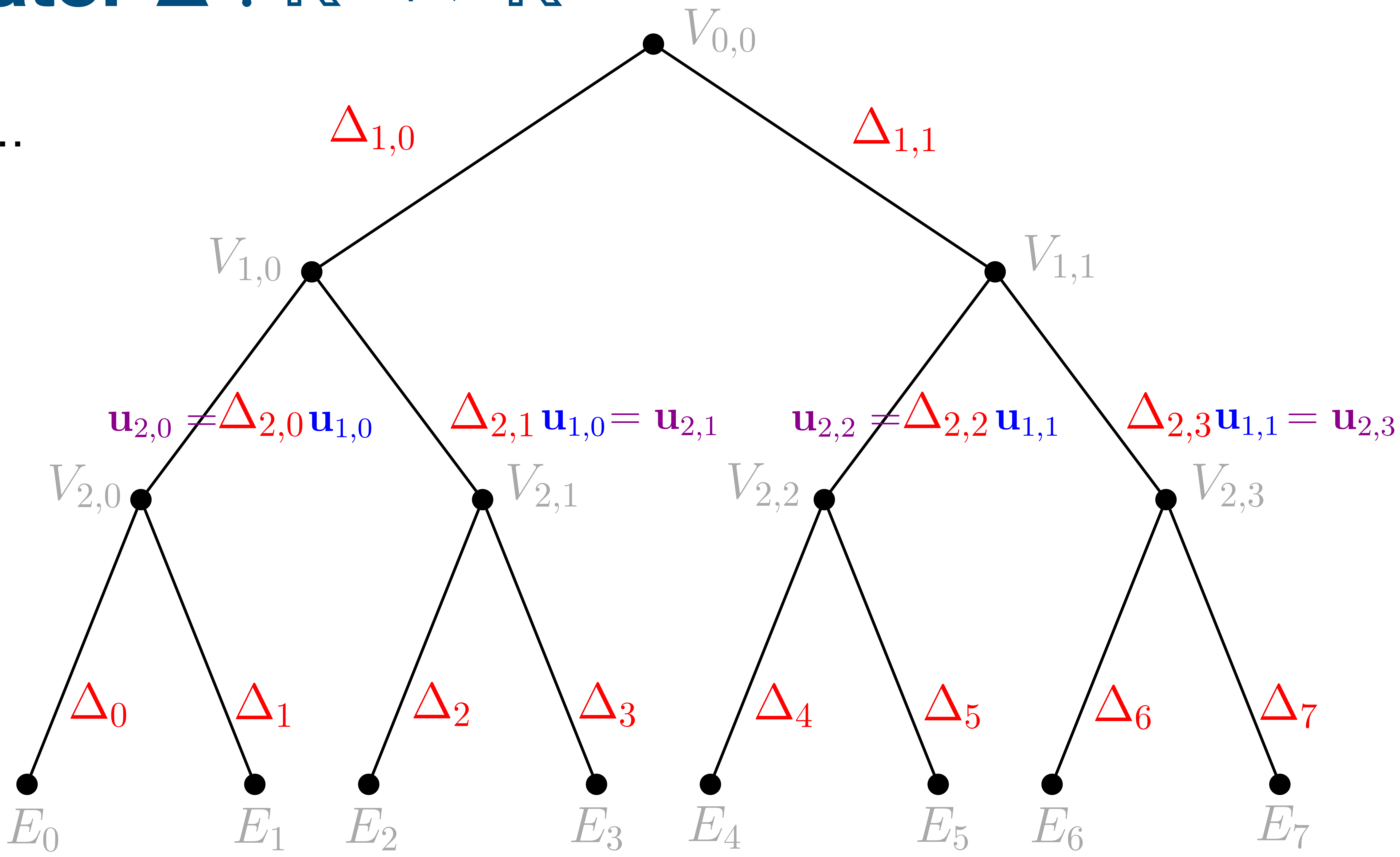
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



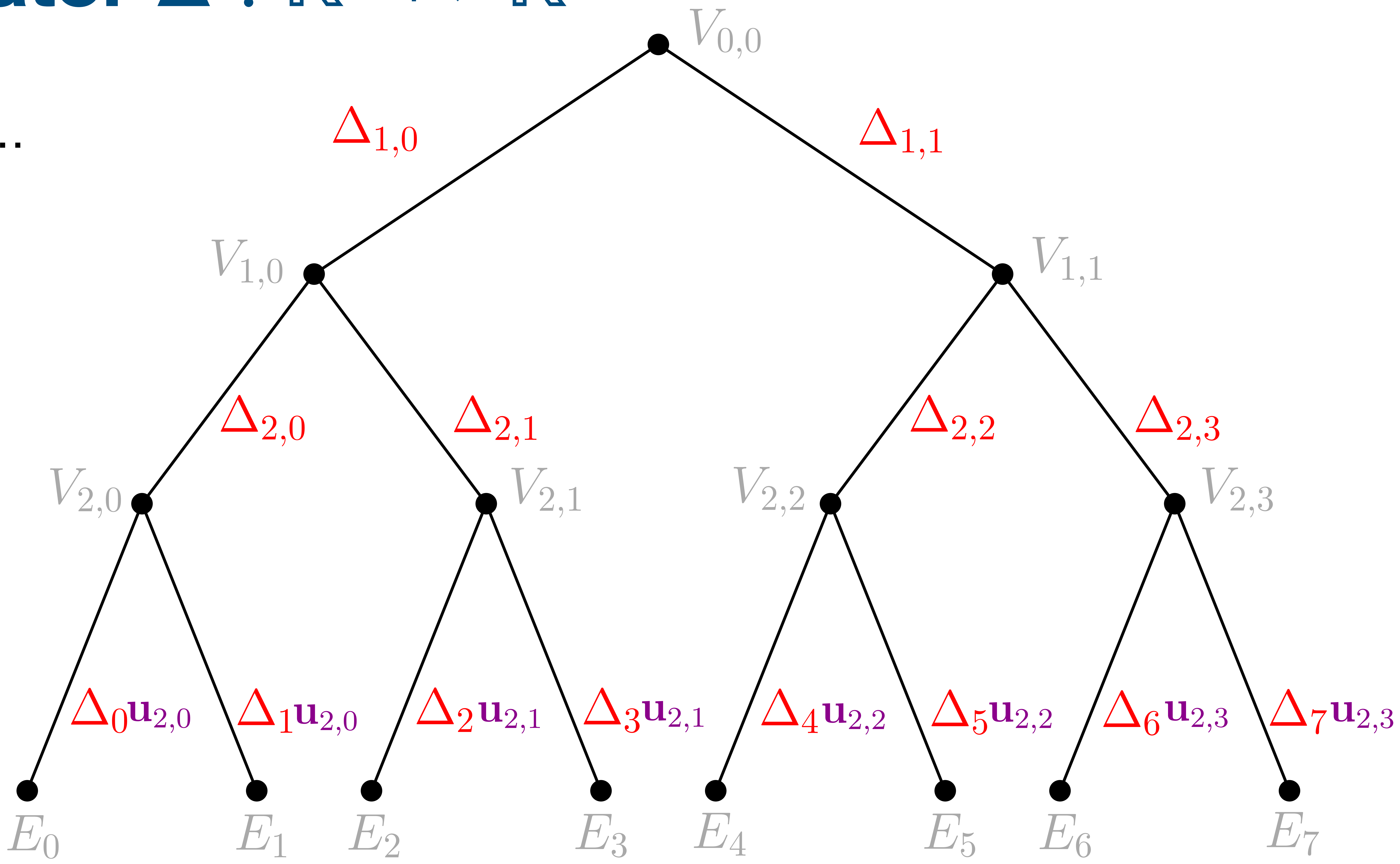
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



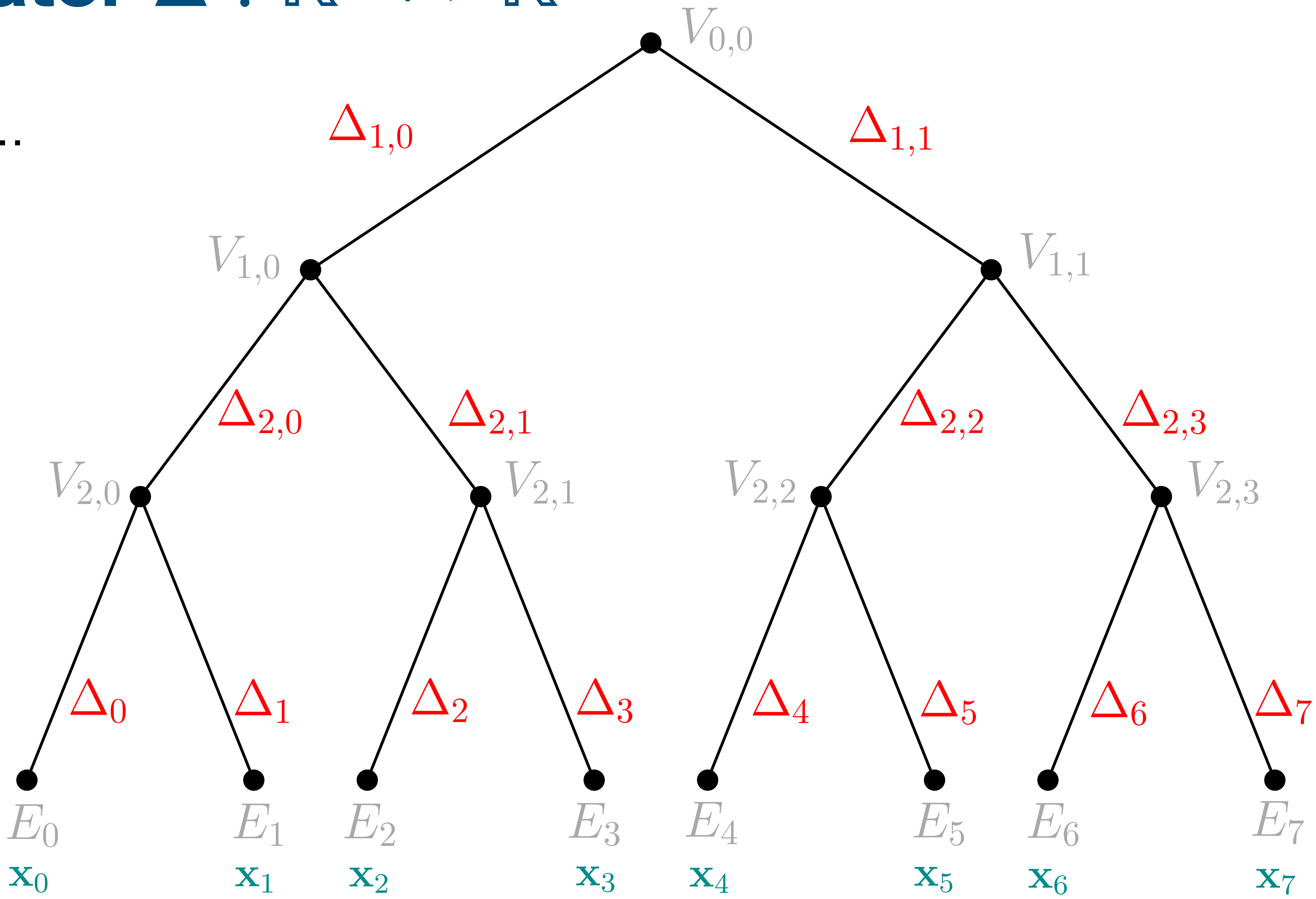
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...



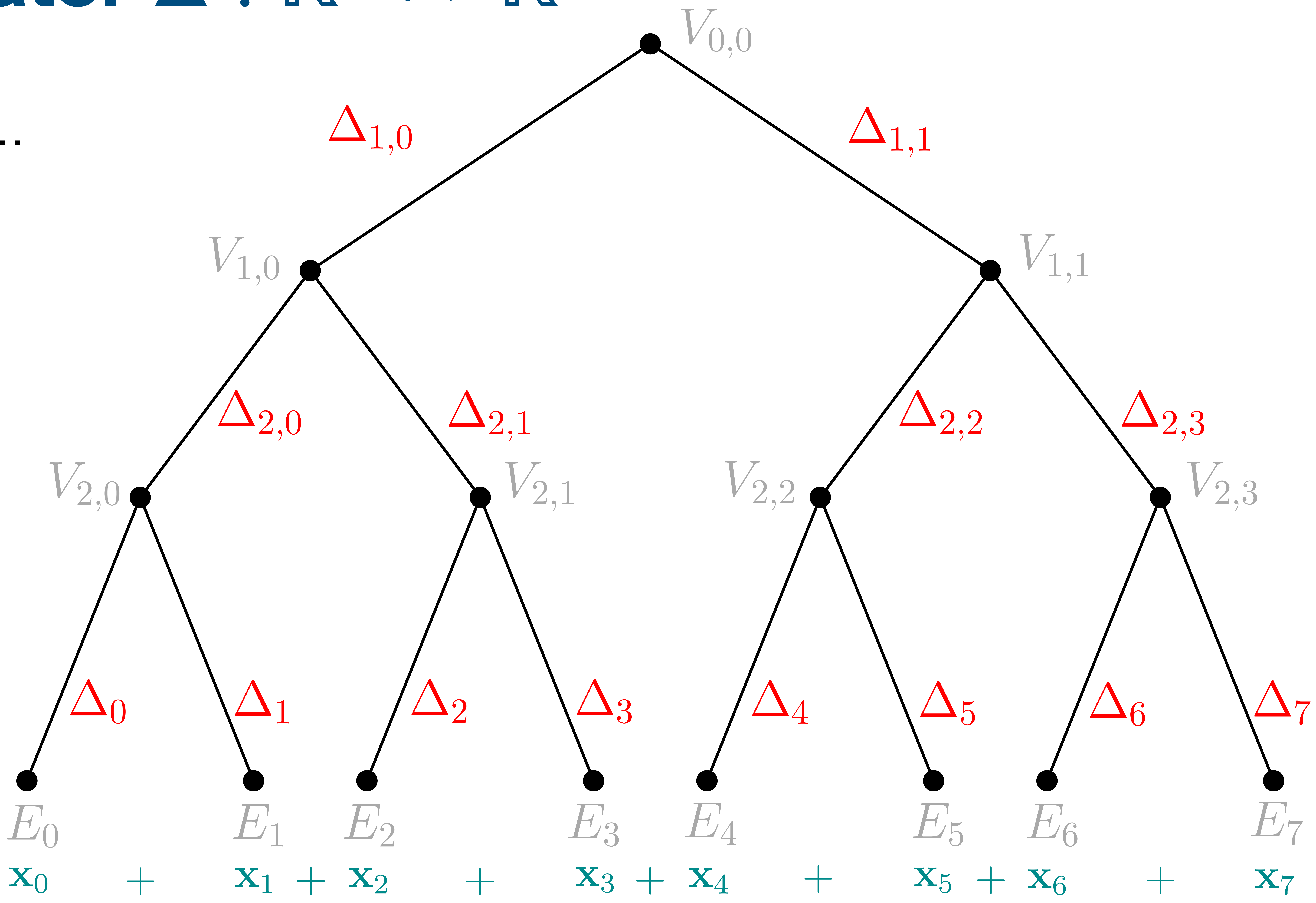
Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta\mathbf{z}$...

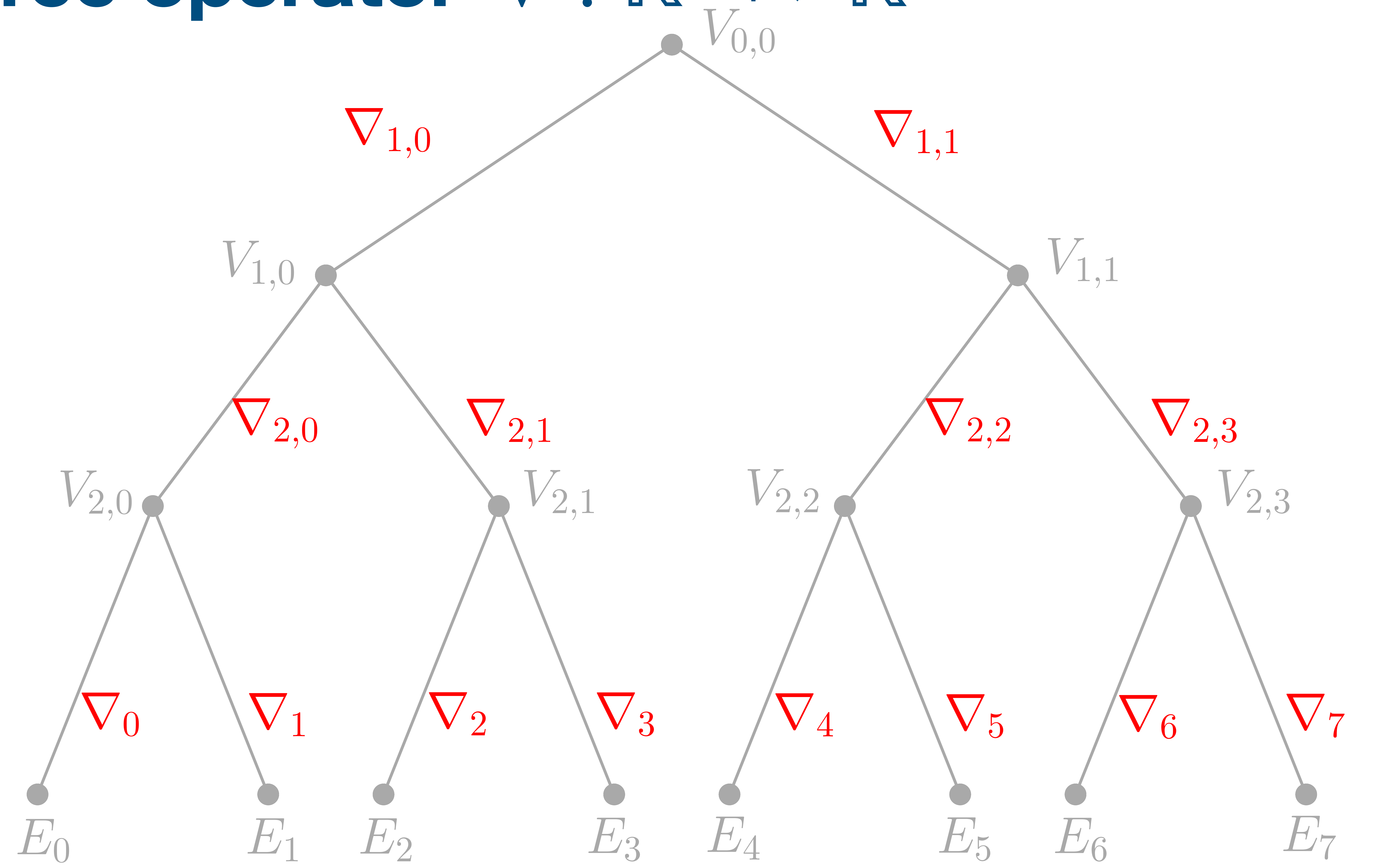


Tree operator $\Delta : \mathbb{R}^V \mapsto \mathbb{R}^E$

To compute $\Delta \mathbf{z}$...

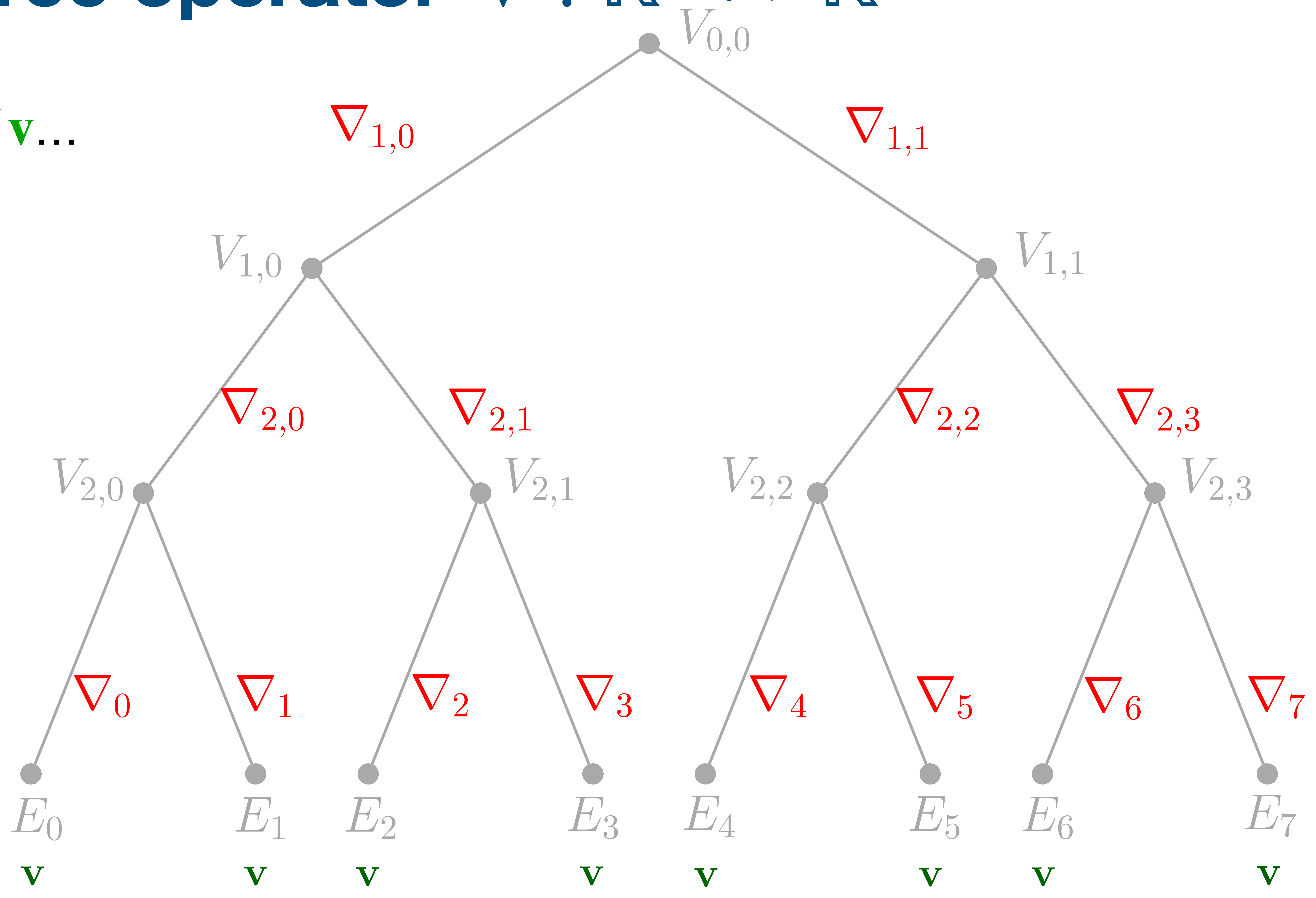


Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$



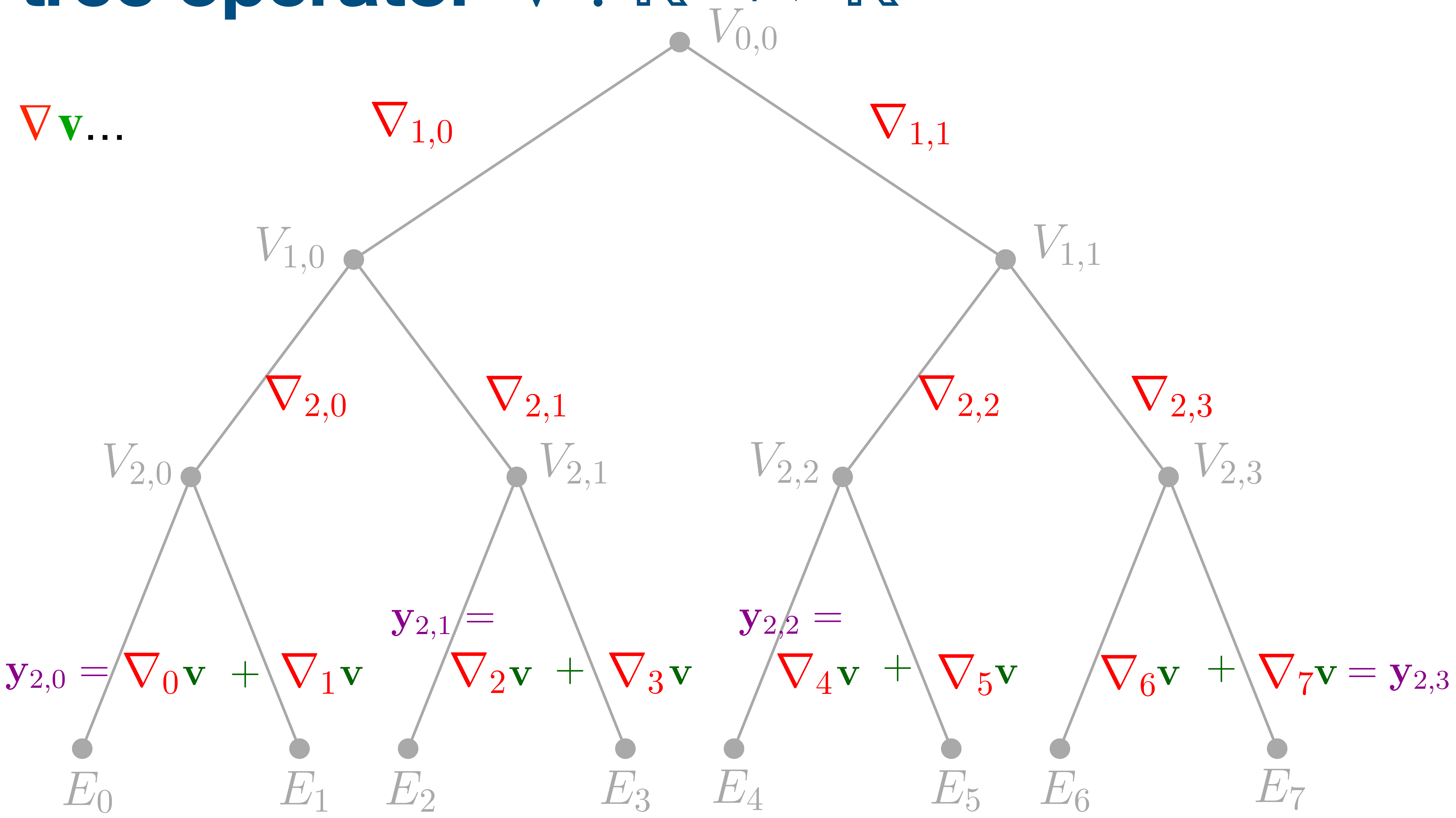
Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$

To compute $\nabla \mathbf{v}$…



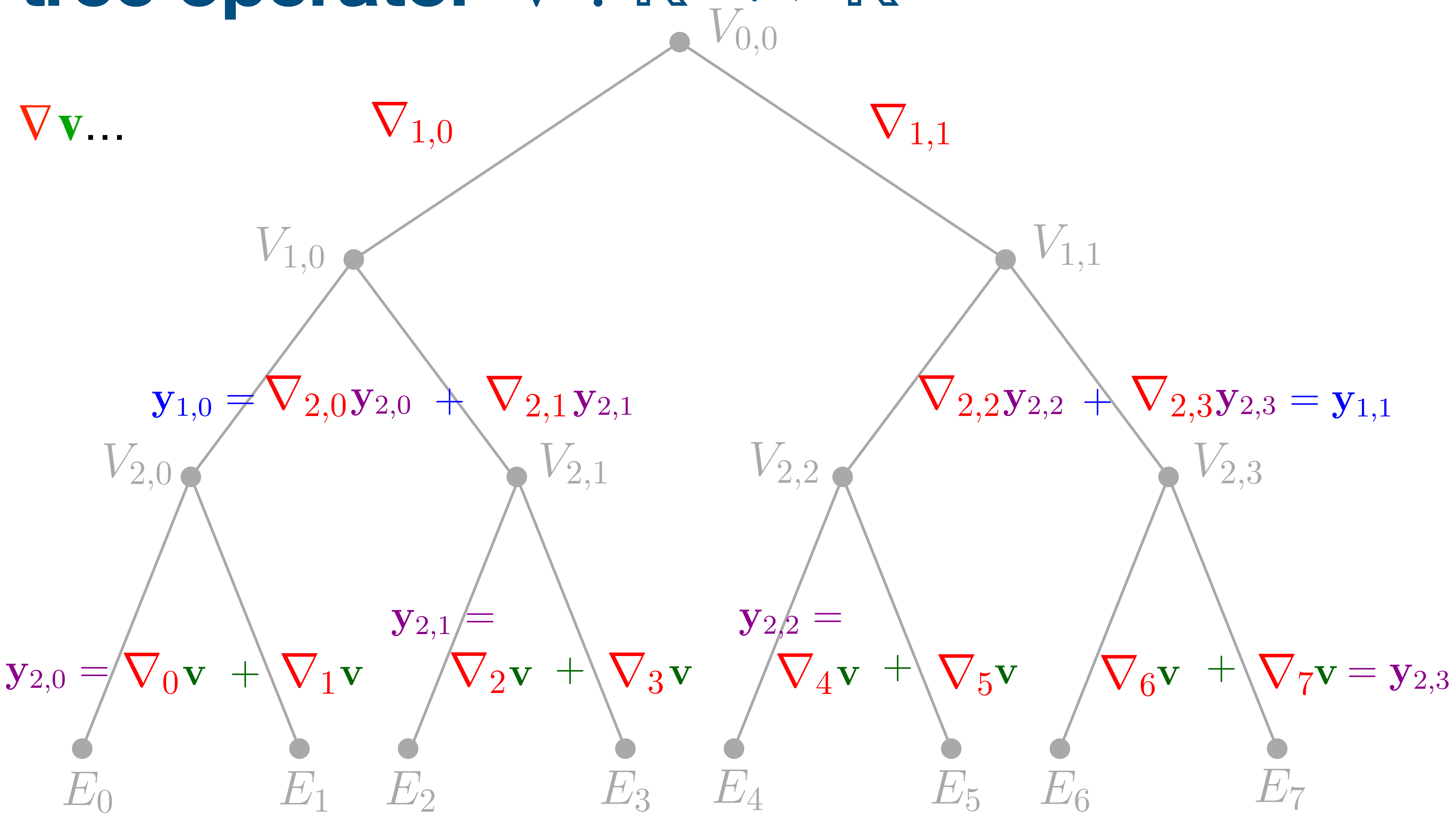
Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$

To compute $\nabla \mathbf{v}$...



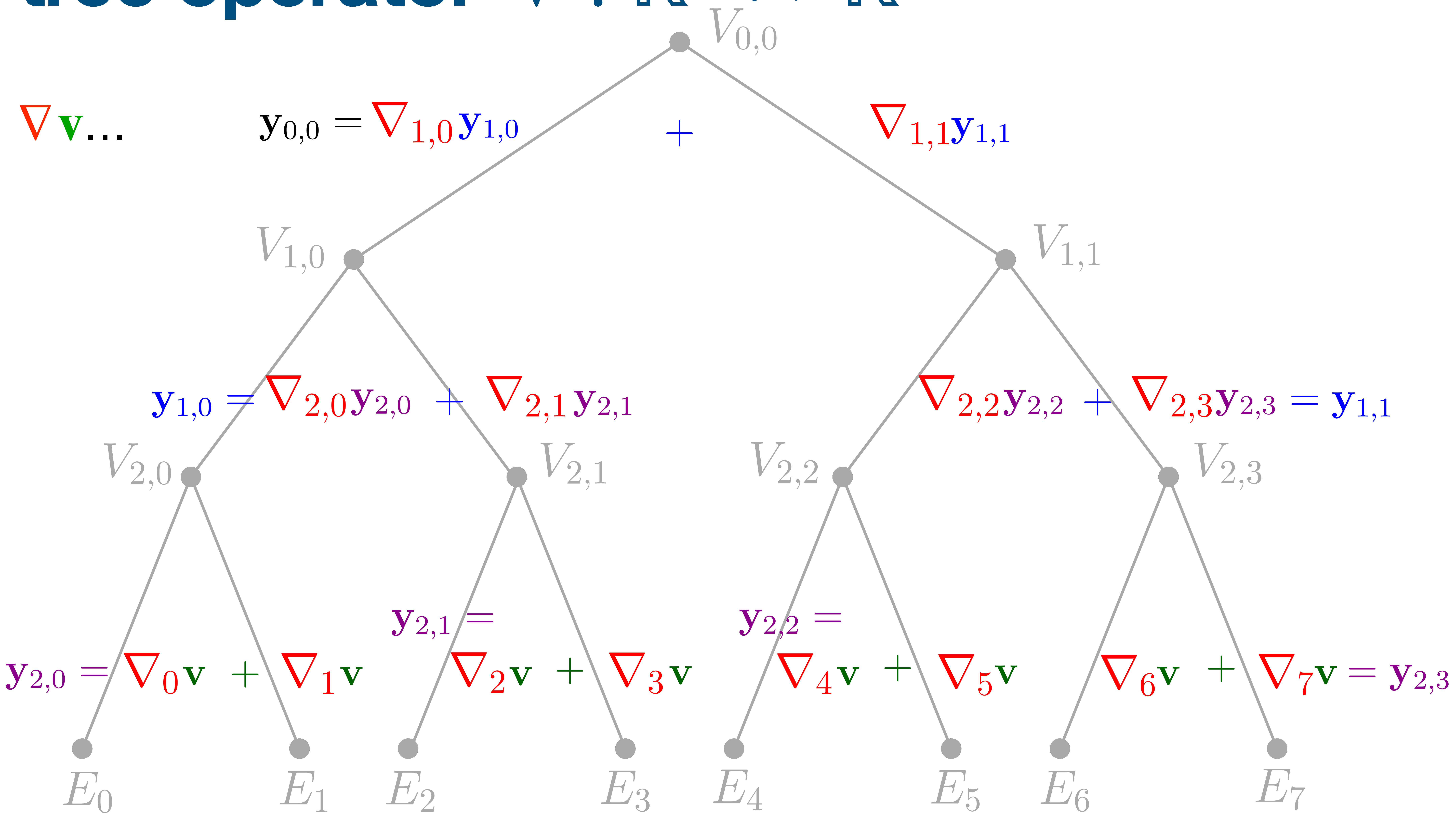
Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$

To compute $\nabla \mathbf{v}$...



Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$

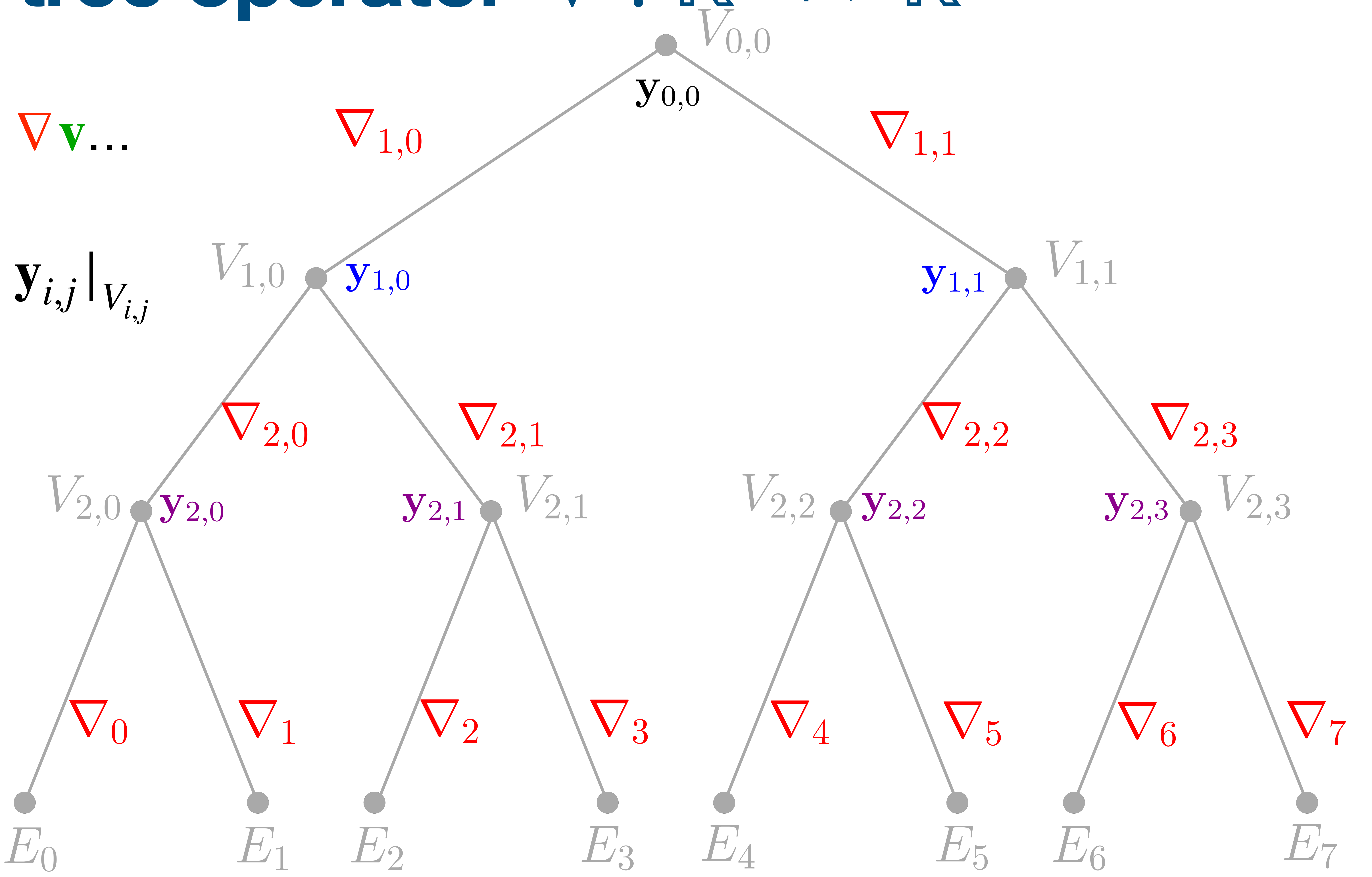
To compute $\nabla \mathbf{v}$...



Inverse tree operator $\nabla : \mathbb{R}^E \mapsto \mathbb{R}^V$

To compute $\nabla \mathbf{v}$...

$$(\nabla \mathbf{v})|_{V_{i,j}} = \mathbf{y}_{i,j}|_{V_{i,j}}$$



Fun lemma:

Δ is a tree operator if and only if $\Delta^\top$ is an inverse tree operator.

Proof:

Take the transpose of all edge operators.

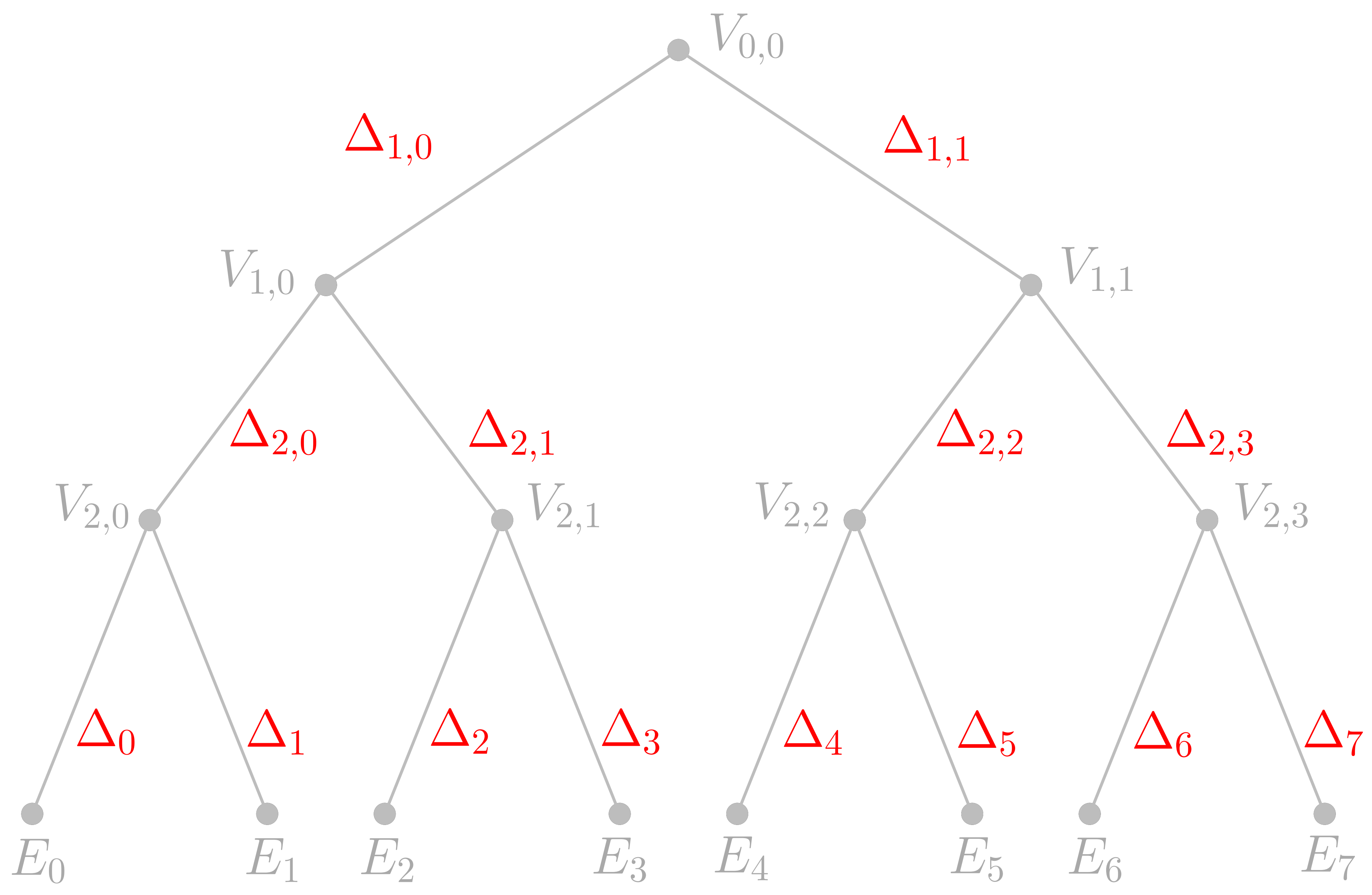
Complexity of Δ (and ∇)

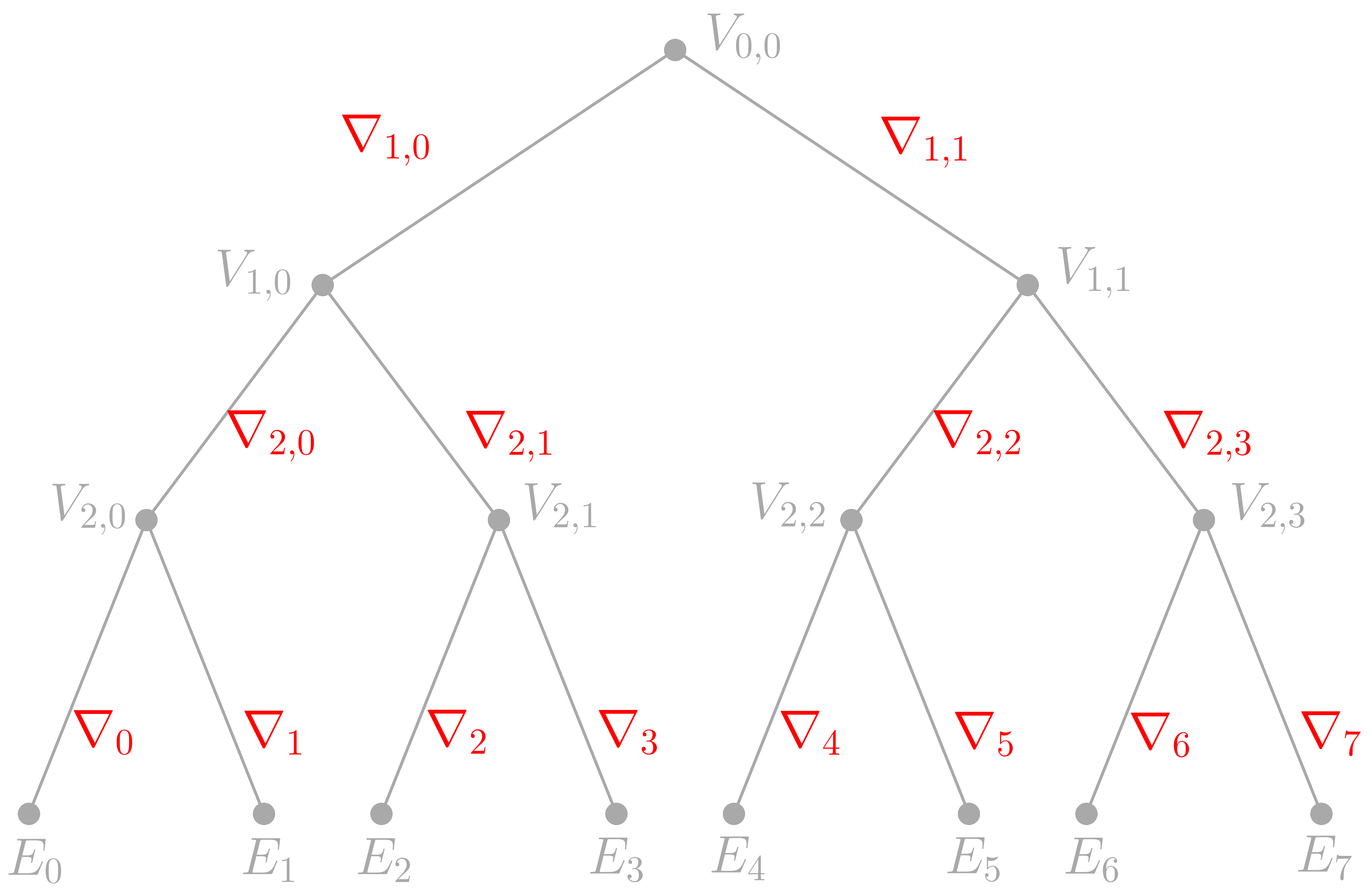
Say Δ has query complexity Q if the max time to apply k edge operators to k arbitrary vectors is at most $Q(k)$.

Recall Δ is a function of $\boldsymbol{w}$.

Say Δ has update complexity U if, when $\boldsymbol{w}$ changes in k coordinates, Δ can be updated in at most $U(k)$ time.

Easier to apply inverse tree operator





Implicit representation and heavy hitter detection both based on the tree structure

For a nice reference on this line of work, keep an eye out for my thesis

Thank you