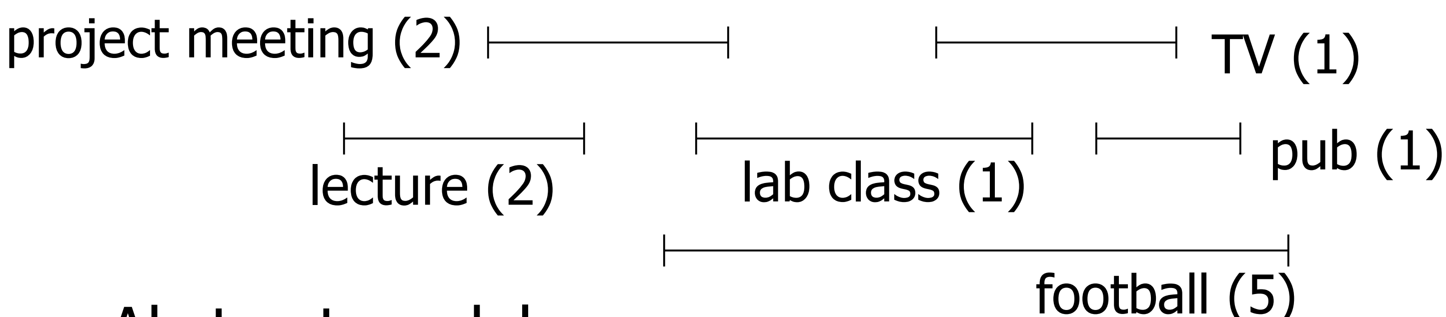
Chapter 9
Dynamic Programming

References:
[KT 6.1-6.2, 6.6, 6.8]
[CLRS 15, 24.1-24.2, 25.1-25.2]
[DPV 4.6, 6.3, 6.6]
[SSS 8]

Activity Selection Revisited

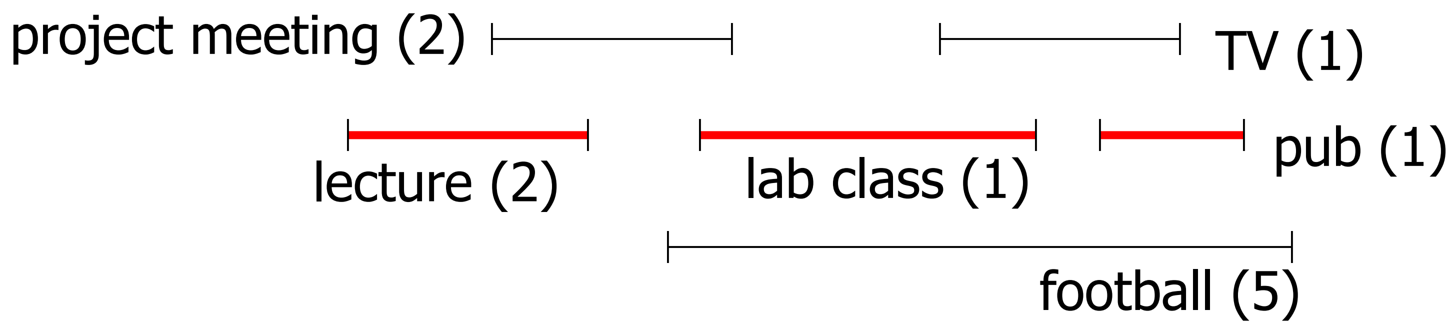
- Recall the activity selection problem
  - Want to maximize number of activities
  - What if each activity has a different "value"?



- Abstract model
  - Given: a set of intervals, each with starting time, finishing time and value
  - Goal: find a set of non-overlapping intervals with *maximum total value*

Greedy Does Not Work Anymore

- This problem generalizes the original (unweighted) one
  - When all value = 1, reduces to original problem
- Previous greedy approaches do not work
  - E.g. earliest finishing first:



- In fact, there seems no way of "locally" adding an interval "safely"

Let's Turn to a Different Problem…

- Given an array A with n elements, choose a subset of elements so that
  - Any two chosen elements are separated by 2 cells or more
  - The sum is maximised
  - Assume empty subset has sum 0
- Example: A =

2	4	1	3	1	6	5	4	2

  - Maximum = A[2] + A[6] + A[9] = 4 + 6 + 2 = 12
- Greedy algorithms do not work
  - E.g. always choose the largest element, discard elements within 2 cells, and repeat
    - Counterexample: A =

6	1	7	6

An Important Observation

- Consider A[n]. We do not know whether A[n] should be part of the solution; but we know that
  - Either A[n] is part of the solution, or it is not.
- This sounds trivial enough, but it allows us to *reduce to a smaller subproblem:*
  - If A[n] is part of the solution, A[n–1] and A[n–2] cannot be; next step consider A[1..n–3]
  - If A[n] is not part of the solution, next step to consider A[1..n–1]
  - Which one is correct? Try both and see which solution is better!

						X	X	
2	4	1	3	1	6	5	4	2

or

								X
2	4	1	3	1	6	5	4	2

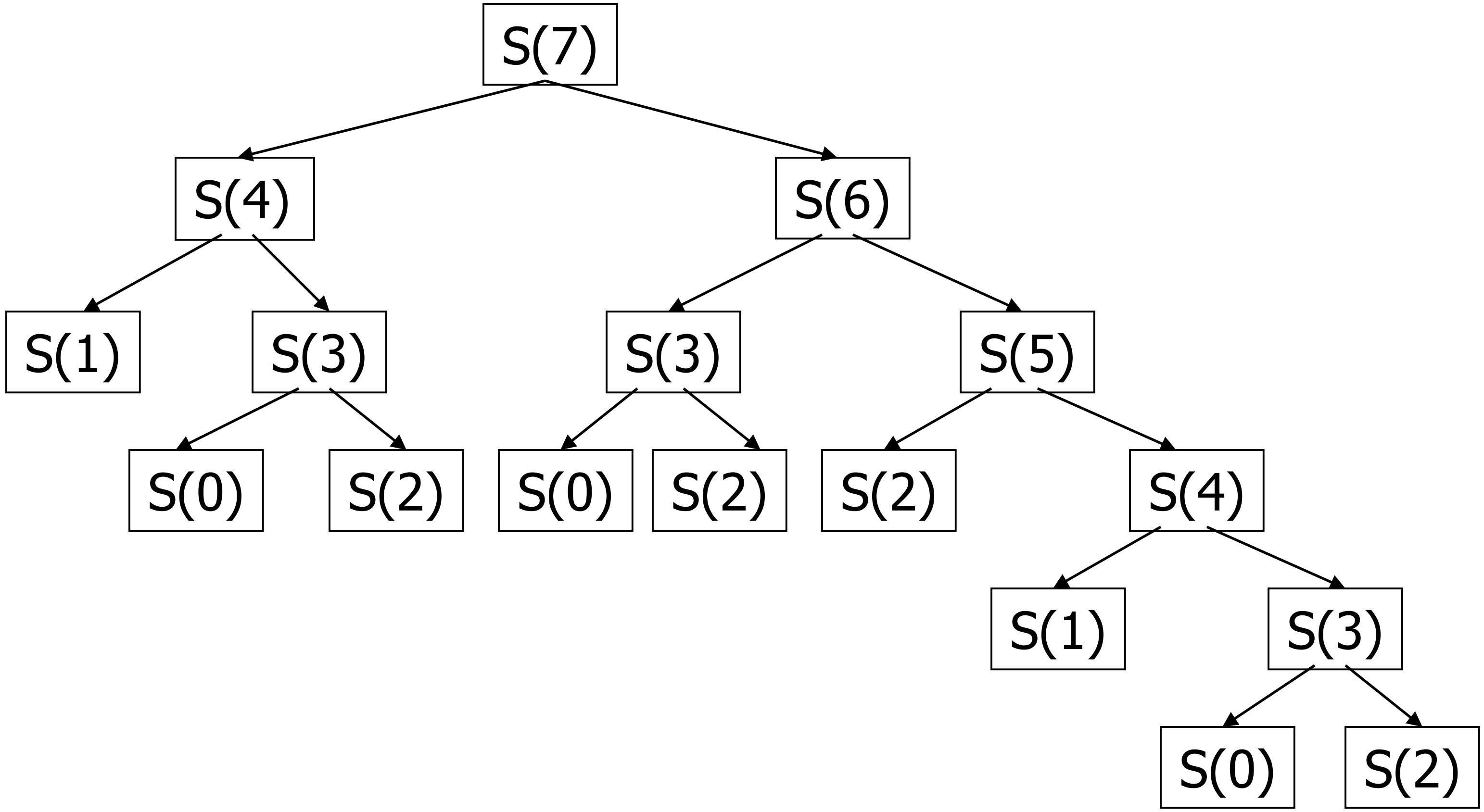
A Recursive Formulation

- First, define subproblems: let S(i) be the solution considering only the first i elements
  - Want to find S(n)
- Recursive formula:
  - S(i) = max(S(i – 3)+A[i], S(i – 1))
  - Base cases: S(0) = 0, S(1) = max(0, A[1]), S(2) = max(0, A[1], A[2])
- Recursive algorithm: (solution: call S(n))

```
S(i) {  // for first i elements
  if (i==0) return 0  // base cases
  if (i==1) return max(0, A[1])
  if (i==2) return max(0, A[1], A[2])
  else return max(S(i-3) + A[i], S(i-1))
}
```

This is not Efficient…

- This approach requires exponential time!
  - Exponentially many subproblems



Memorising Solution of Subproblems

- Observation: there is no point in recomputing S(i) so many times!
  - Use an array to store solutions of subproblems computed before
  - Only compute fresh if not previously computed
- Pseudocode:

```
initialize M[0] := 0, M[1] := max(0,A[1]),
M[2]:= max(0,A[1],A[2]), other M[]:= infinity

S(i) {
  if (M[i] == infinity) {
    M[i] := max(S(i-3) + A[i], S(i-1))
  }
  return M[i]
}
```

An Equivalent Iterative Algorithm

- Observe that recursion is only called on smaller values of i
- Hence, if we compute the array M[] in the correct order (increasing i), we can eliminate recursion completely!

```
S(n) {
  M[0] := 0, M[1] := max(0,A[1]),
  M[2] := max(0,A[1],A[2])   // base cases
  for i := 3 to n
    M[i] := max(M[i-3] + A[i], M[i-1])
  return M[n]
}
```

Time and Space Complexity

- Time complexity:
  - for loop executed O(n) times
  - O(1) time to compute one array entry
  - Therefore, *O(n) time*
- Space complexity: how much working memory is used
  - Array M[]: *O(n) space*

Finding the Actual Solution

- The function S() or the array M[] only gives the *value of the objective function* (i.e. the sum), not the *actual solution* (i.e. which elements?)
- Record the information about which case is chosen

```
S(n) {
  M[0] := 0, M[1] := … // same base cases
  Take[0] := false
  Take[1] := (A[1]>0) ? true : false
  Take[2] := (A[2]>A[1] && A[2]>0) ? true : false
  for i := 3 to n
    if (M[i-3] + A[i] > M[i-1])
      { M[i] := M[i-3] + A[i]; Take[i] := true }
    else
      { M[i] := M[i-1]; Take[i] := false }
}
```

Finding the Elements

- With this information, we can recover the solution

j	0	1	2	3	4	5	6	7	8	9
M[j]	0	2	4	4	5	5	10	10	10	12
Take[j]	F	T	T	F	T	F	T	F	F	T

- Optimal solution:
  - Sum = M[9] = 12
  - Take[9] = true, take A[9]=2, consider M[6]
  - Take[6] = true, take A[6]=6, consider M[3]
  - Take[3] = false, do not take A[3], consider M[2]
  - Take[2] = true, take A[2]=4

Traceback

- Pseudocode for this “tracing back” procedure:

```
i := n
while (i > 0) {
  if (Take[i] == true) {
    Include A[i] as part of solution
    i := i - 3
  }
  else {
    i := i - 1
  }
}
```

  - O(n) time

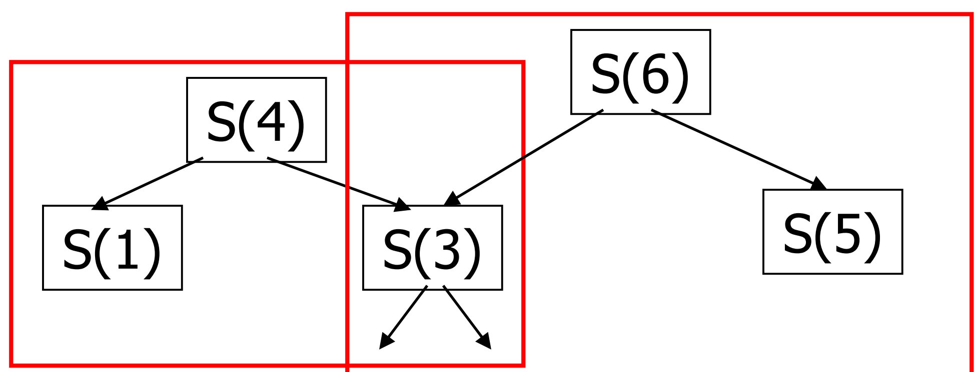
Principles of Dynamic Programming

- 1) *A recursive formulation*
- 2) *Optimal substructure*
  - Example: optimal solution for A[1..9] = {4, 6, 2}. Then {4, 6} is also optimal solution for A[1..6]

A =

2	4	1	3	1	6	5	4	2

- 3) *Overlapping subproblems*
  - Example: subproblems of S(4) and S(6) overlap
  - Use a table to memorize solutions of subproblems



Steps in Dynamic Programming

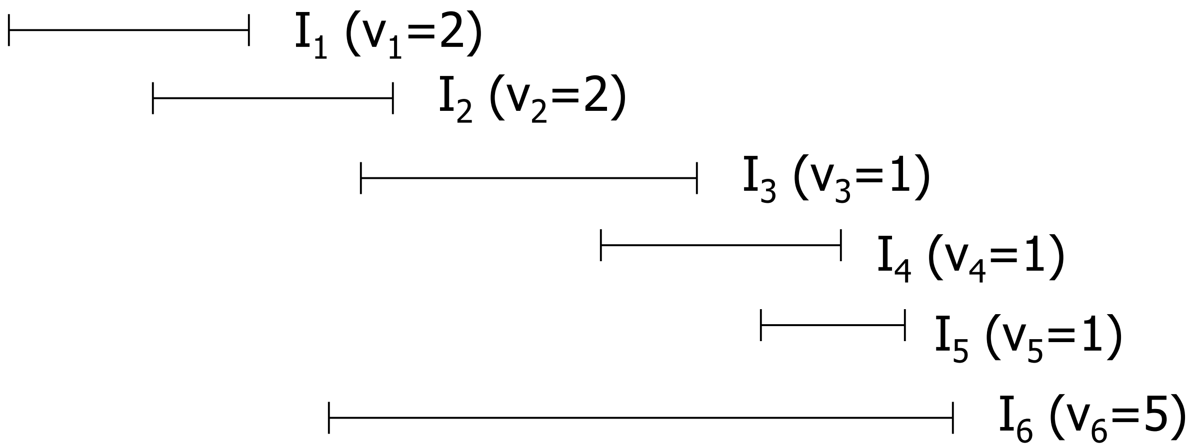
- Step 1: Define a recursive formulation that utilises the structure of the optimal solution
- Step 2: Compute the optimal cost using a table. Two approaches:
  - *Top-down:* use recursion to call smaller subproblems. Stored results used if possible
  - *Bottom-up:* iterate all subproblems starting from smaller ones
- Step 3: Construct the actual solution (traceback)

More on Top-down vs. Bottom-up

- Top-down approach:
  - Start with large subproblems
  - Recursively call for smaller subproblems
  - Subproblems only solved when required (so not all table entries always filled)
- Bottom-up approach:
  - Build solutions of subproblems systematically
  - When large subproblems encountered, solution to small subproblems already computed
  - No recursion overhead

Relation to Interval Selection

- The weighted interval selection problem is very similar:
  - Let $S(i)$ = optimal solution for $\{I_1, \ldots, I_i\}$
  - Either include I_i and discard all intervals overlapping it
  - Or discard I_i and recursively solve $S(i-1)$



  - $S(6) = \max(S(1) + v_6, S(5))$
  - $S(5) = \max(S(3) + v_5, S(4))$
  - …

Sequence Comparisons

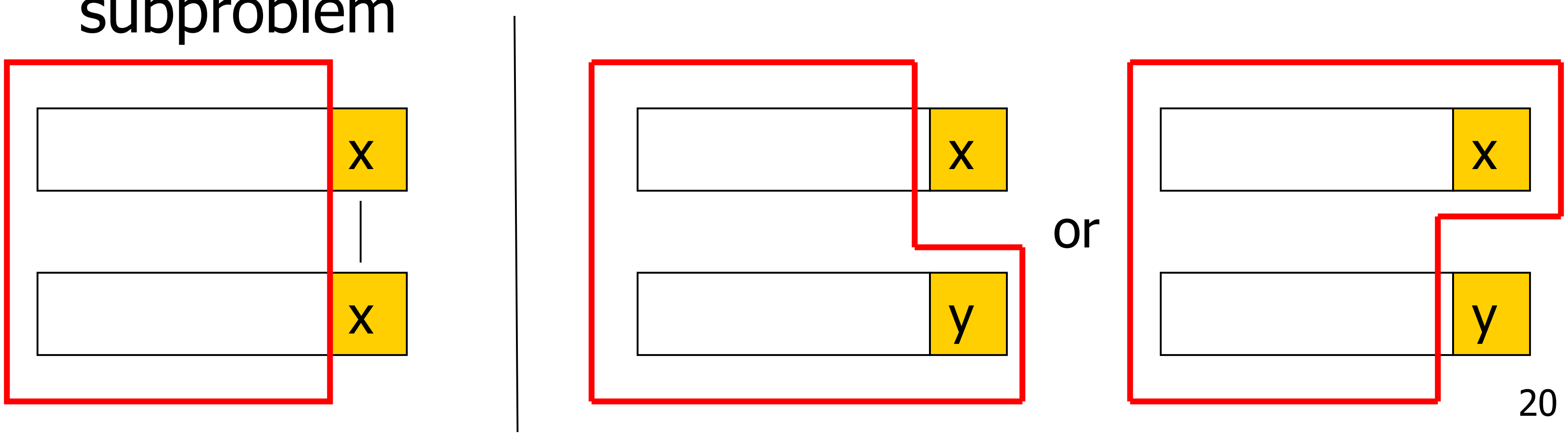
Longest Common Subsequence

- Given two sequences of letters
  - S1 = abcaabcabc, S2 = aabcabbabcc
- A *subsequence* is a subset of letters from a sequence in original order
  - E.g. accab (from S1, but not from S2)
- A *common subsequence* of S1 and S2 is a subsequence of both sequences
  - E.g. ababbc (from S1 and S2)
- A *longest common subsequence* is a common subsequence with maximum length

```
abcaabcabc
|\\\ |  ||\
aabcabbabcc
```

Step 1: Optimal Solution's Structure

- Common subsequences imply non-crossing matching
- Given two sequence A[1..m] and B[1..n]
- If A[m] = B[n] (last character match), then it is always safe to match them; reduce to a smaller subproblem
- If A[m] $\neq$ B[n] (doesn't match), then one of A[m] or B[n] (or both) is not part of LCS; reduce to a smaller subproblem



Step 2: Recursive Formulation

- Let LCS(i, j) = length of LCS of A[1..i] and B[1..j]
- So we have, for any i and j,

$$\mathrm{LCS}(i, j) = \begin{cases} \mathrm{LCS}(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max \begin{cases} \mathrm{LCS}(i-1, j) \\ \mathrm{LCS}(i, j-1) \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

- Base case: LCS(0, j) = 0, LCS(i, 0) = 0
- This gives us a recursive algorithm
- Use a table to memorize solutions of subproblems (top-down approach)

Step 3: Bottom-up Algorithm

- We can use a bottom-up algorithm instead
  - Observe each table entry (i, j) depends on at most 3 others: (i–1, j–1), (i, j–1), (i–1, j)
  - A standard double-for-loop ensures solutions to smaller subproblems already computed

```
LCS(A[1..m], B[1..n]) {
  Let L be (m+1) by (n+1) 2D array with all entries
  initialised to zero
  for i := 1 to m {
    for j := 1 to n {
      if (A[i] == B[j]) L[i][j] := L[i-1][j-1] + 1
      else L[i][j] := max(L[i-1][j], L[i][j-1])
    }
  }
}
```

Step 3: Tabular Solution

- LCS of bdcaba and abcbdab

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1
b	0	1	1	1	1	2	2
c	0	1	1	2	2	2	2
b	0	1	1	2	2	3	3
d	0	1	2	2	2	3	3
a	0	1	2	2	3	3	4
b	0	1	2	2	3	4	4

Time and Space Complexity

- Space complexity: O(mn)
  - m+1 rows and n+1 columns
- Time complexity:
  - Fill each cell in the table sequentially
  - Each cell value can be determined in O(1) time (by checking whether $a_i = b_j$, and looking at adjacent cell values)
  - Hence time complexity = O(mn)

Step 4: Retrieving the Actual LCS

- Start from L[m][n] and "trace back" the decisions

		b	d	c	a	b	a
	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1
b	0	1	1	1	1	2	2
c	0	1	1	2	2	2	2
b	0	1	1	2	2	3	3
d	0	1	2	2	2	3	3
a	0	1	2	2	3	3	4
b	0	1	2	2	3	4	4

```
bdcab a
|  | | |
ab c bdab
```

- (can have more than one possible solutions)

Sequence Alignment

- Problem: given two sequences A[1..m] and B[1..n], change one into the other with minimum “cost”
  - Operations: insertion, deletion, substitution
  - Each operation have a cost
  - Also called “edit distance”
- Example:

```
a bcaabcabc
 |   | |   |
aabca bbabcc
```

- Applications: bioinformatics, text editing (spell checking)
- Similar to the LCS problem

Recursive Formulation of ED

- Assume all costs = 1
- Let D(i, j) = edit distance of A[1..i] and B[1..j]
- For any i and j,

$$D(i, j) = \begin{cases} D(i-1, j-1) & \text{if } A[i] = B[j] \\ 1 + \min \begin{cases} D(i-1, j) \\ D(i, j-1) \\ D(i-1, j-1) \end{cases} & \text{if } A[i] \neq B[j] \end{cases}$$

- Base case: D(0, j) = j, D(i, 0) = i (why?)
- Similar dynamic programming algorithm

Sequence Alignment Example

- Example

		b	d	c	a	b	a
	0	1	2	3	4	5	6
a	1	1	2	3	3	4	5
b	2	1	2	3	4	3	4
c	3	2	2	2	3	4	4
b	4	3	3	3	3	3	4
d	5	4	3	4	4	4	4
a	6	5	4	4	4	5	4
b	7	6	5	5	5	4	5

Edit distance = 5

Alignment:

```
 bdcab a
 | | | |
ab c bdab
```

Shortest Paths Revisited

(1) Directed Acyclic Graphs

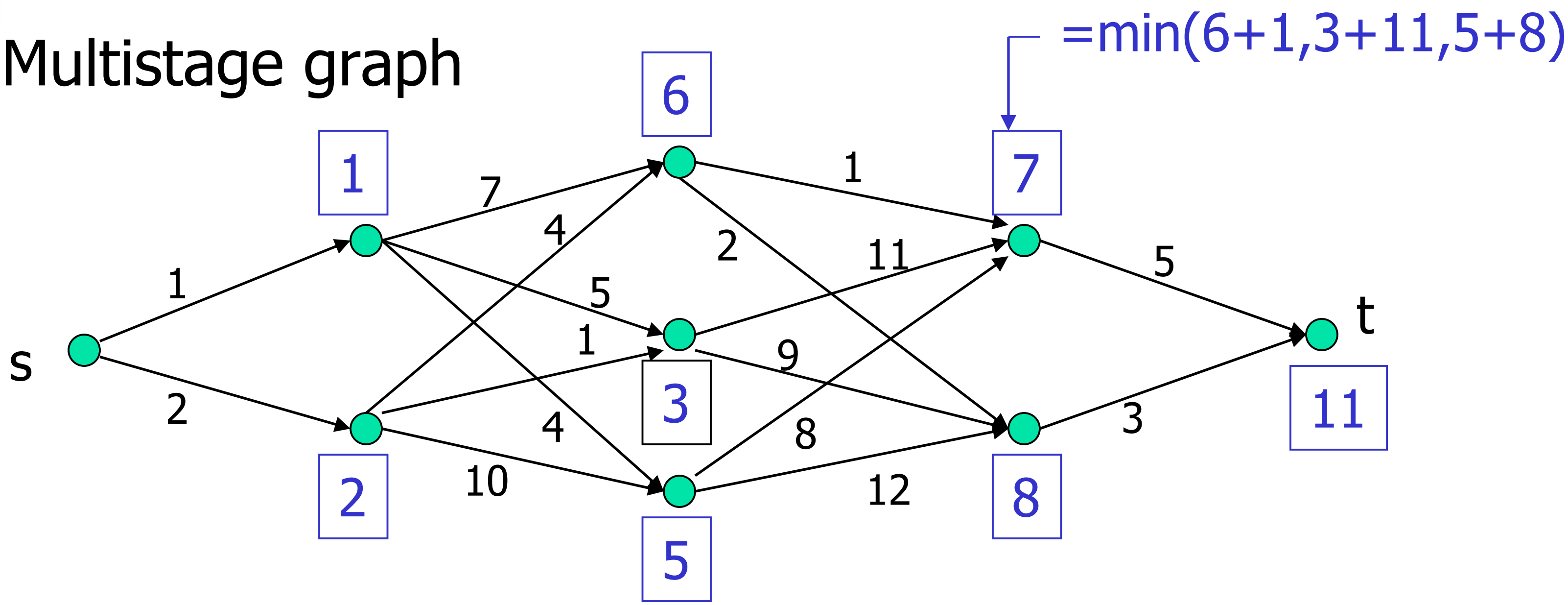
- A core part of Dijkstra's algorithm is to check for shorter distances:

```
if (D[u] + d(u,v) < D[v])
  D[v] := D[u] + d(u, v)
```

  - This operation is always "safe"
  - If we apply this procedure to each edge "in some right order", then the shortest paths can be found
- *Directed Acyclic Graph (DAG)*
  - A directed graph without a (directed) cycle
  - Shortest paths can be computed easily

Applying Edge Updates in DAGs

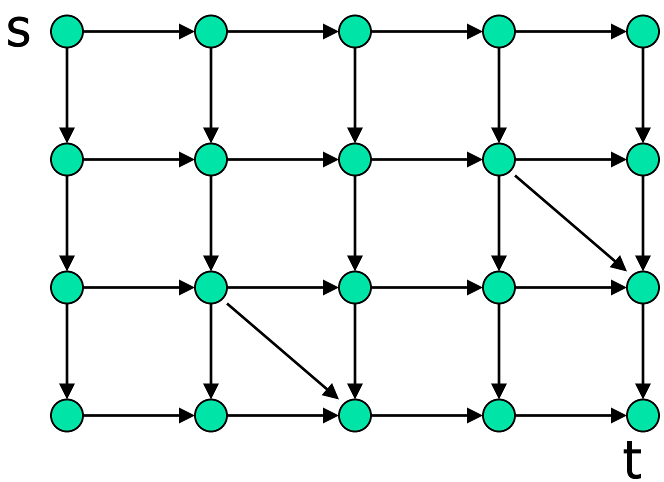
- Multistage graph



- Just apply the edge update procedure to all edges in the “correct” order (left to right)
  - Time complexity: O(m)
- Can be viewed as dynamic programming

Grid Graphs and Relation to LCS

- Grid graph
  - E.g. Horizontal & vertical edge weights = 2, diagonal edge weights = 3
  - Let D(i, j) = distance from s to node at row i, column j
  - D(i,j) = min(D(i-1,j)+2, D(i,j-1)+2, D(i-1,j-1)+3)
    (if the corresponding edges exist)
  - Same as LCS/edit distance!

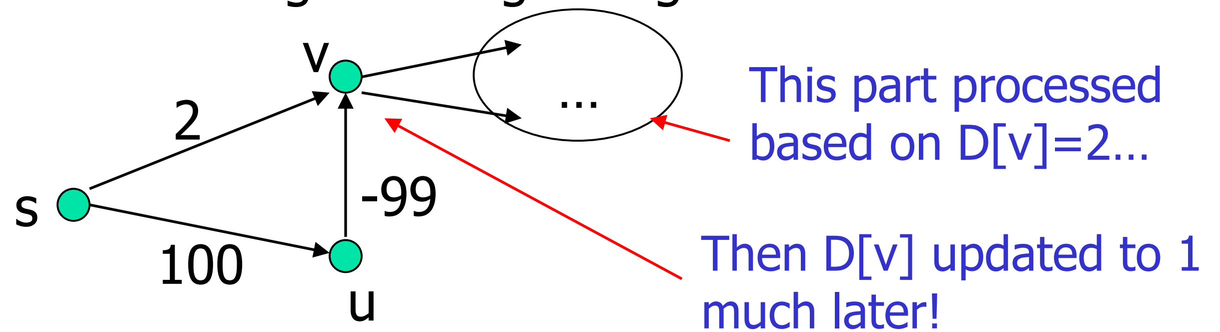


Any DAG

- More generally, for any DAG
  - we can assign ordering to vertices so that any edge always goes from a smaller vertex to a larger vertex (*topological sort*; details omitted)
  - This allows us to apply the edge update procedure only once, following the natural order, to give correct shortest paths
- Can solve more general problems:
  - Negative edges (not possible with Dijkstra)
  - Other optimizations e.g. longest paths (no efficient algorithm at all for non-DAG)

(2) Negative Edges and Cycles

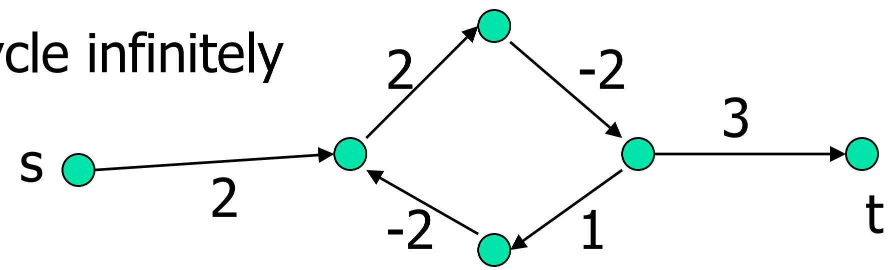
- Recall Dijkstra's algorithm for shortest paths: extending wavefront approach
  - Does not work for negative edge weights
  - Example:



- We will develop an algorithm for finding shortest paths when the graph have negative edges
  - More general
  - Similar ideas used in decentralised settings, i.e. no node has global information (e.g. distributed routing algorithms)

Algorithm for Negative-Edge Case

- The presence of negative edges may introduce *negative cycles* (cycles with total edge weight < 0)
- A graph with negative cycles does not have a well-defined shortest path
  - Go through negative cycle infinitely



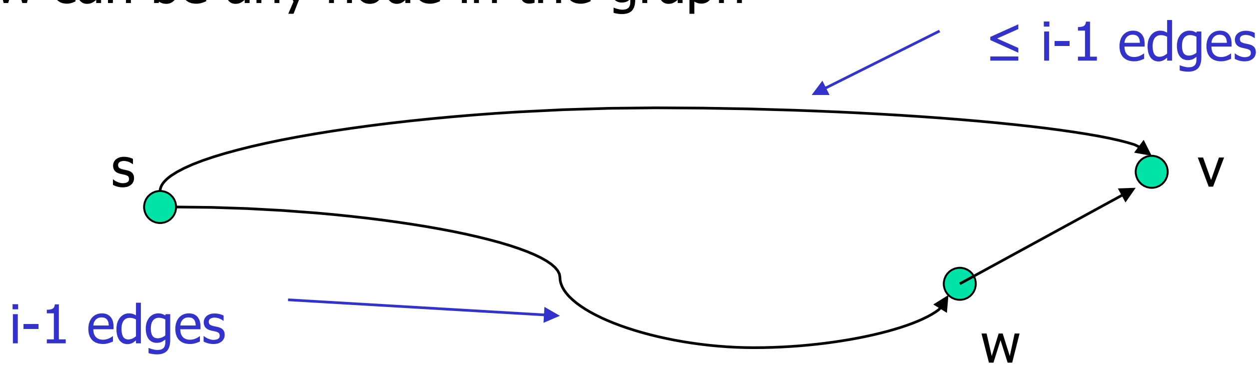
- A graph with negative edges but not negative cycles still has shortest paths well-defined
- We will assume the graphs we consider have no negative cycles

Two Properties of Shortest Paths

- Shortest paths do not contain cycles
  - Positive cycle: can remove to give a shorter path
  - Negative cycle: assumed not to exist
- Shortest paths have at most n-1 edges
  - A path with > n-1 edges has > n vertices $\Rightarrow$ some vertices must be visited twice $\Rightarrow$ cycles

A Dynamic Programming Formulation

- Let S(i, v) denote the shortest path length from s to v using a path *with at most i edges*
- Observation 1: in the path S(i, v), either it uses fewer than i edges, or it uses exactly i edges
- Observation 2: if it uses exactly i edges, let (w,v) be the last edge in this path. Then
  - We need optimal solution from s to w (i-1 edges)
  - w can be any node in the graph



Recursive Formula and Algorithm

- Which choice is correct? Take the shortest one
- So we have

$$S(i, v) = \min \begin{cases} S(i-1, v) \\ \min\{ S(i-1,w) + d(w,v) \} \text{ for all } w \text{ in } V \end{cases}$$

- Algorithm:

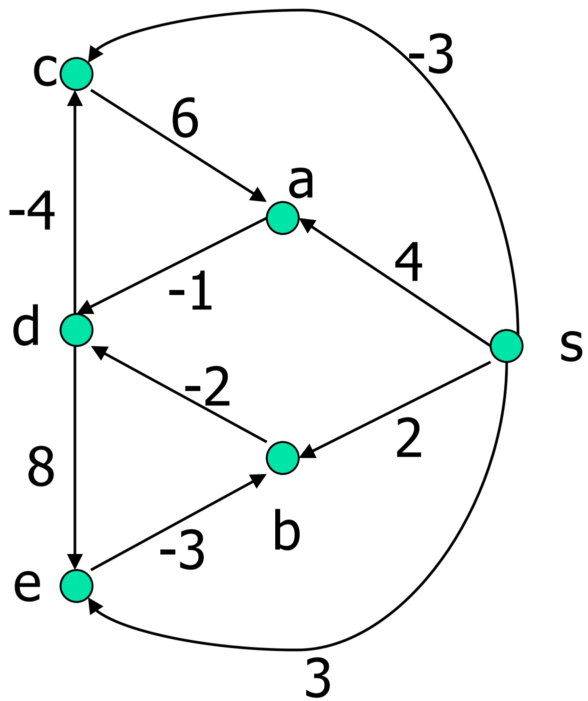
```
SP(G, s)
  // initialise array M
  M[0][s] := 0
  M[0][v] := infinity for all other v

  for i := 1 to n-1
    for each node v  // in any order
      compute M[i][v] using formula above
```

$i \leq n-1$ since shortest paths have at most n-1 edges

Example

- From s to all other nodes



	0	1	2	3	4	5
s	0	0	0	0	0	0
a	∞	4	3	3	2	0
b	∞	2	0	0	0	0
c	∞	-3	-3	-4	-6	-6
d	∞	∞	0	-2	-2	-2
e	∞	3	3	3	3	3

Example: S(2, a) =
min(S(1,a), S(1,c)+d(ca))

Simplifying the Algorithm

- Observe that S(i, *) depends on S(i-1, *) only; no need to keep earlier entries (S(i-2, *), …, S(0, *))
- Reuse memory
- Simplified recursive formula: let M[v] be an 1-D array. Then $M[v] = \min(M[v], \min_{\text{all } w} \{ M[w] + d(w, v) \})$
- We can further simplify the algorithm to give the *Bellman-Ford algorithm*
  - Principle: apply the edge update procedure to all edges. We don't know the right order, so choose an arbitrary one and do it a "large enough" number of times
  - (proof of correctness omitted)

Bellman-Ford Algorithm

```
Bellman-Ford(G, s)
  D[s] := 0
  D[v] := ∞ for all v except s
  for i := 1 to n-1 {
    for each edge (u, v) { // in any order
      if (D[v] > D[u] + d(u, v)) {
        D[v] := D[u] + d(u, v)
        P[v] := u
      }
    }
  }
```

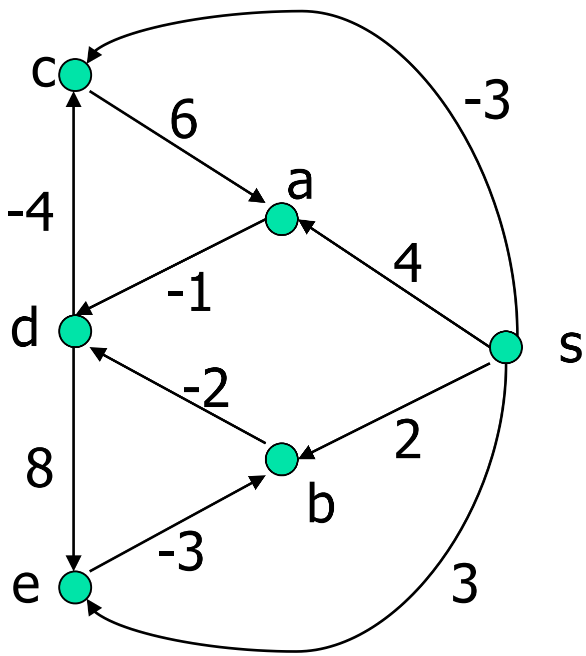
Note: i has no meaning anymore

predecessor information

- Time complexity: O(mn)
- Space complexity: O(n)

Example

- Edges: (sa) (sb) (sc) (se) (ca) (ad) (bd) (eb) (dc) (de)



(Values updated more than once in one round)

initial

v	a	b	c	d	e
D[v]	∞	∞	∞	∞	∞

i = 1

v	a	b	c	d	e
D[v]	~~4~~	~~2~~	~~-3~~	~~2~~	3
	3	0	-4	0	

i = 2

v	a	b	c	d	e
D[v]	2	0	-4	-2	3

And so on…

(3) All-Pairs Shortest Paths

- Given a graph, we want to find shortest paths between *all pairs of vertices*
- Straightforward approach: run single-source shortest path algorithms at each vertex
  - Positive edge weights: Dijkstra's algorithm. $O(m \log n) \times n = O(mn \log n)$
  - General edge weights: Bellman-Ford algorithm. $O(mn) \times n = O(mn^2)$
- Faster algorithms?
  - Dynamic programming…

Using Bellman-Ford in APSP

- Adapt the dynamic programming formulation in Bellman-Ford algorithm to the all-pairs case
- Let $D_{ij}^{(k)}$ denote shortest distance from v_i to v_j using at most k edges
- $D_{ij}^{(k)} = \min\left(D_{ij}^{(k-1)}, \min_{\text{all } w}\{ D_{iw}^{(k-1)} + d(w,j) \} \right)$
- This gives an $O(n^4)$ time algorithm
- Can be improved to $O(n^3 \log n)$
  - Details omitted. See textbook if interested

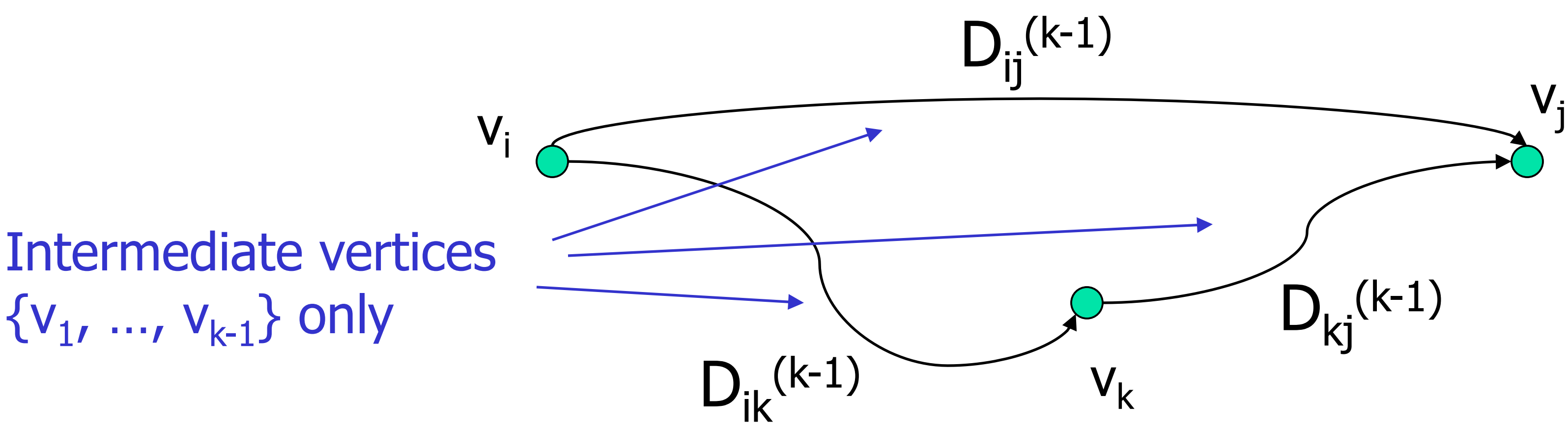
Faster Algorithm: Another DP

- Consider an alternative DP formulation:
- Let $D_{ij}^{(k)}$ denote shortest distance from vertex v_i to v_j, *using intermediate vertices with label at most k*
  - $D_{ij}^{(0)}$: edge from v_i to v_j (no intermediate vertex)
  - $D_{ij}^{(n)}$: shortest path (can use any intermediate vertex)

- Consider a pair of vertices v_i and v_j, and $D_{ij}^{(k)}$
- Observation: either the shortest path $D_{ij}^{(k)}$ uses v_k as intermediate vertex, or it does not (i.e. uses intermediate vertices with label at most k-1)

DP using Intermediate Vertices

- This allows us to *reduce to smaller subproblems:*
  - If it uses intermediate vertex v_k, then two paths $v_i \rightarrow v_k$ and $v_k \rightarrow v_j$ uses intermediate vertices with label at most k-1
  - If it does not use v_k, simply have a smaller subproblem
  - Pick the shorter one

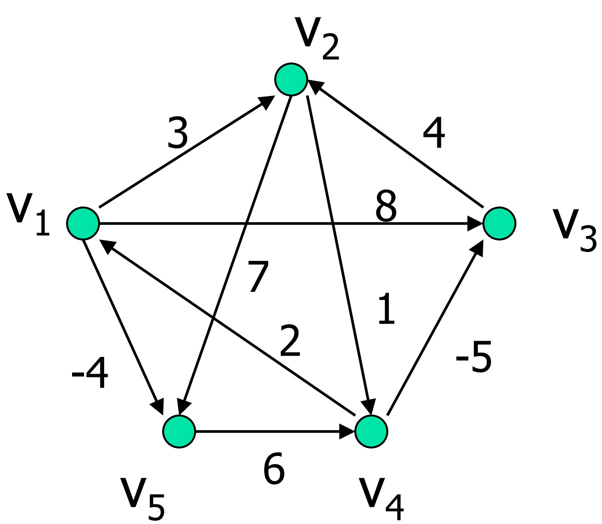


Floyd-Warshall Algorithm

- We arrange $D_{ij}^{(k)}$ in matrices
  - Let $D^{(k)}$ be an n x n matrix, k = 0, 1, …, n
  - $D_{ij}^{(k)}$ = i-th row, j-th column entry in $D^{(k)}$
- Recursion: $D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$

```
Floyd-Warshall() {
  D(0) := W  // adjacency matrix
  for k := 1 to n {
    for i := 1 to n {
      for j := 1 to n {
        Dij(k) := min( Dij(k-1), Dik(k-1) + Dkj(k-1) )
      }
    }
  }
}
```

Example



$D^{(0)}$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

$P^{(0)}$

/	1	1	/	1
/	/	/	2	2
/	3	/	/	/
4	/	4	/	/
/	/	/	5	/

$D^{(1)}$

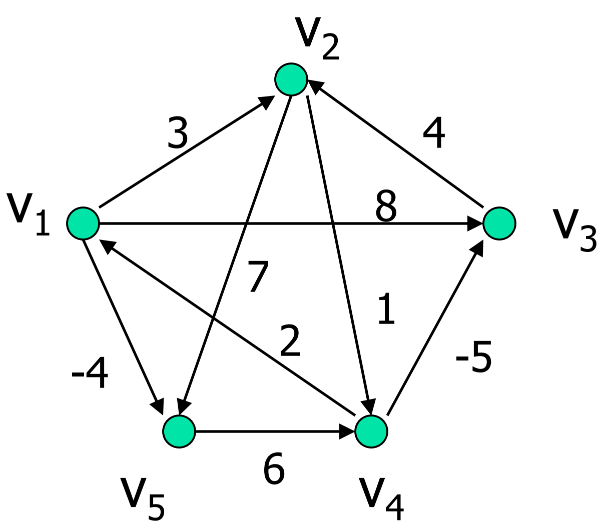
0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

$P^{(1)}$

/	1	1	/	1
/	/	/	2	2
/	3	/	/	/
4	1	4	/	1
/	/	/	5	/

Example: $D_{42}^{(1)} = \min(D_{42}^{(0)}, D_{41}^{(0)}+D_{12}^{(0)})$

Example (cont'd)



$D^{(2)}$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	5	-5	0	-2
∞	∞	∞	6	0

$P^{(2)}$

/	1	1	2	1
/	/	/	2	2
/	3	/	2	2
4	1	4	/	1
/	/	/	5	/

$D^{(3)}$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

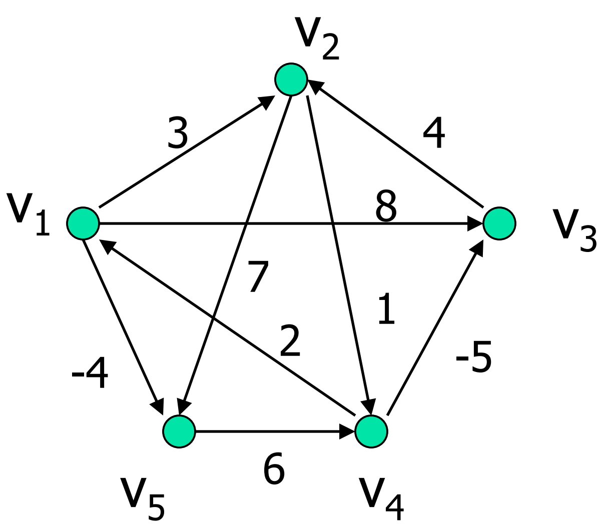
$P^{(3)}$

/	1	1	2	1
/	/	/	2	2
/	3	/	2	2
4	3	4	/	1
/	/	/	5	/

Example: $D_{42}^{(3)} = \min(D_{42}^{(2)}, D_{43}^{(2)}+D_{32}^{(2)})$

Example (cont'd)

These are not 4 (why?)



$D^{(4)}$

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

$P^{(4)}$

/	1	4	2	1
4	/	4	2	1
4	3	/	2	1
4	3	4	/	1
4	3	4	5	/

$D^{(5)}$

0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

$P^{(5)}$

/	3	4	5	1
4	/	4	2	1
4	3	/	2	1
4	3	4	/	1
4	3	4	5	/

Time and Space Complexity

- Time complexity: $O(n^3)$
  - 3 nested for loops
- Space complexity: $O(n^2)$
  - Each D array has n x n elements
  - No need to keep all D arrays, just the most recent one

- To find the actual shortest paths (not just the distances), keep track of predecessor information in another array, as before…

Finding the Actual Shortest Paths

```
Floyd-Warshall() {
  D(0) := W
  Pij(0) := i if (i, j) is an edge,
    otherwise nil
  for k := 1 to n {
    for i := 1 to n {
      for j := 1 to n {
        if (Dij(k-1) < Dik(k-1) + Dkj(k-1)) {
          Dij(k) := Dij(k-1)
          Pij(k) := Pij(k-1)
        } else {
          Dij(k) := Dik(k-1) + Dkj(k-1)
          Pij(k) := Pkj(k-1)
        }
}}}}
```