Lecture 7:
DAGs & Dynamic Programming

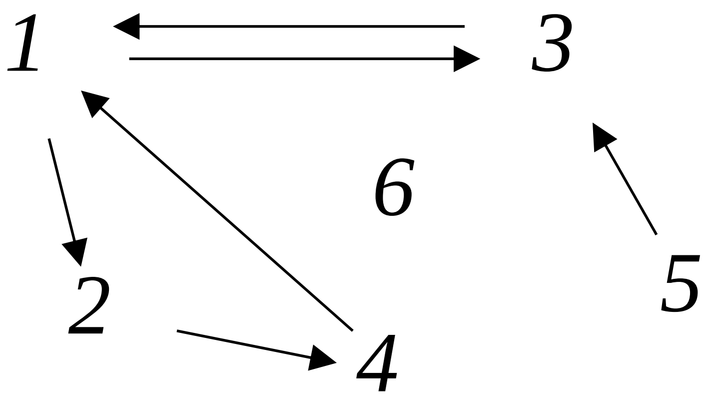
- Directed acyclic graphs
- Dynamic programming (‘The Fundamental Algorithm of Computational Biology’)
  - highest weight paths in weighted DAGs

Directed Graphs

- A *directed graph* is a pair (V, E) where
  - V is a finite set of *vertices*, or *nodes*.
  - E is a set of ordered pairs (called *edges*) of vertices in V.
- An edge (v_i, v_j) is said to *leave* v_i and to *enter* v_j.
  - (v_i and v_j are vertices)
- *in-degree* of a vertex = # edges entering it;
- *out-degree* = # edges leaving it.

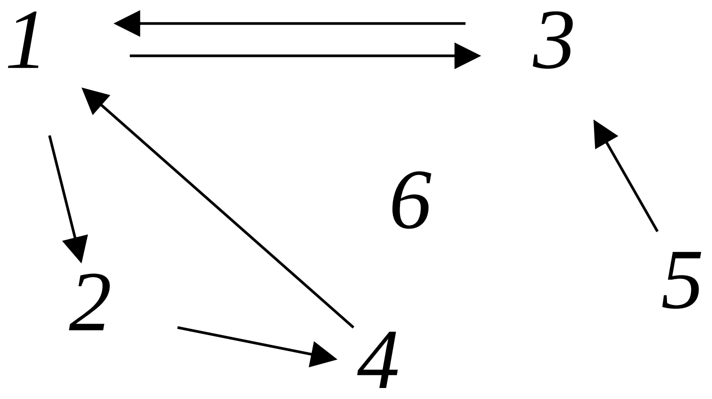
Example:

- $V = \{1,2,3,4,5,6\}$,
- $E = \{(1,2), (1,3), (2,4), (4,1), (5,3), (3,1)\}$
- Vertex 3 has in-degree 2 and out-degree 1.



Paths and Cycles

- A *path* of *length k* in *G from u* to *u'* (vertices) is
  - a sequence *P* of vertices ($v_0, v_1, \dots, v_k$) such that
    - $v_0 = u$,
    - $v_k = u'$, and
    - (v_{i-1}, v_i) is an edge for $i = 1,2, \dots, k$.
- A path can have length 0.
- We write $|P| = k$.
- A *cycle* is a path of length ≥ 1 from a vertex to itself.
- In example at right,
  - (*1*,*2*,*4*) is a path,
  - (*1*,*3*,*5*) is not, and
  - (*1*,*2*,*4*,*1*) and (*1*,*3*,*1*) are cycles.



- Can join
  - any path (u, ... , v) from u to v, to
  - any path (v, ... , w) from v to w

to get a path (u, ... , v, ... , w) from u to w.

DAGs

- A *directed acyclic graph* (DAG) is a directed graph with no cycles.
- In a DAG, for distinct nodes v_i and v_j, we say
  - v_i is a *parent* of v_j, and v_j is a *child* of v_i, if
    - there is an edge (v_i, v_j)
  - v_i is an *ancestor* of v_j, and v_j is a *descendant* of v_i, if
    - there is a path from v_i to v_j
- In a DAG the length of a path cannot exceed $|V| - 1$,
  - (where $|V|$ = total # vertices in graph)

because

  - in a path of length $\geq |V|$,
    - at least one vertex v would have to appear twice in the path;
  - but then there would be a path from v to v, i.e. a cycle.

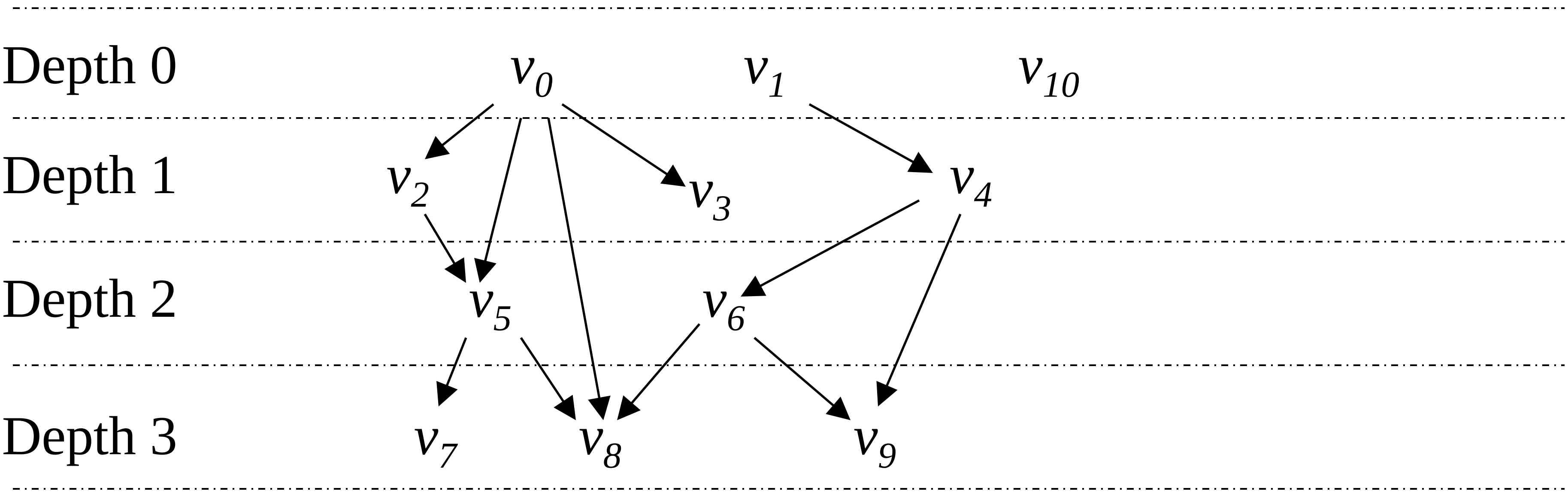
Structure of DAGs

- Define the *depth* of a node v in V as:
  - the length of the longest path ending at v;

  by above, the depth is well-defined and $\leq |V| - 1$.
- *Every descendant w of a node v has higher depth than v*: If
  - $(u, \ldots, v)$ is path of length $n = \text{depth}(v)$ ending at v, and
  - $(v, \ldots, w)$ is path from v to w,

  then $(u, \ldots, v, \ldots, w)$ is a path of length $> n$ ending at w, so $\text{depth}(w) > n$.

- *The nodes on any path are of increasing depth*.
- *Every node v of positive depth has a parent of depth exactly one less*:
  - Let (u, ... , v', v) be path of length n = depth(v) ending at v.
  - Then v' is a parent of v.
  - Since (u, ... , v') has length n – 1, depth(v') $\geq n - 1$.
  - Since also depth(v') $< n$ (because v is a descendant of v'), depth(v') is exactly n – 1.

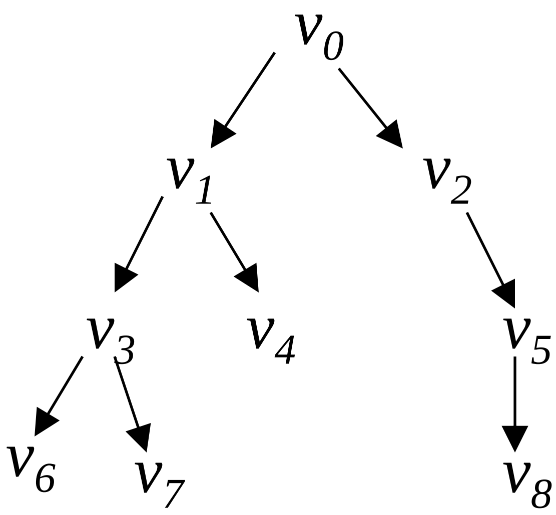

Depth 0
v_0
v_1
v_{10}
Depth 1
v_2
v_3
v_4
Depth 2
v_5
v_6
Depth 3
v_7
v_8
v_9

⋮

Important special cases:

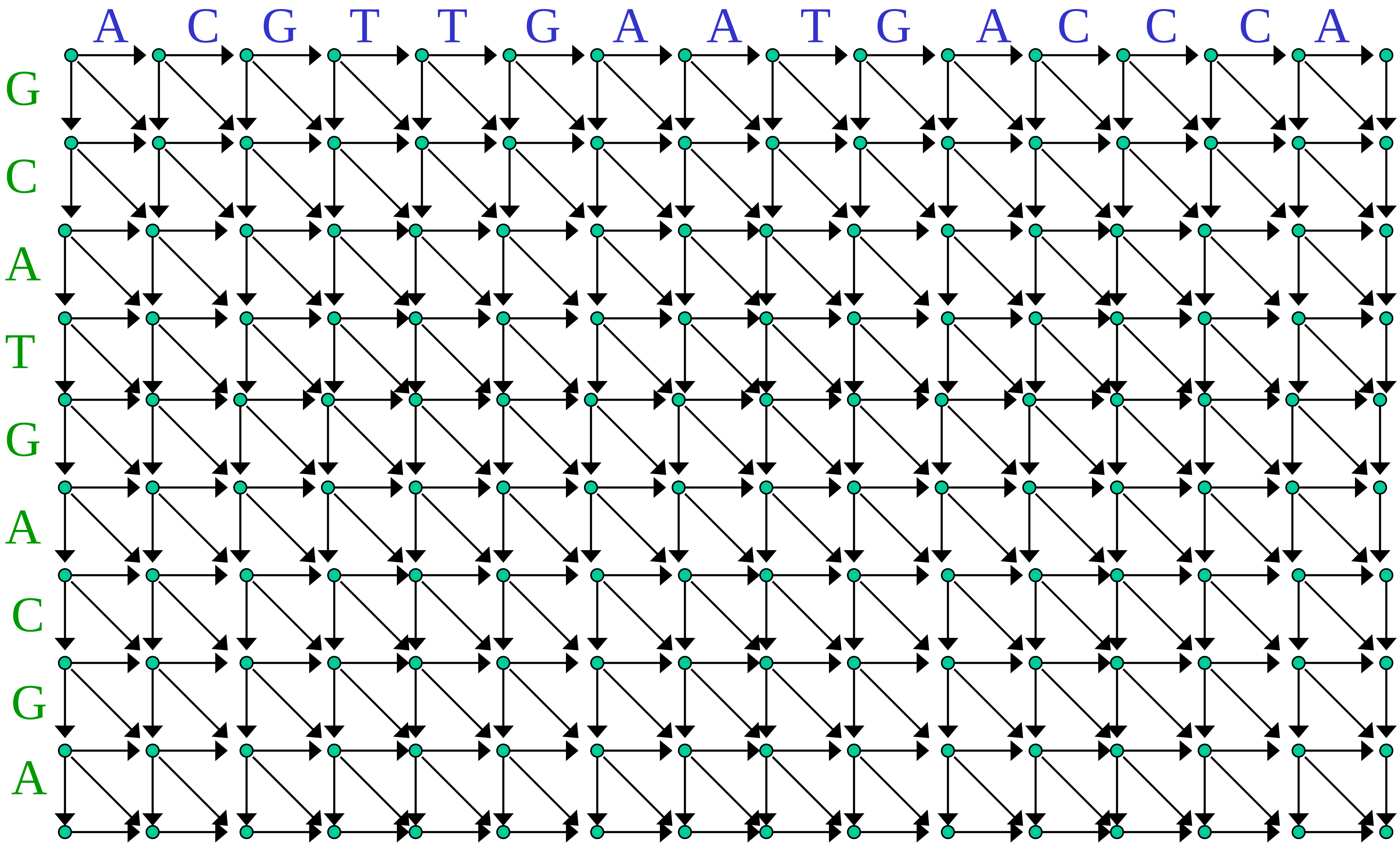
- A (*rooted*) *tree* is a DAG which
  - has unique depth 0 node (the *root*), *and*
  - every other node has in-degree 1
    - (i.e. has a unique parent, of depth one less than that of the node).
- A *binary tree* is a tree in which
  - every node has out-degree at most 2.
- A *linked list* is a tree in which
  - every node has out-degree at most 1
  - or equivalently, a DAG in which ∃ at most one node of each depth

binary tree



linked list

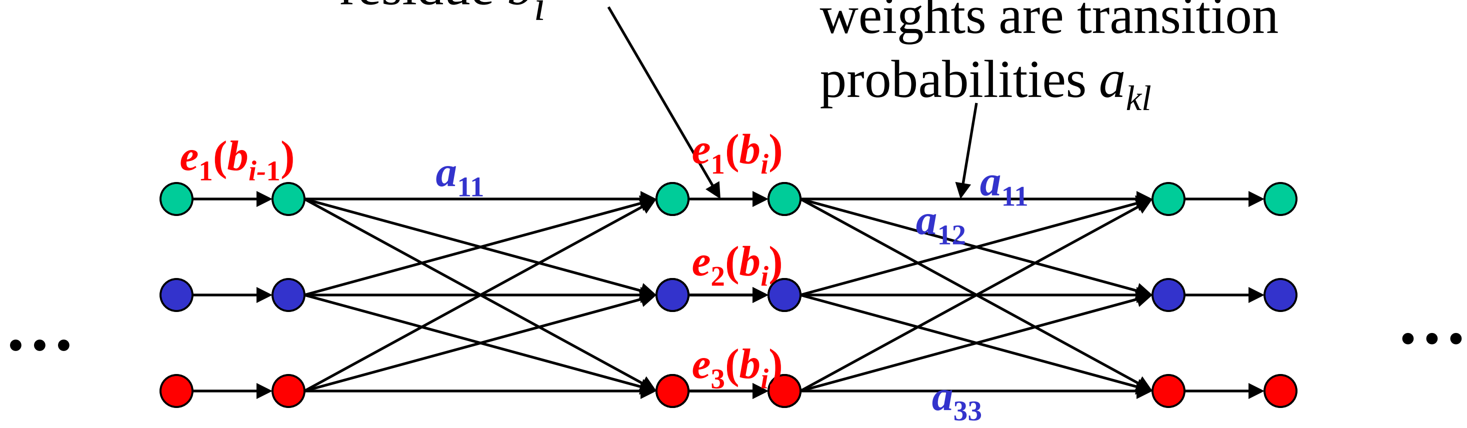
The *Edit Graph* for a Pair of Sequences



WDAG for 3-state HMM, length *n* sequence

weights are emission probabilities $e_k(b_i)$ for i^{th} residue b_i

weights are transition probabilities a_{kl}



b_{i-1}
position i-1

b_i
position i

b_{i+1}
position i+1

Remarks on Depth Structure

- For *dynamic programming* algorithm
  - we need an order $v_1, v_2, \ldots, v_n$ for the vertices
    - (not a path!)

    in which parents appear before children.
  - From the above, *depth order*
    - (in which depth 0 nodes are listed first, then depth 1 nodes, etc.)

    is such an order.
  - In general there are many other such orders.
- We haven't given constructive procedure for finding the depths of all vertices.
  - For an arbitrary DAG, can be done in $O(|V| + |E|)$ time;
  - we omit algorithm, since for DAGs related to sequence analysis, the depth structure is obvious.

Weighted Directed Graphs

- A *weighted directed graph* is
  - a directed graph (*V*, *E*) together with
  - a function *w* from *E* to the real numbers,
    - i.e. with a numerical *weight* *w*(*e*) (which may be positive, negative, or 0) associated to each edge *e*.

A weighted DAG is called a WDAG.

- In our applications, the weights usually come from a probability model:
  - probabilities
  - log(probabilities)
  - LLRs

Path Weights

- The (*sum*) *weight of a path* is defined to be the sum of the weights on the edges of the path.
- Similarly, the *product weight of a path* is the product of the edge weights
  - usually only consider this when all weights are non-negative.
- weight of a path P is written $w(P)$
- For a path of length 0 (i.e. consisting of a single vertex):
  - the sum weight is 0
  - the product weight is 1

Highest Weight Paths on WDAGs

- *Problem*: find a path with the highest possible weight.
- *Solution*:
  - "Brute force" approach
    - i.e. simply enumerating all possible paths and comparing their weights

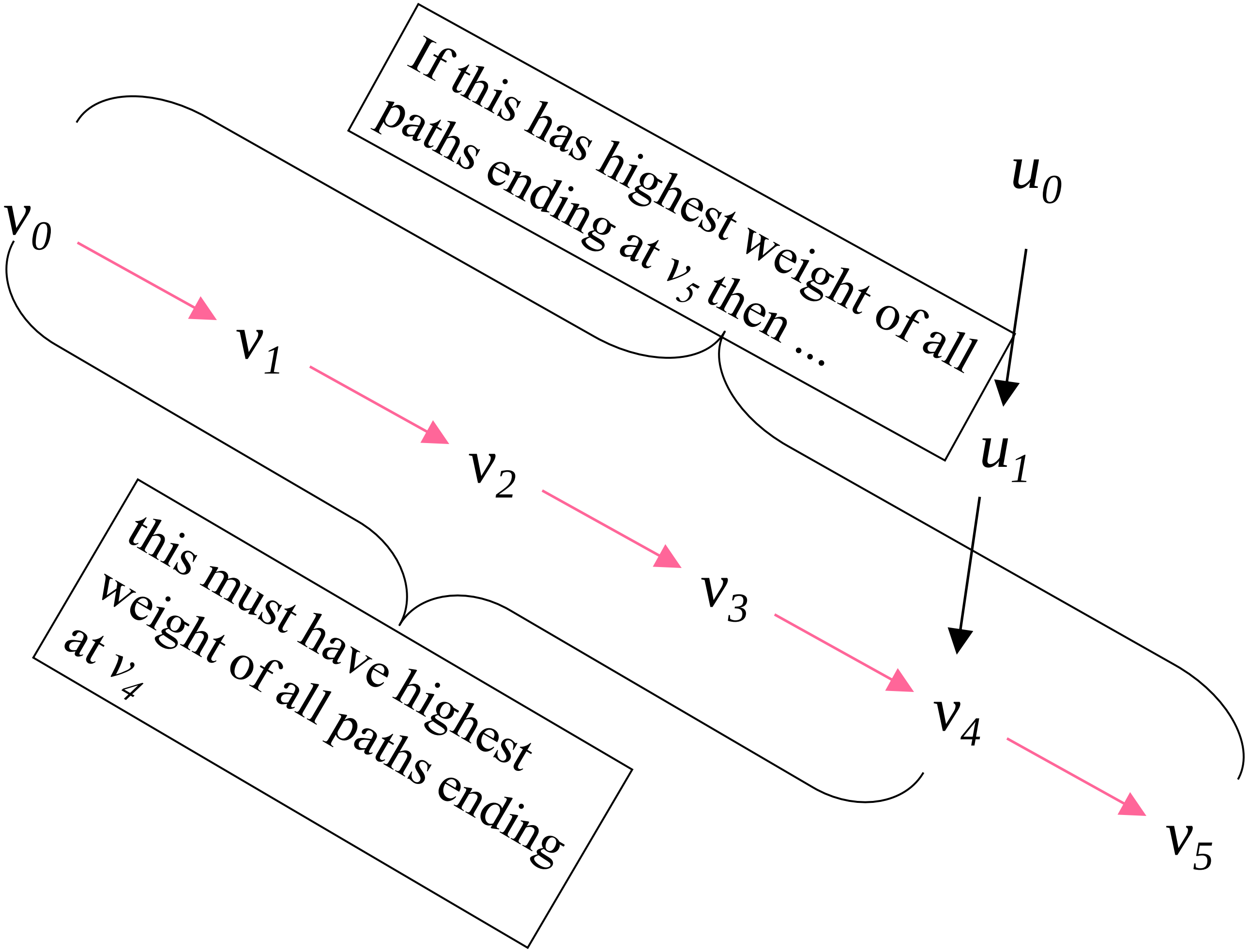
    is usually impractical (too many paths!)
  - Instead, use the method of *dynamic programming*
    - Richard Bellman (~1950)
    - Reduction to nested subproblems

- Let $P_n = (v_0, v_1, \ldots, v_n)$ be a path of highest weight.
- Then for each $k < n$, the sub-path $P_k = (v_0, v_1, \ldots, v_k)$ must have highest weight of all paths ending at v_k, because
  - *if* $Q = (u_0, u_1, \ldots, v_k)$ were another path ending at v_k and having higher weight than P_k,
  - *then* the path $(Q, v_{k+1}, \ldots, v_n)$ would have weight

$$w((Q, v_{k+1}, \ldots, v_n)) = w(Q) + w((v_k, \ldots, v_n))$$
$$> w(P_k) + w((v_k, \ldots, v_n)) = w(P_n),$$

contradicting assumption that P_n has highest weight.

Subpaths of a highest-weight path can't be improved:



If this has highest weight of all paths ending at v_5 then …

this must have highest weight of all paths ending at v_4

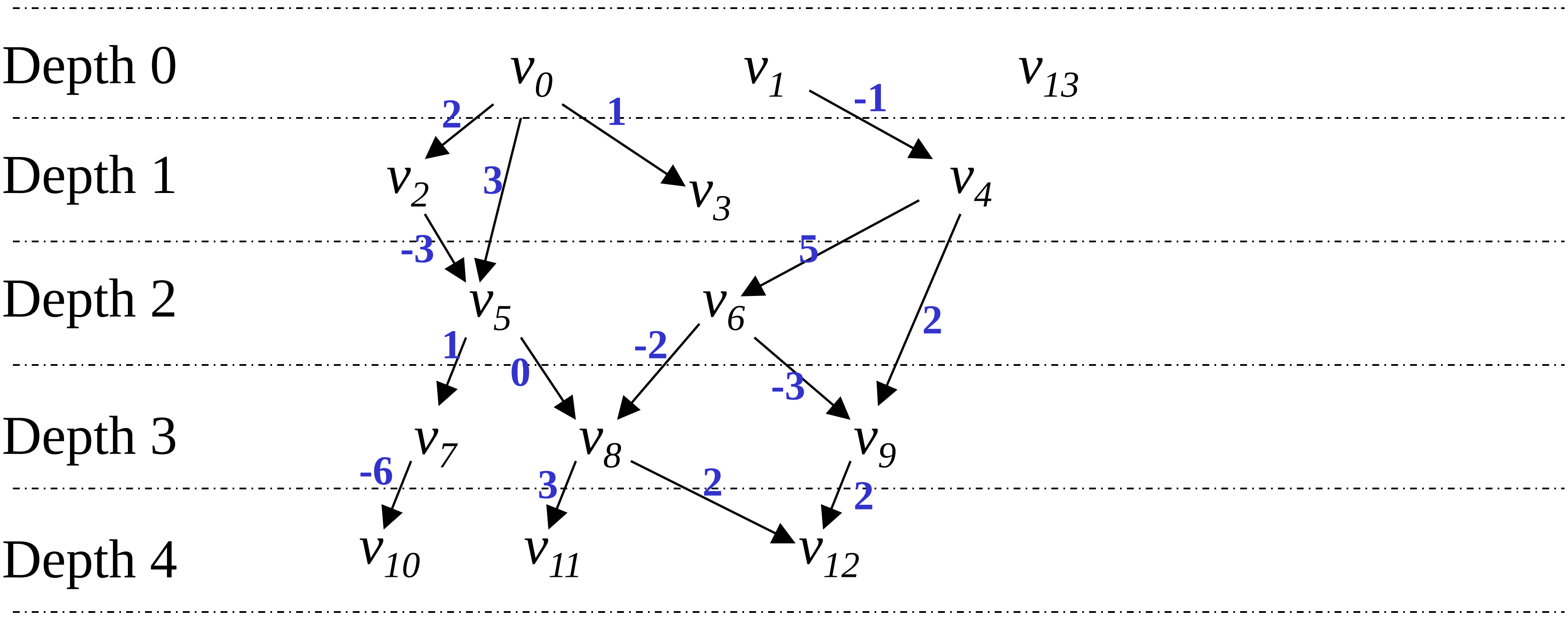
- So generalize the problem as follows:
- find, for *each* vertex v, the highest weight of all paths ending at v – call this $w(v)$
- Can find $w(v)$ in single pass through V, as follows:
  - process the v in depth order (*or any order in which parents precede children*)
  - if v has no parents, $w(v) = 0$ (the only path ending at v is (v)).
  - for any other v, except for the path (v) (which has weight 0), any path ending at v is of form $(v_0, v_1, . . ., v_k , u , v)$. Then
  - u is a parent of v, so $w(u)$ has already been computed, and

    $$w((v_0, v_1, . . . , v_k , u , v)) \leq w(u) + w((u,v))$$

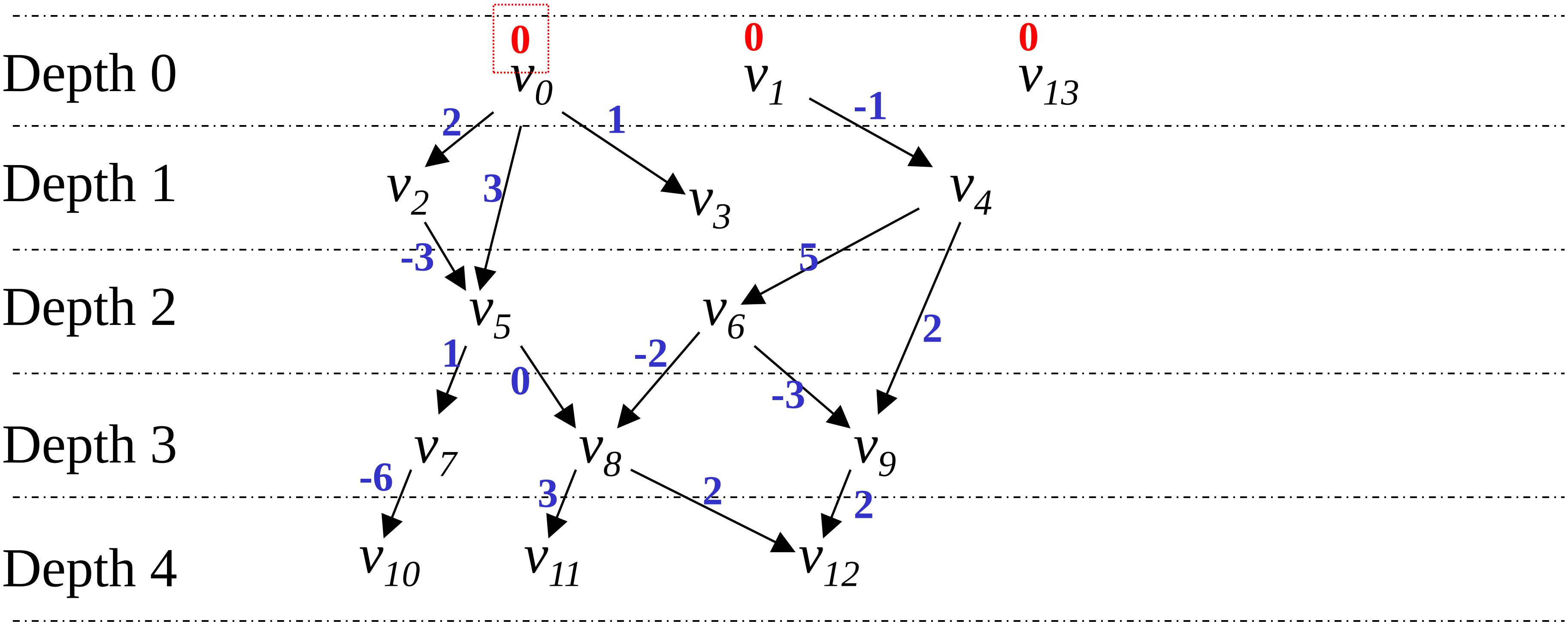
    with equality for an appropriate choice of v_i.
  - Therefore we may compute $w(v)$ as

$$w(v) = \max(0, \max_{u \in parents(v)} (w(u) + w((u,v))))$$

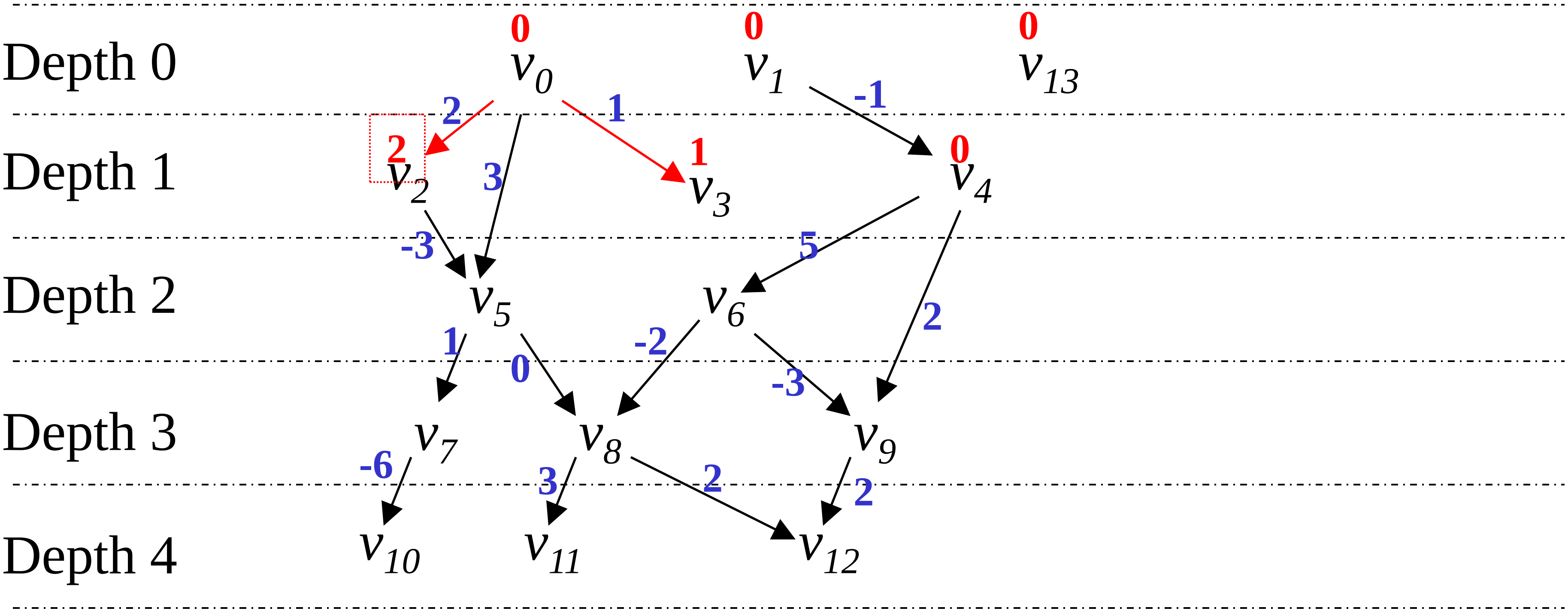
Example


Depth 0
Depth 1
Depth 2
Depth 3
Depth 4
v_0
v_1
v_{13}
2
1
-1
v_2
3
v_3
v_4
-3
5
v_5
v_6
2
1
0
-2
-3
v_7
v_8
v_9
-6
3
2
2
v_{10}
v_{11}
v_{12}

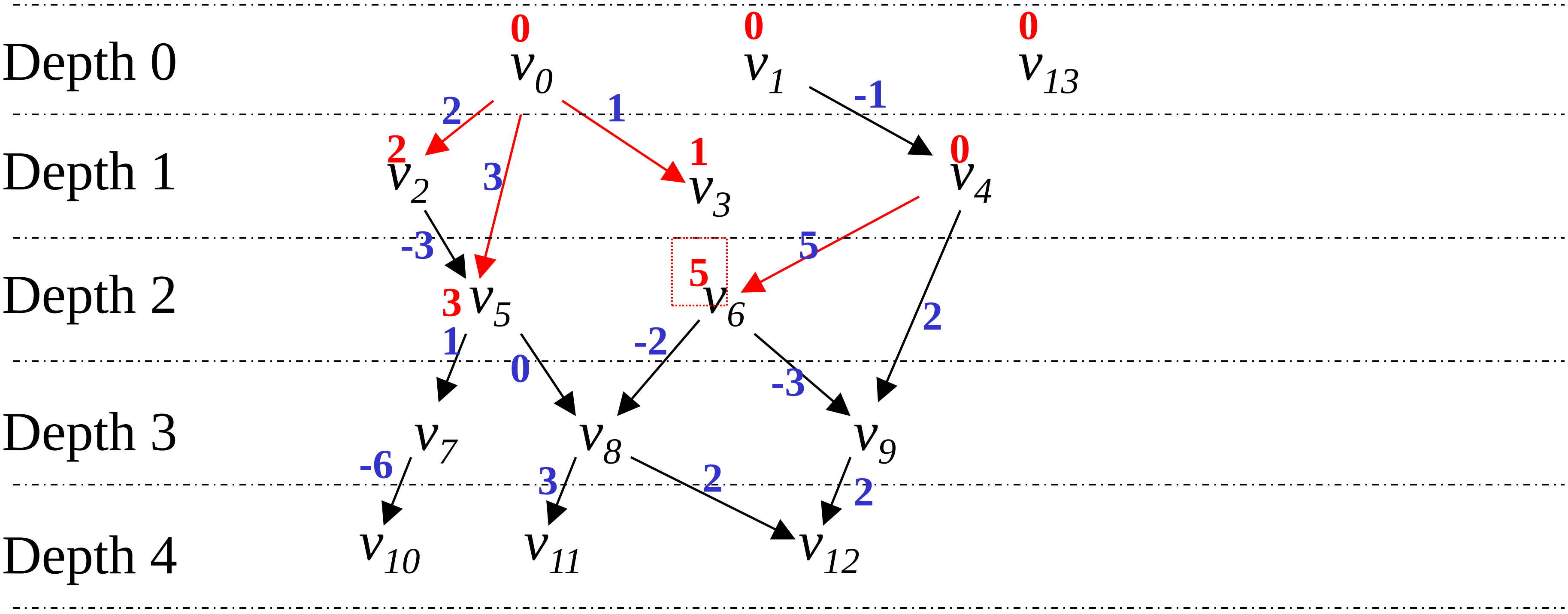
$w(v)$ – depth 0 nodes



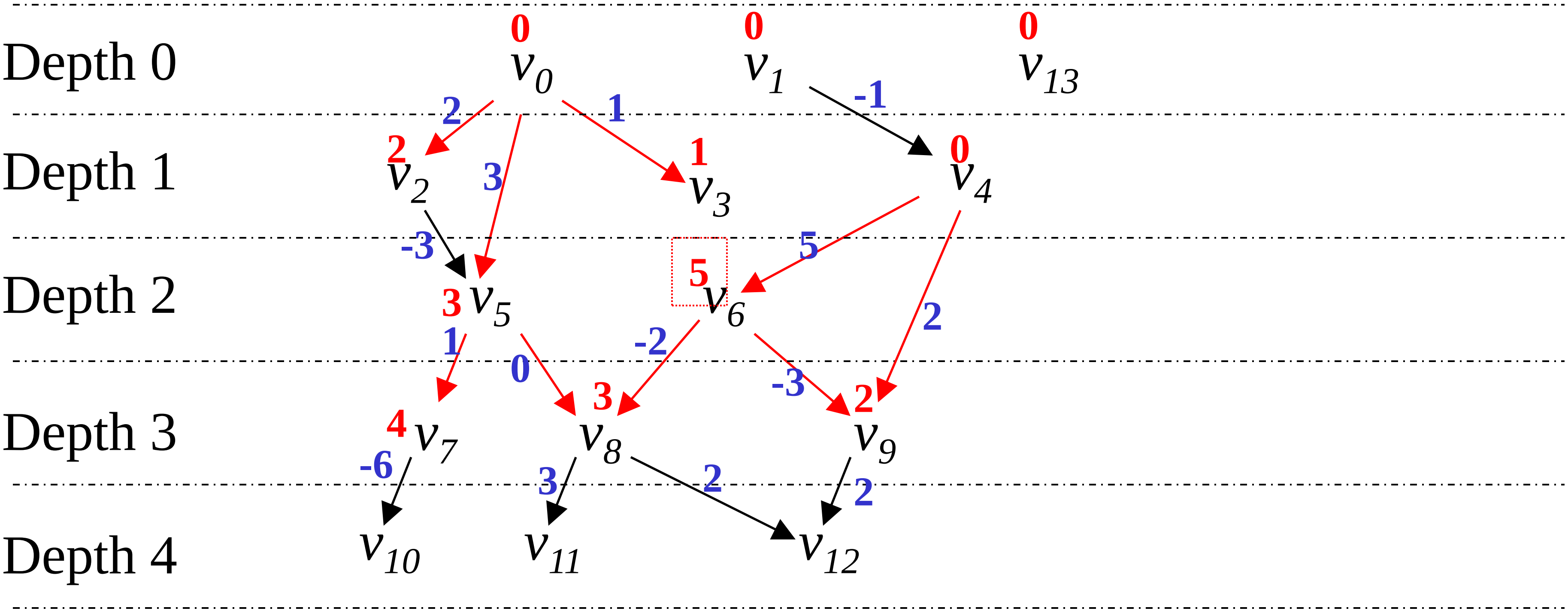
w(*v*) – depth 1 nodes



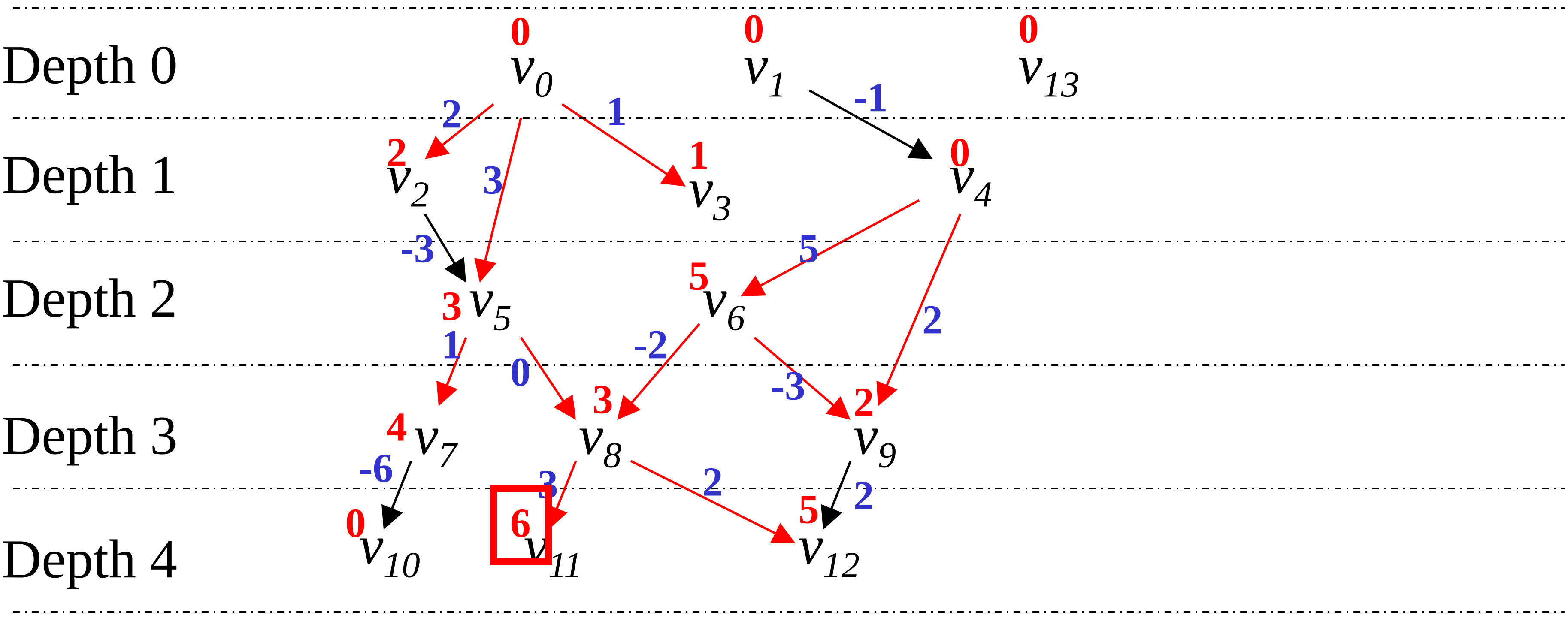
$w(v)$ – depth 2 nodes



w(*v*) – depth 3 nodes



w(*v*) – depth 4 nodes



- To reconstruct best path, need "traceback" pointer to immediate predecessor of *v* in best path:

$$T(v) = \begin{cases} v & w(v) = 0 \\ \underset{u \in \text{parents}(v)}{\arg\max} \; (w(u) + w((u,v))) & w(v) \neq 0 \end{cases}$$

  - in preceding graph, *T*(*v*) is the *parent* on *red edge* coming into *v*
    - if more than one such edge, pick one at random;
    - if no such edge, *T*(*v*) = *v*
- Sometimes useful to record *beginning* of best path:

$$B(v) = \begin{cases} v & w(v) = 0 \\ B(T(v)) & w(v) \neq 0 \end{cases}$$

- Then highest weight of any path in graph is

$$\max_{v \in V} (w(v))$$

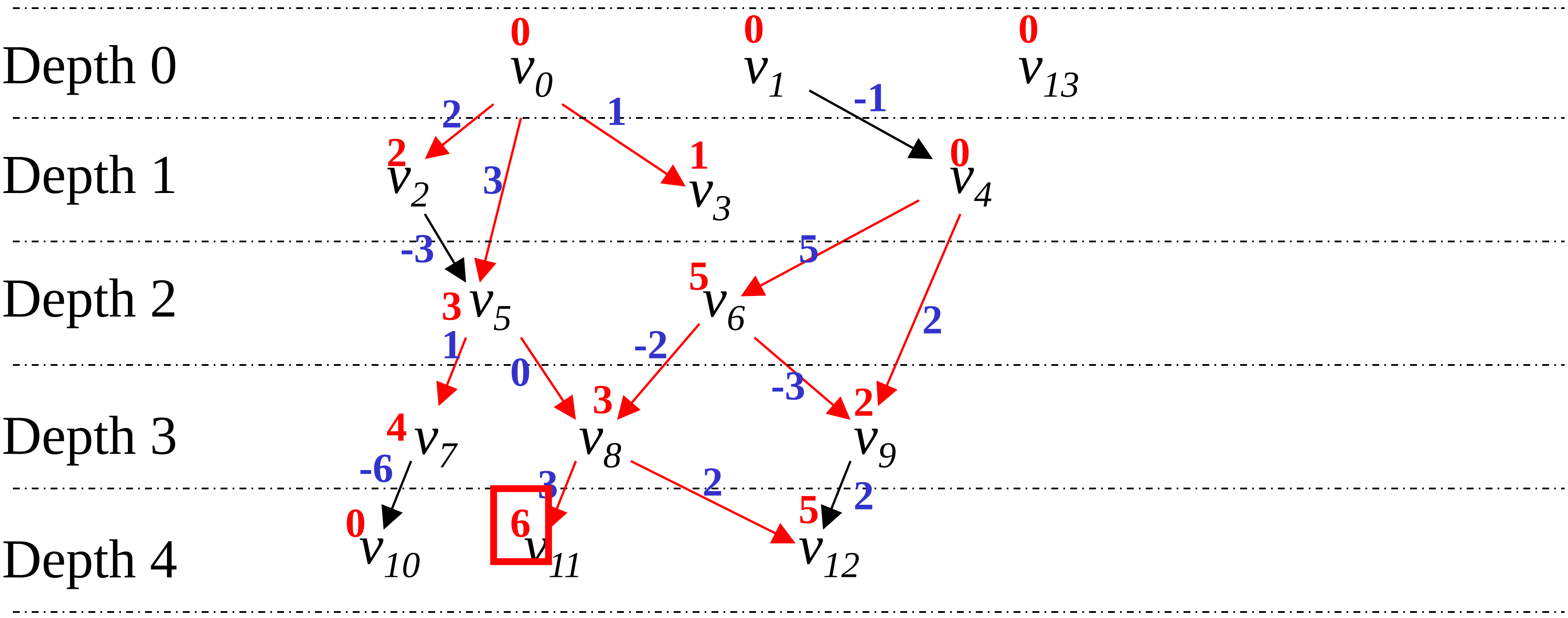
  - updated as each node is visited
    - indicated by ⬚ in preceding graph –

  and so doesn’t require additional pass through vertices

- if $u = \operatorname{argmax}_{v \in V} (w(v))$, can reconstruct highest weight path by tracing back from u, using T:
  - path ends at u;
  - immediate predecessor of u is $T(u)$;
  - predecessor of $T(u)$ is $T(T(u))$; etc.
  - stop when $T(v) = v$.

- In preceding example, highest weight is 6 and $u = v_{11}$

Dynamic programming on WDAGs



Complexity of Dynamic Programming

- Time to find a best path is $O(|E|+|V|)$:
  - in initial pass, visit each node, and each edge into that node: $O(|E|+|V|)$
  - in traceback, visit subset of nodes, and unique edge from each node: $O(|V|)$

(Complexity to find *all* highest weight paths can be higher)

For very large graphs, even $O(|E|+|V|)$ may be unacceptable!

- Space requirements:
  - If only want *weight* of best path, and beginning and end, then
    - don’t need $T(v)$, and
    - only need retain $w(v)$ and $B(v)$ until have processed all children of v (or when best path found so far ends at v).

    Space depends on graph structure, but usually << $O(|V|)$.
  - If want path itself, must store $T(v)\ \forall\ v$
    - space = $O(|V|)$
    - ∃ algorithms (for some graphs) to reduce this, but may take more time.

Imposing *constraints* on allowed paths

- Above algorithm can easily be modified to find highest weight path that
- *starts* in particular subset V' of vertices
  - don’t consider paths that start outside V' :
    - i.e. when computing $w(v)$, don’t consider trivial path unless $v \in V'$
- *ends* in particular subset V''
  - only scan for the maximum $w(v)$ over V''
- *goes through* a particular vertex v
  - use *forward/backward* algorithm (future lecture)
- or a *combination* of these!

Same dynamic programming approach can be used to find:

- Highest product weight path (if weights are ≥ 0)
- Sum of product weights of all paths ending at particular vertex
  - *sum* over all edges coming into *v*, instead of *maximizing*
- Useful for HMM and phylogeny probability calculations!

Finding *multiple* high-scoring paths

- If high-weight paths are important, we'll want more than one!
  - – But *not* slight perturbations of highest-weight path
- 'Brute force' algorithm:
  - – Find highest-weight path
  - – 'Mask it' (remove its edges from graph)
  - – Repeat above two steps until scores uninteresting
  - – can be $O(N^2)$, but often acceptable
- $O(N)$ algorithms for WLLs
  - – Ruzzo-Tompa
  - – HMMs (Viterbi algorithm)