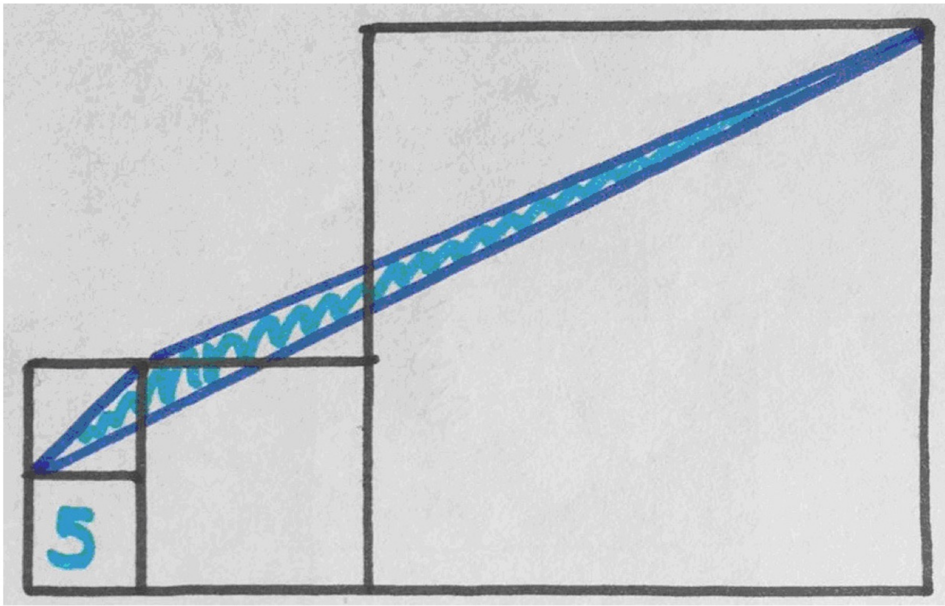
CS2040S
Data Structures and Algorithms

Dynamic Programming…

Puzzle of the Week:

The area of the bottom left square is 5. What's the area of the blue triangle?



Catriona Agg
https://twitter.com/cshearer41/status/1027844515338616832

Roadmap

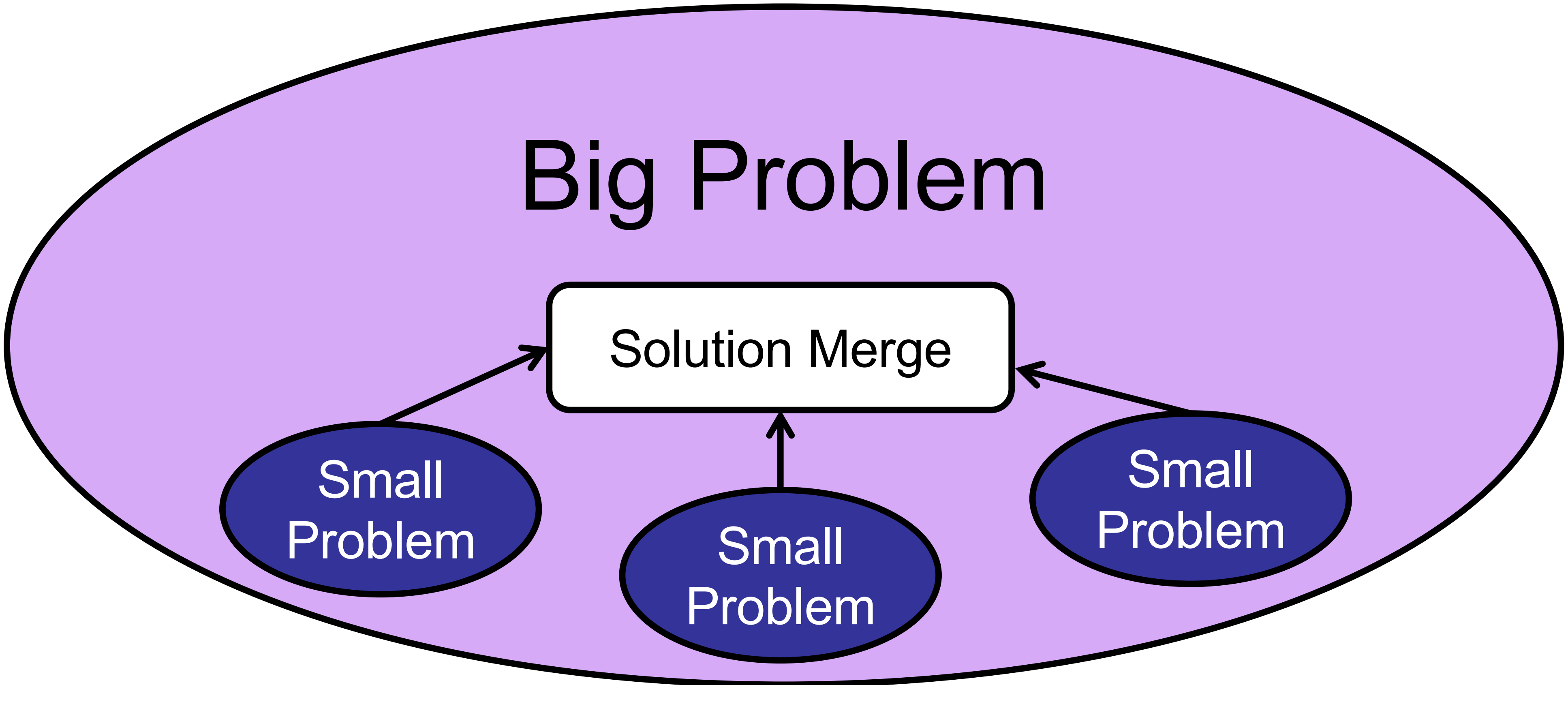
Dynamic Programming

- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths

Dynamic Programming Basics

Optimal sub-structure:

Optimal solution can be constructed from optimal solutions to smaller sub-problems.



Optimal Sub-structure

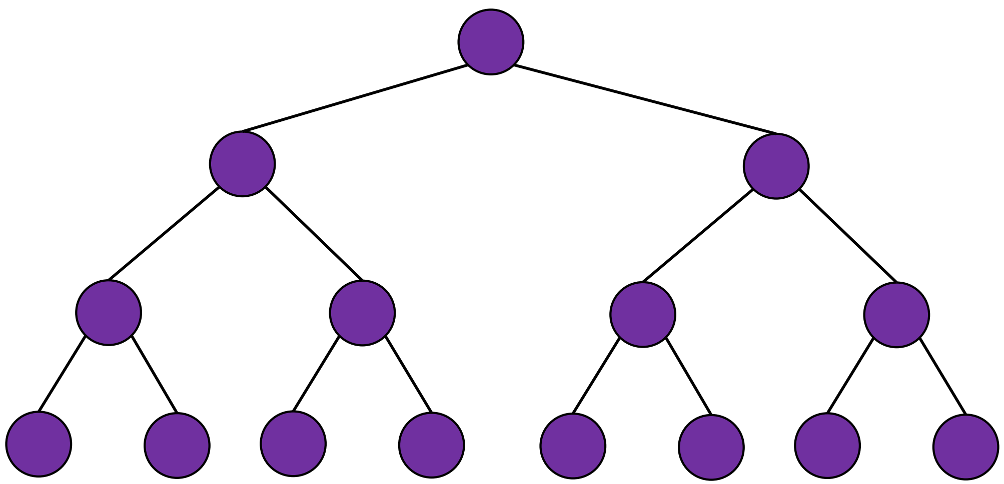
Property of (nearly) every problem we study:

- Greedy algorithms
  - Dijkstra's Algorithm
  - Minimum Spanning Tree algorithms
- Divide-and-conquer algorithms
  - MergeSort
  - Fast Fourier Transform

Dynamic Programming

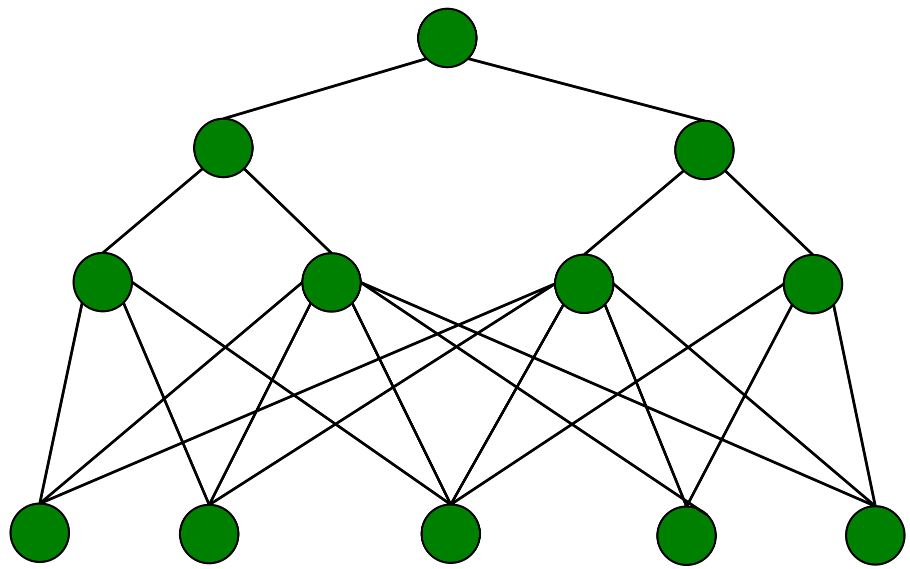
Contrast: Both have optimal substructure

No overlapping subproblems



Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

Dynamic Programming

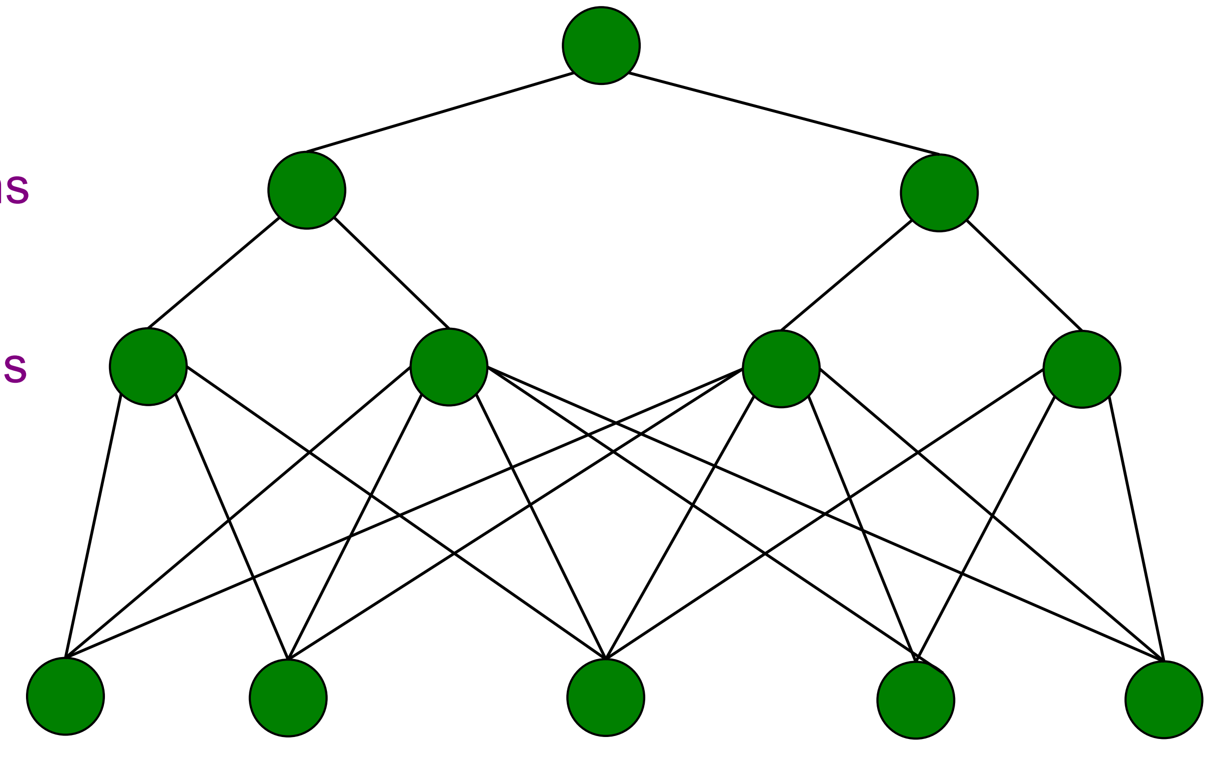
Basic strategy: *(bottom up dynamic programming)*

Step 4: solve root problem

Step 3: combine smaller problems

Step 2: combine smaller problems

Step 1: solve smallest problems

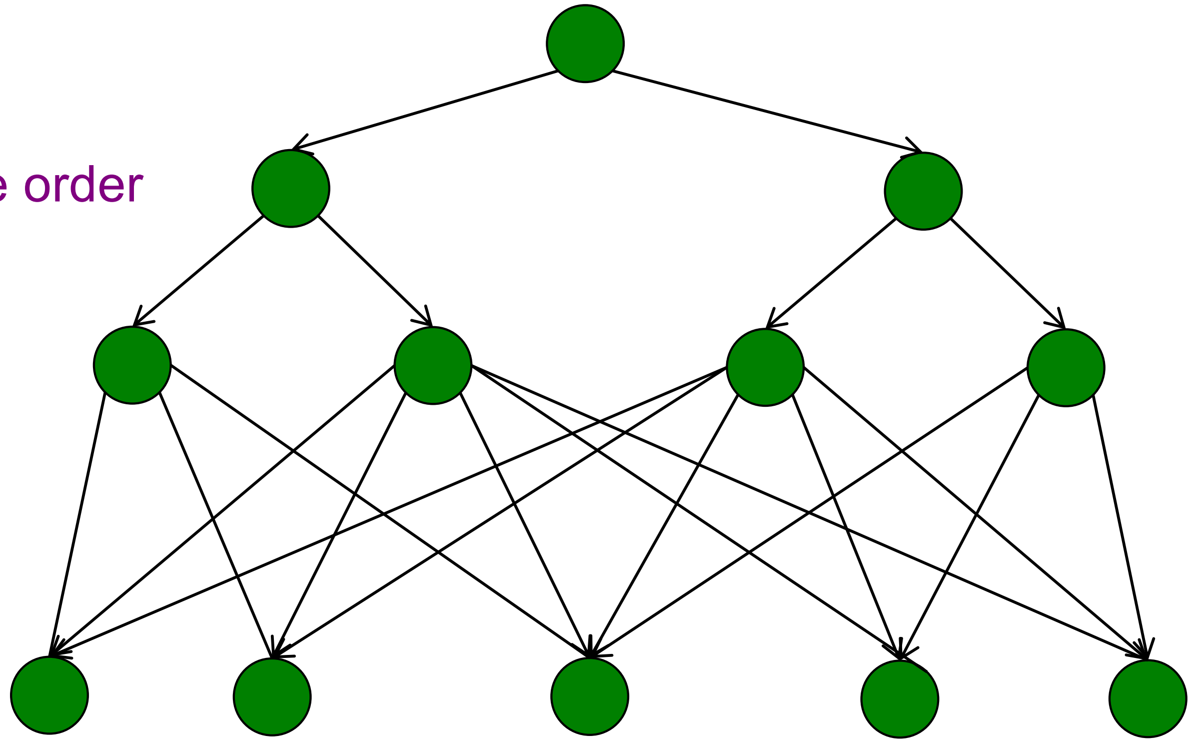


Dynamic Programming

Basic strategy: *(DAG + topological sort)*

Step 1: Topologically sort DAG

Step 2: Solve problems in reverse order



Dynamic Programming

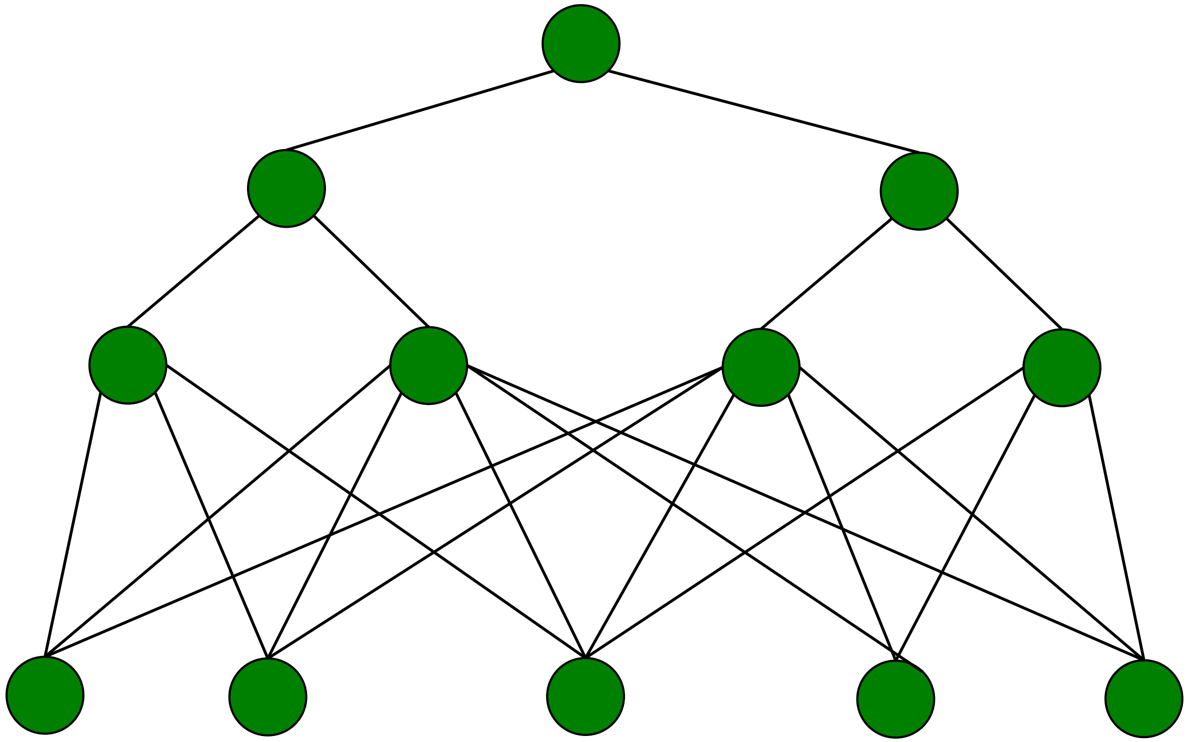
Basic strategy: *(top down dynamic programming)*

Step 1: Start at root and recurse.

Step 2: Recurse.

Step 3: Recurse.

Step 4: Solve and memoize.
Only compute each
solution once.



Dynamic Programming

Table view:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
1	17	22	14	19	8	4	9	12	15	7	5	9	13	14	18	4
2	15	12	13	13	7											
3																
4																
5																
6																
7																
8																
9																
10																
11																

Roadmap

Dynamic Programming

Basics of DP

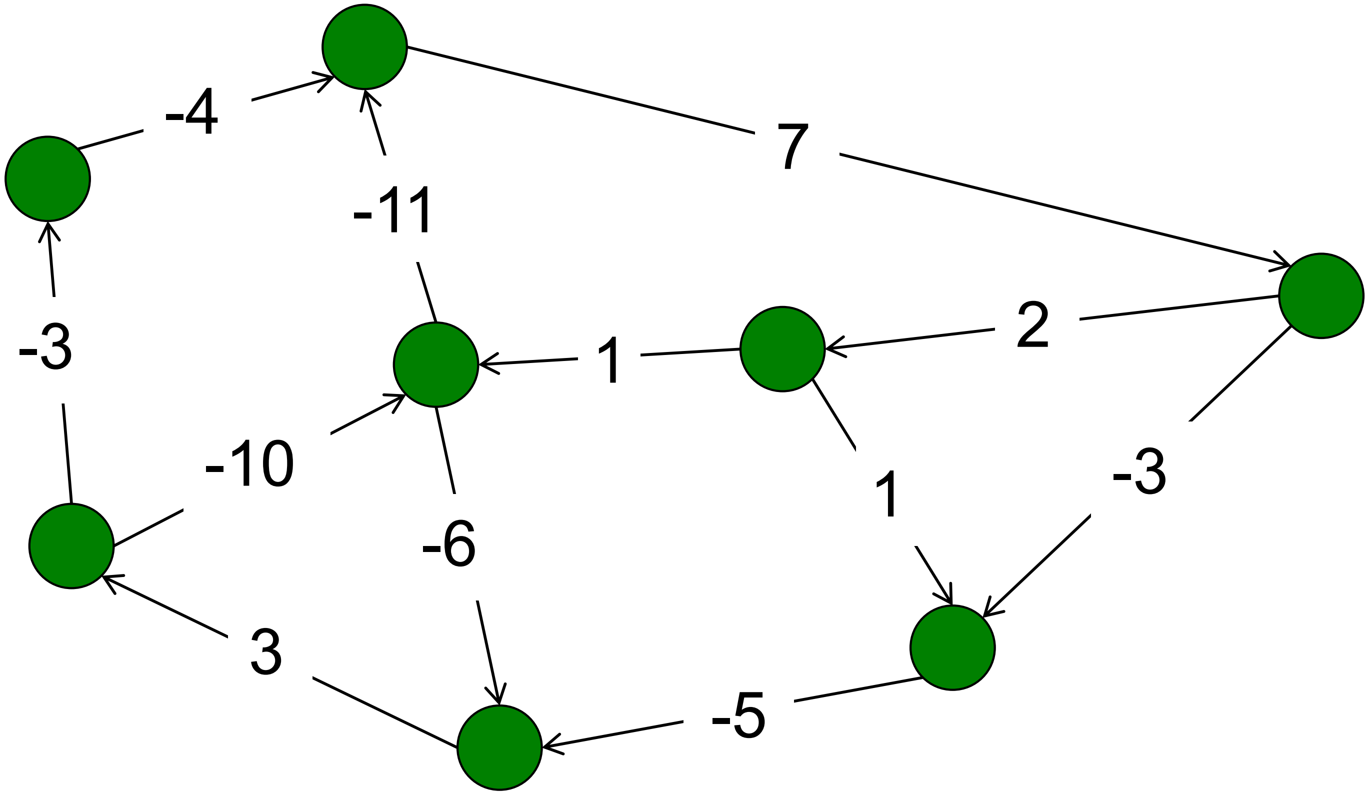
Example: Longest Increasing Subsequence

- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths

Prize Collecting

Input:

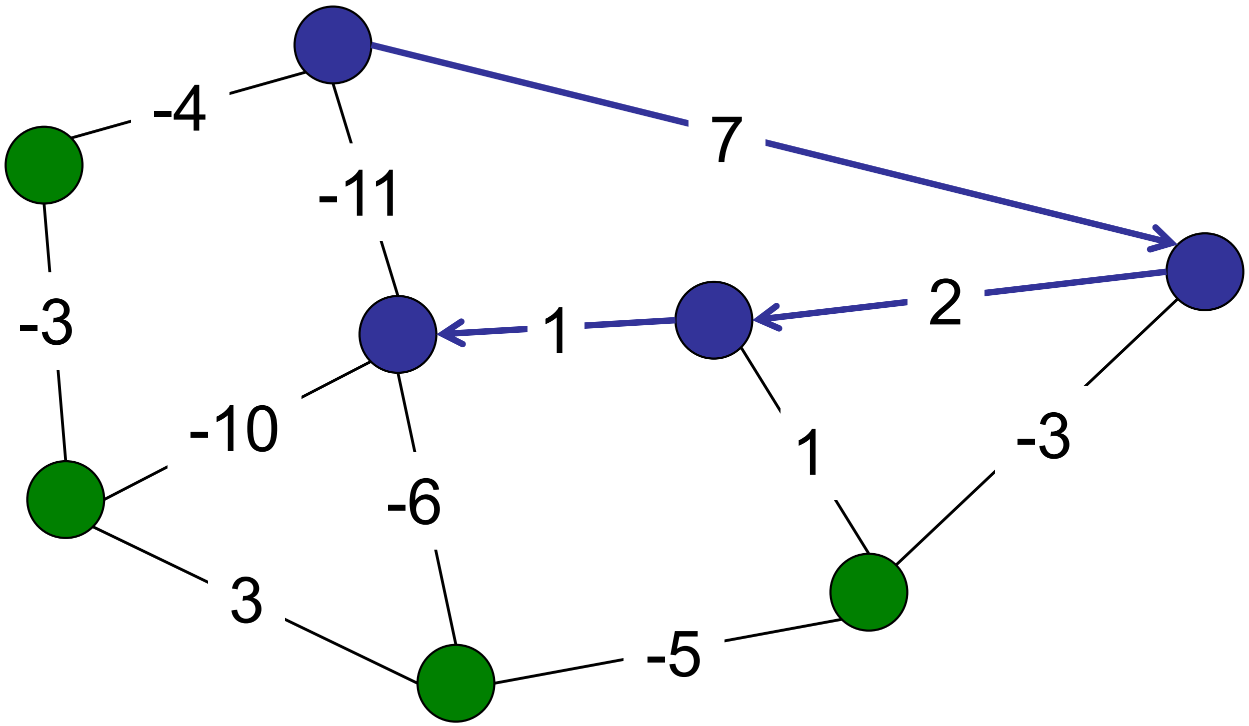
- Directed Graph G = (V,E)
- Edge weights **w** = prizes on each edge



Prize Collecting

Output:

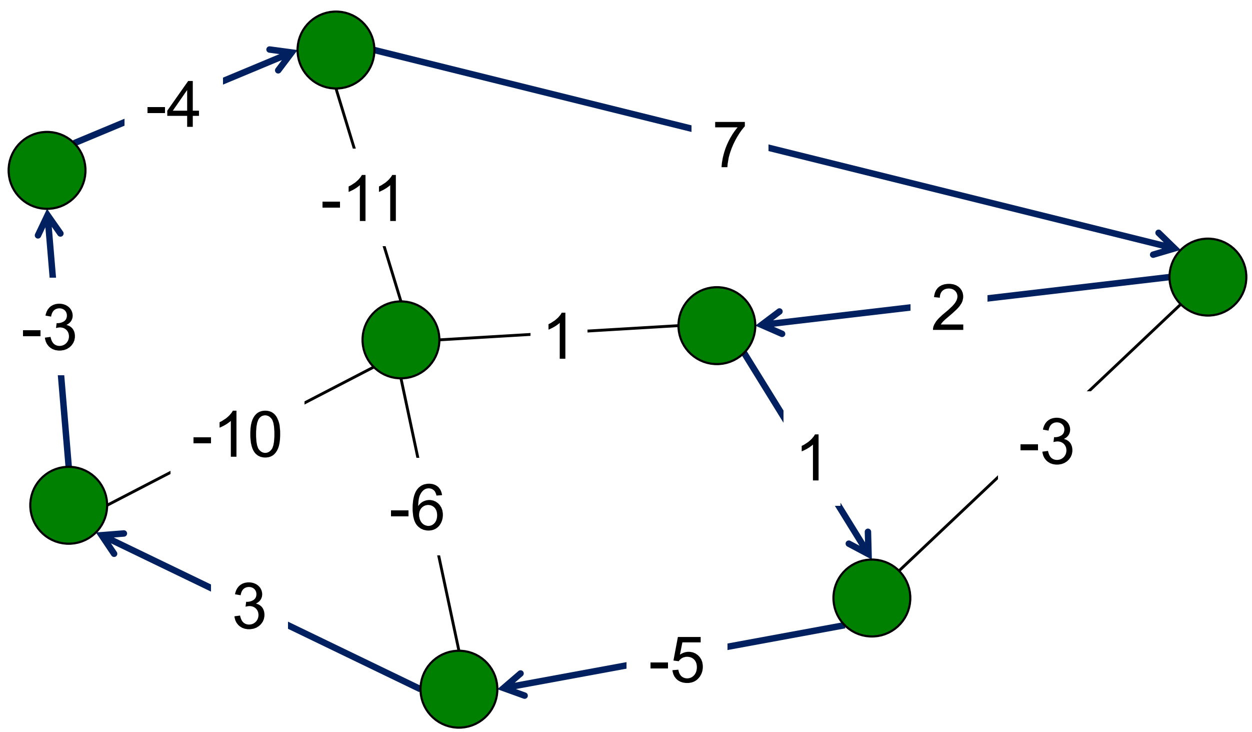
- Prize collecting path
- Example: 7 + 2 + 1 = 10



Prize Collecting

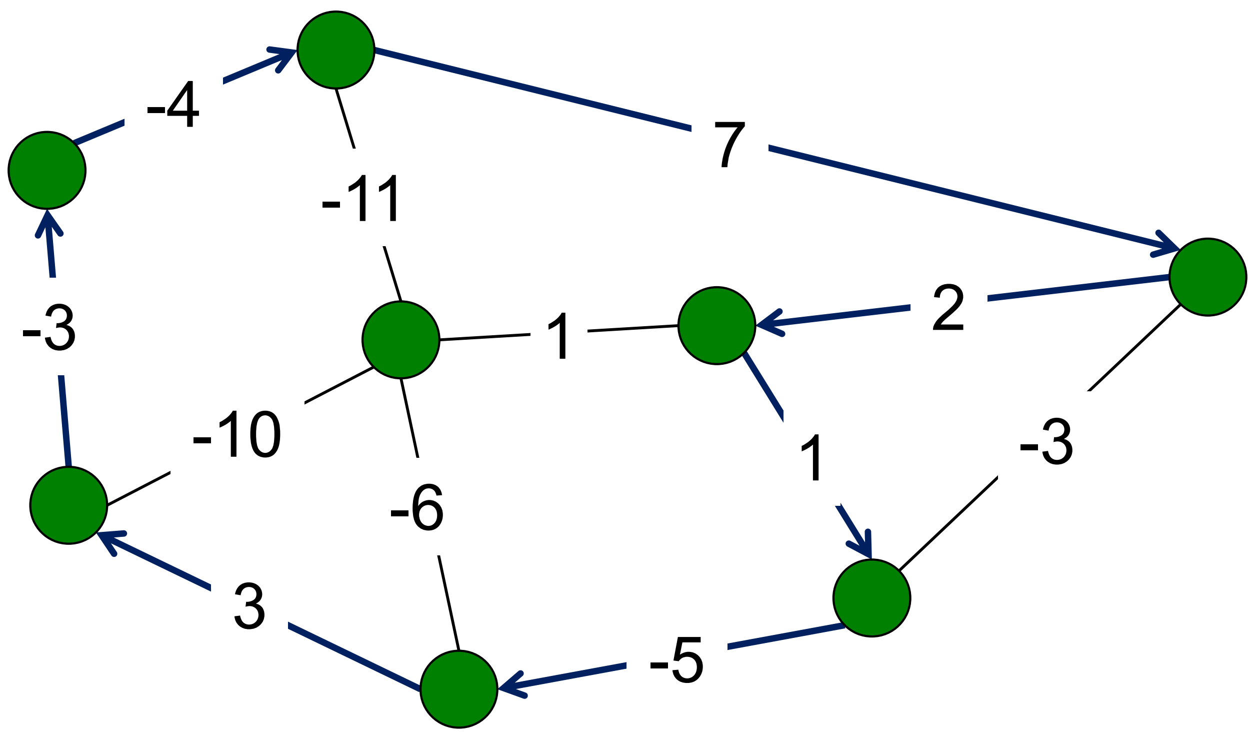
Output:

- Prize collecting path: 7 + 2 + 1 - 5 + 3 - 3 - 4 = 1
- Positive weight cycle → infinite prizes!



Prize Collecting

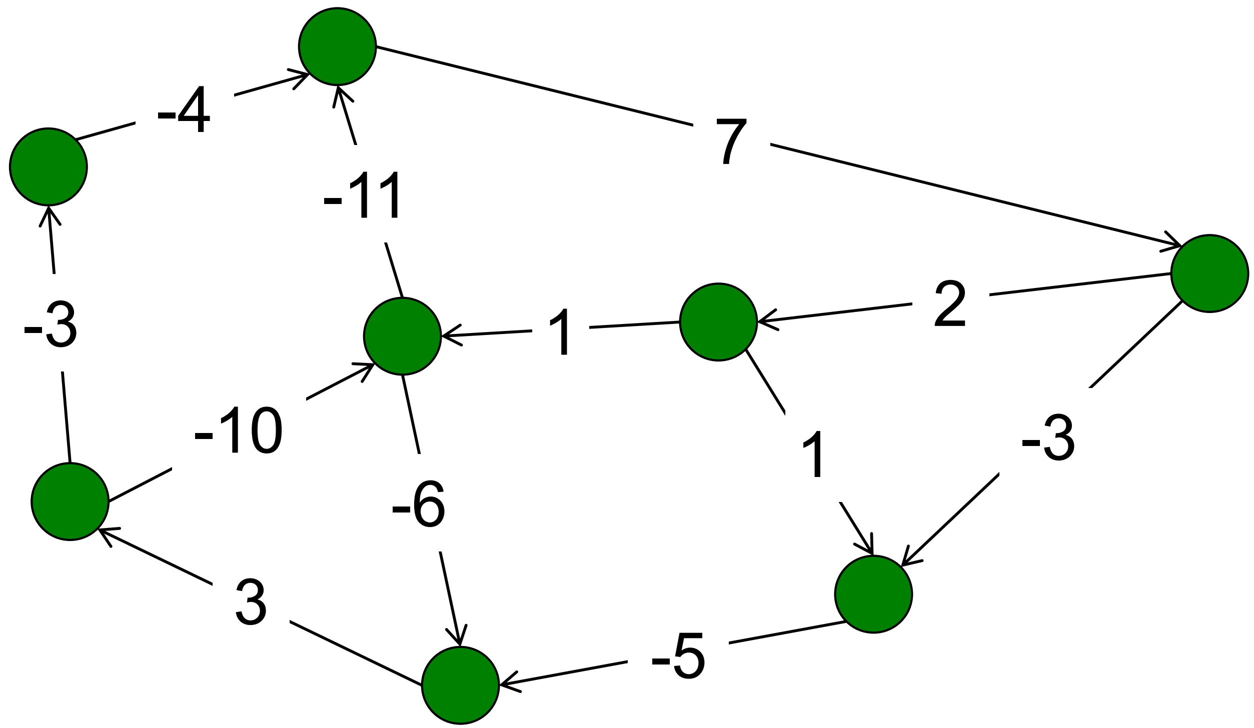
Check for positive weight cycles using Bellman-Ford (negating the edges).



Lazy Prize Collecting

Input:

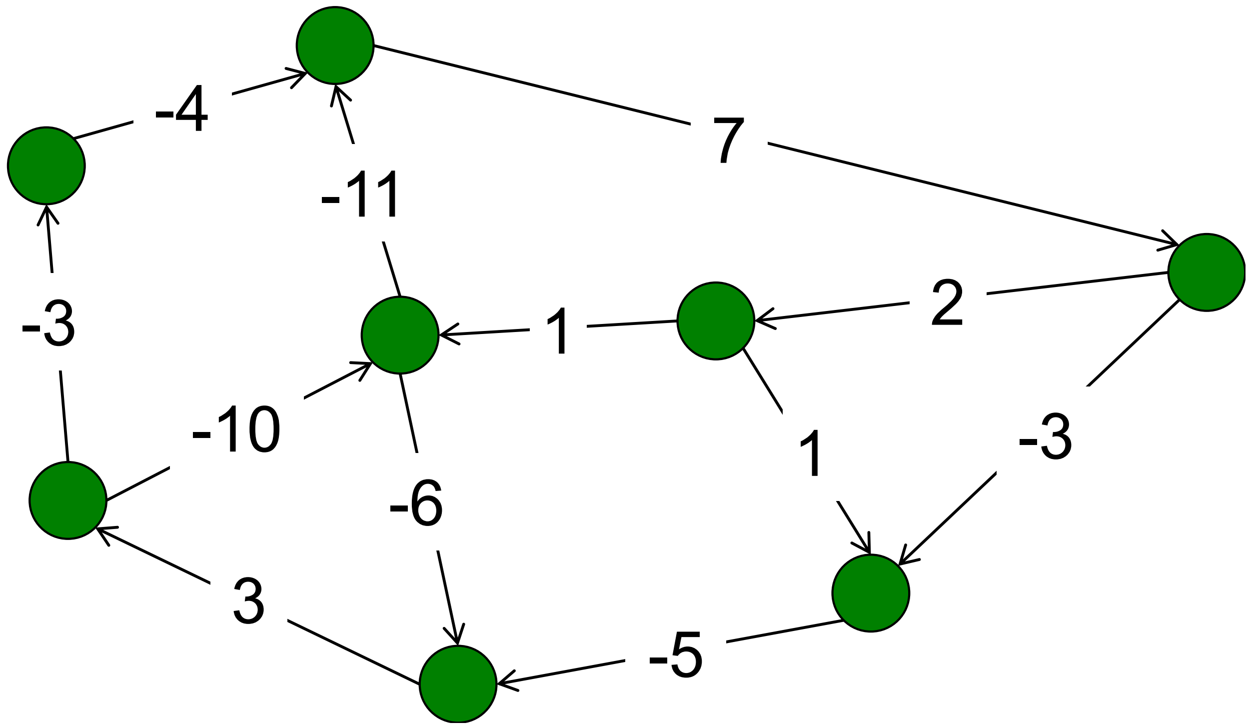
- Graph G = (V,E)
- Edge weights w = prizes on each edge
- Limit k: only cross at most k edges



Lazy Prize Collecting

Note: Not a shortest path problem

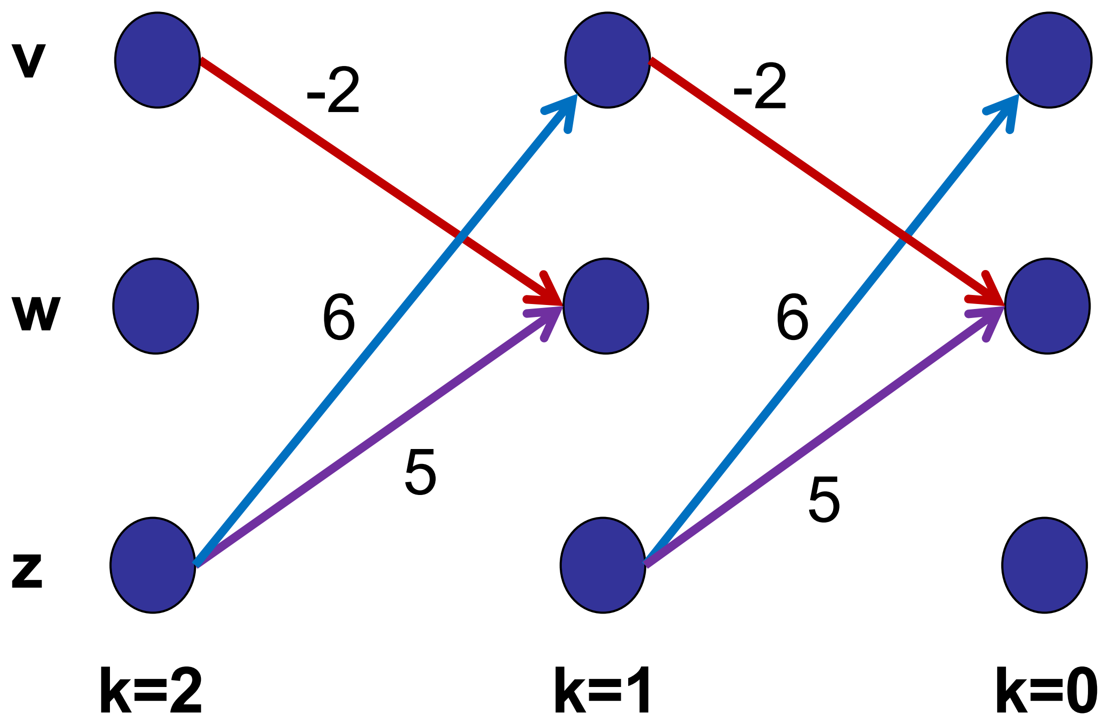
- Not a shortest path problem! Longest path…
- Negative weight cycles.
- Positive weight cycles.

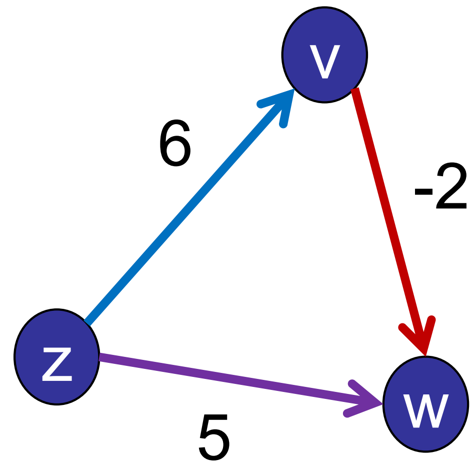


Lazy Prize Collecting

Idea 1:

- Transform G into a DAG
- Make **k** copies of every node: (v,1), (v,2), (v,3), …
- Solve prize collecting via DAG_SSSP (longest path)

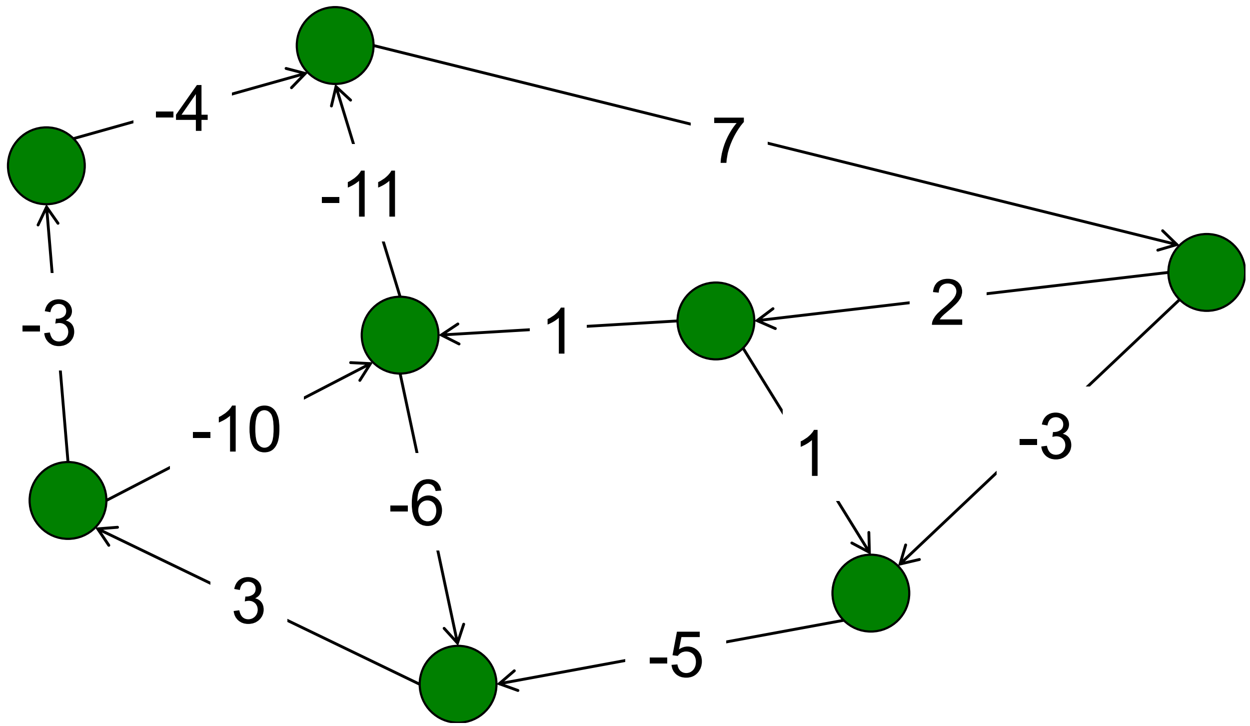




Lazy Prize Collecting

Idea 1:

- Transform G into a DAG
- Make **k** copies of every node: (v,1), (v,2), (v,3), …
- Solve longest-path problem for each source.



What is the running time of Idea 1?

1. $O(E)$
2. $O(VE)$
3. ✓ $O(kE)$
4. ✓ $O(kVE)$
5. $O(kV^2E)$
6. None of the above

Lazy Prize Collecting

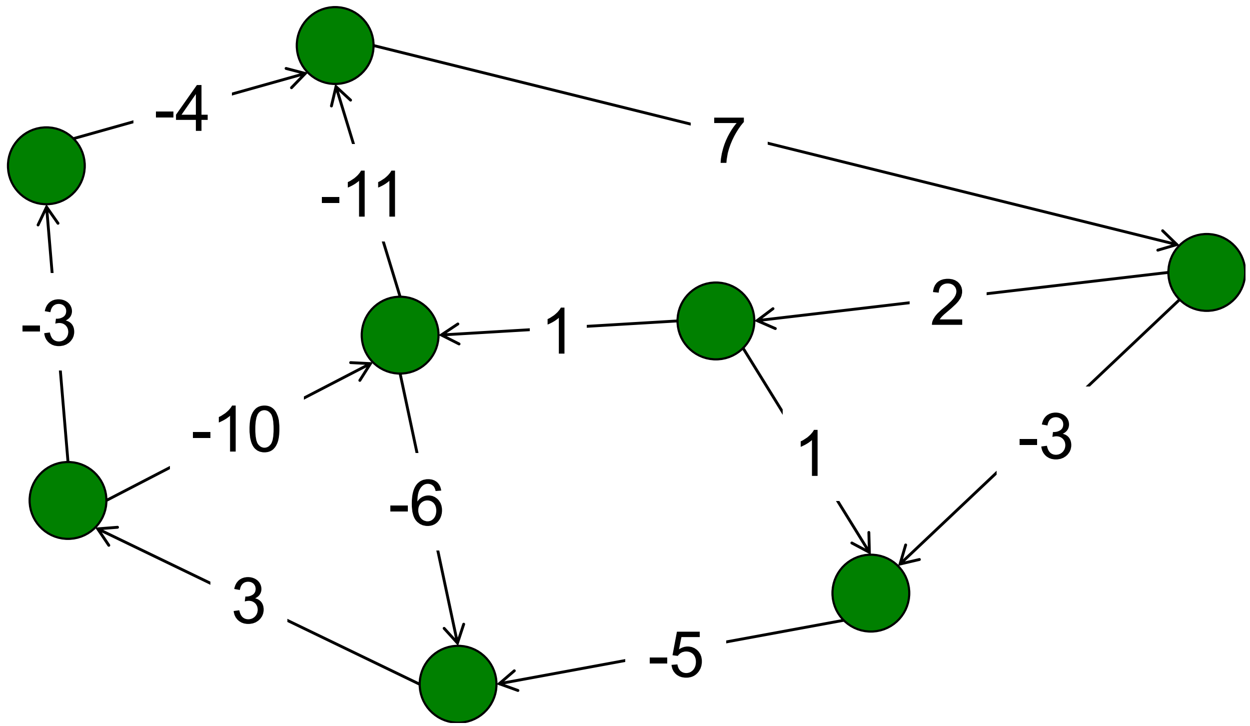
Running Time:

– Transformed graph: kV nodes, kE edges

– Topo-sort / Longest path: O(kV + kE)

– Once per source: repeat V times ➔ O(kVE)?

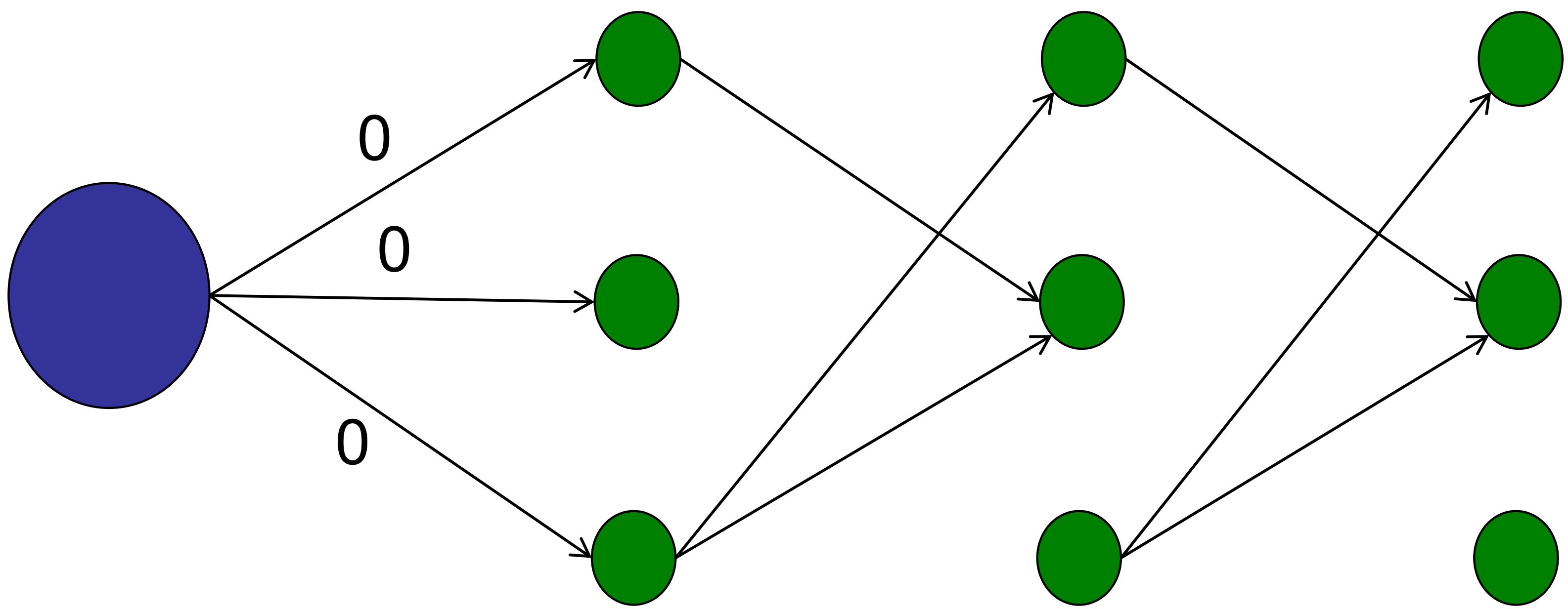
Whenever you transform a graph, do NOT forget to recompute the number of nodes and edges in the new graph.



Lazy Prize Collecting

Running Time:

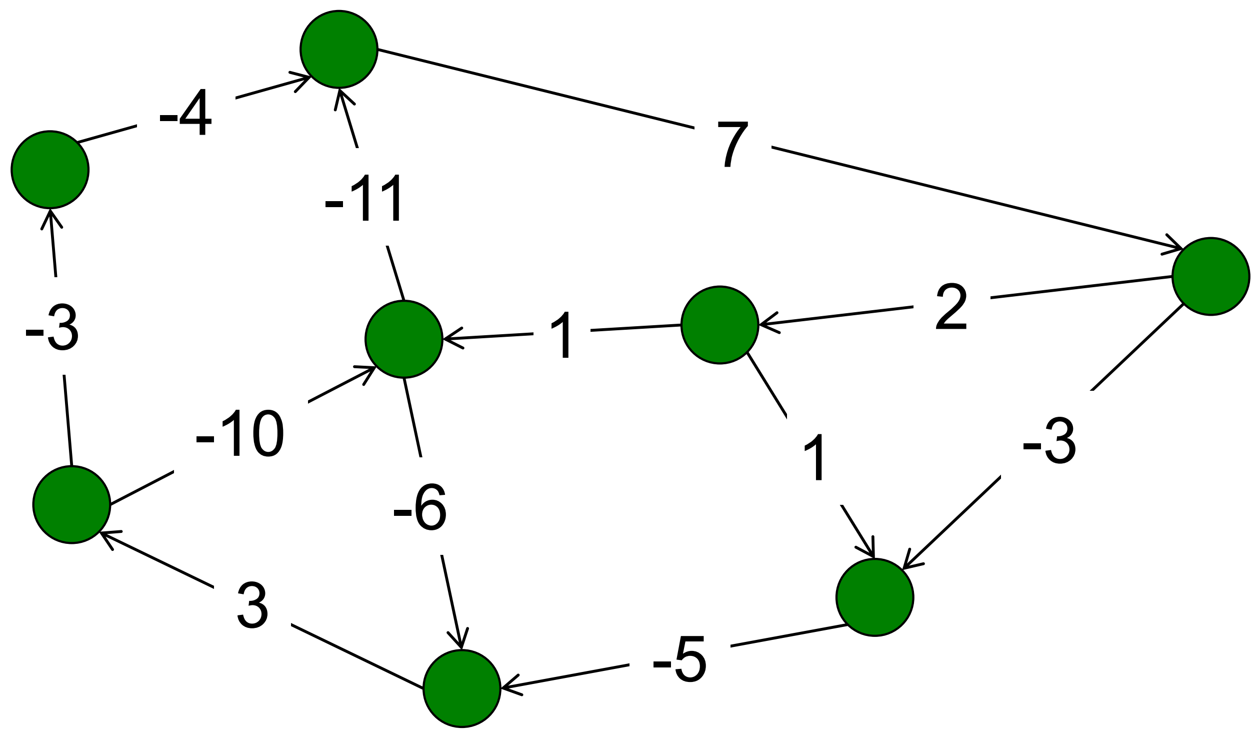
- Transformed graph: kV nodes, kE edges
- Topo-sort / Longest path: O(kV + kE)
- Create super-source….



Lazy Prize Collecting

Idea 2: Dynamic Programming

If you know the optimal solution for (k-1), then it is easy to computer optimal solution for k.



Dynamic Programming Recipe

Step 1: Identify optimal substructure

E.g., solution for (k-1) ➔ solution for k

Step 2: Define sub-problems

Step 3: Solve problem using sub-problems

Step 4: Write (pseudo)code.

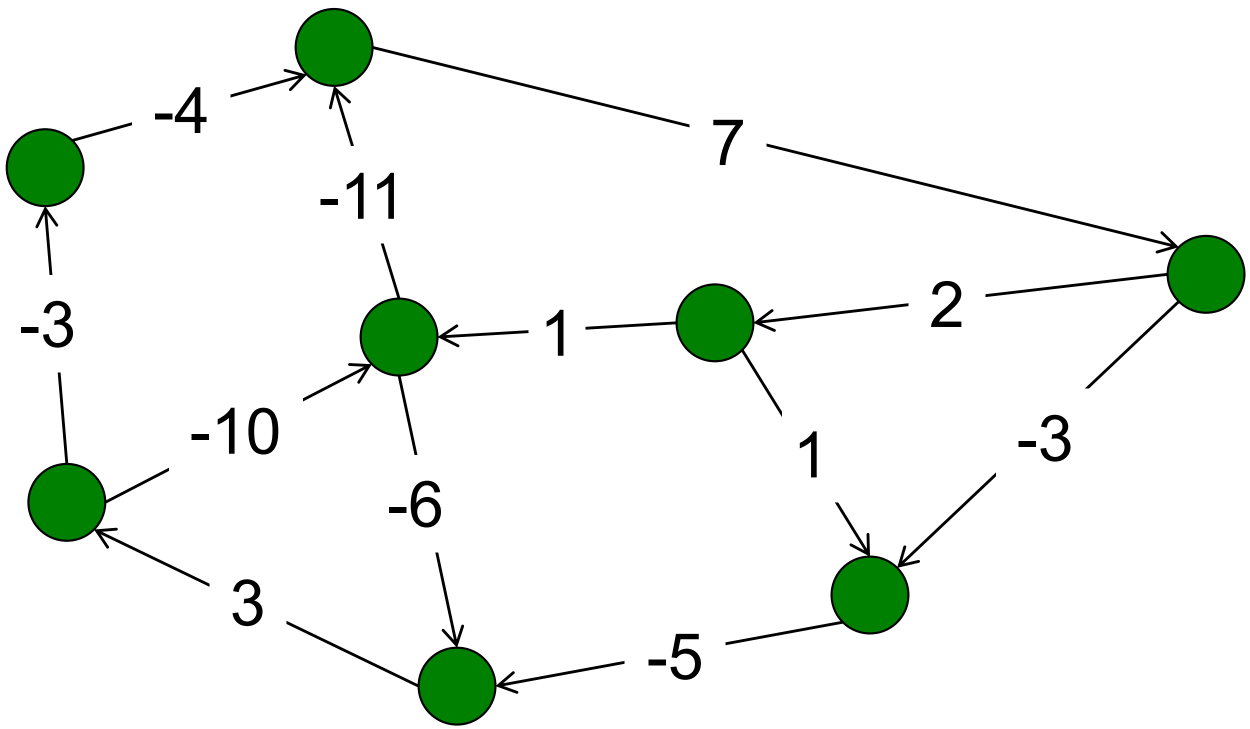
Lazy Prize Collecting

Dynamic Programming

P[v, k] = maximum prize that you can collect starting at v and taking *exactly* k steps.

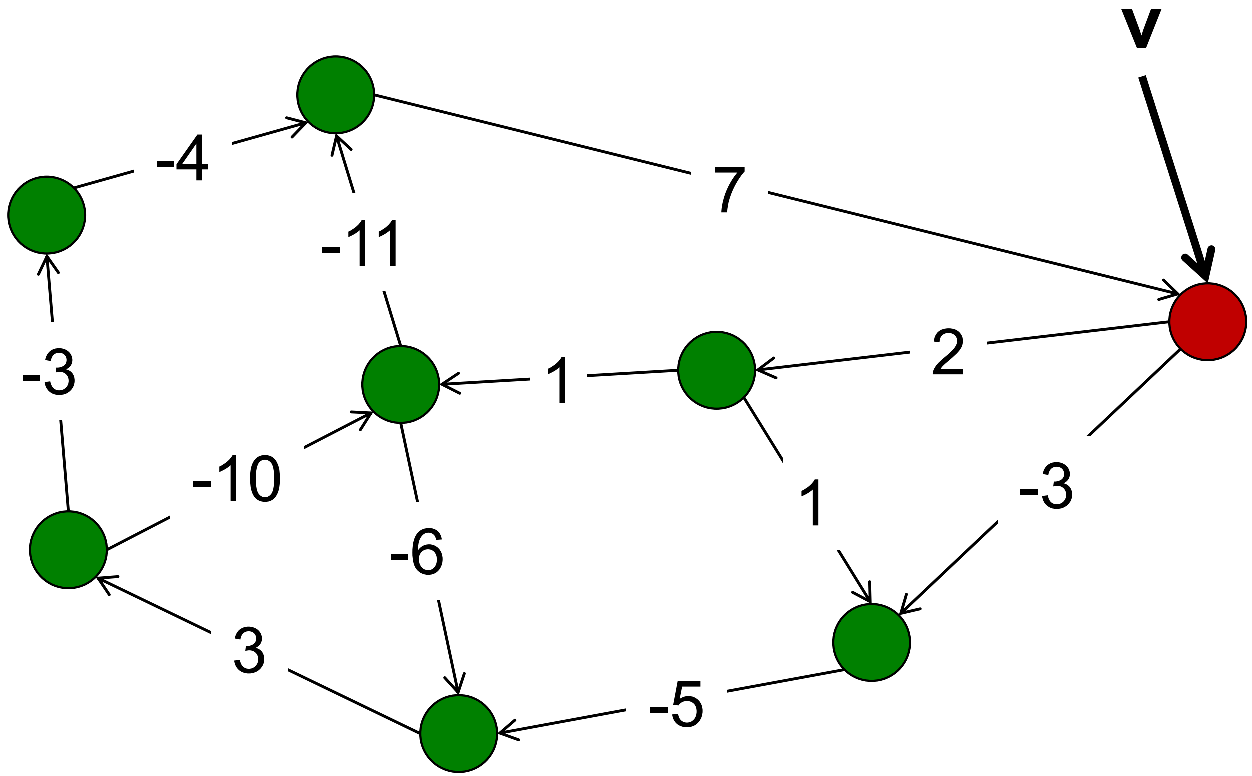
<u>Modified subproblem:</u>
Leads to better optimal substructure.

Often, useful to solve modified problem.



P(v, 0) = ??

1. 0
2. 2
3. -3
4. 4
5. 5



ARCHIPELAGO

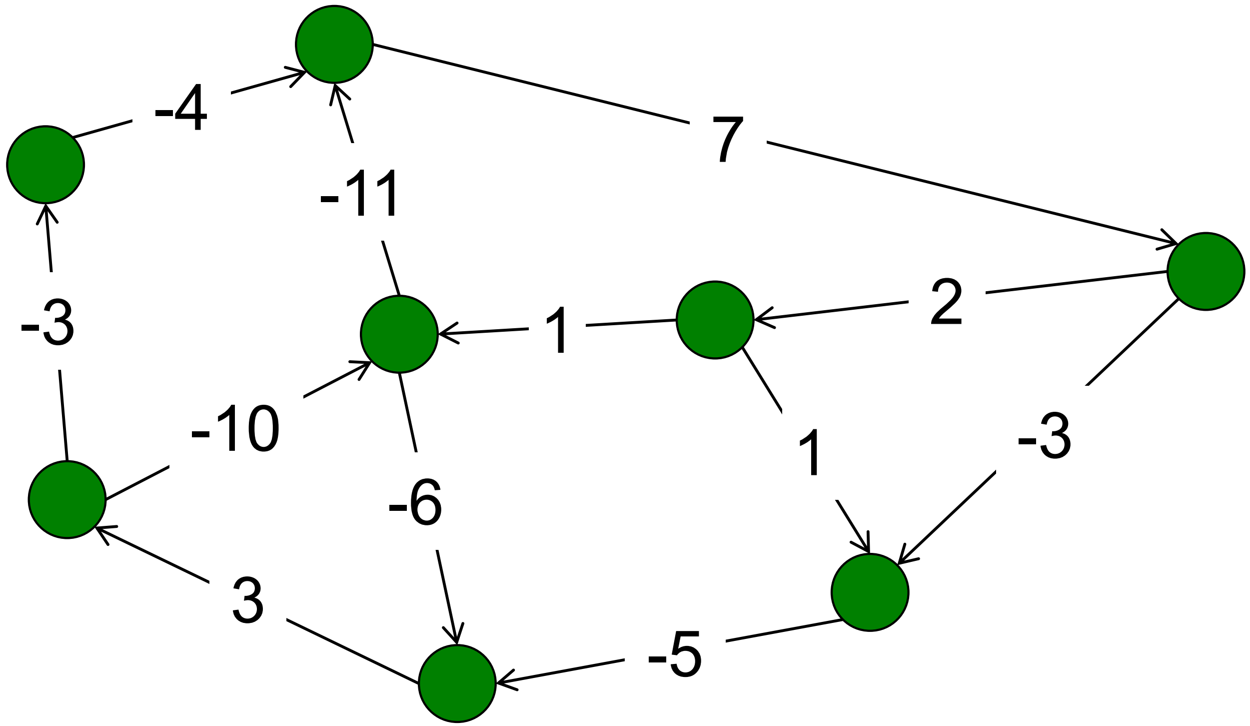
is open

Lazy Prize Collecting

Dynamic Programming

P[v, k] = maximum prize that you can collect starting at v and taking exactly k steps.

P[v, 0] = 0



Lazy Prize Collecting

Dynamic Programming

$P[v, k]$ = maximum prize that you can collect starting at v and taking *exactly* k steps.

<u>Solve P[v,k] using subproblems:</u>

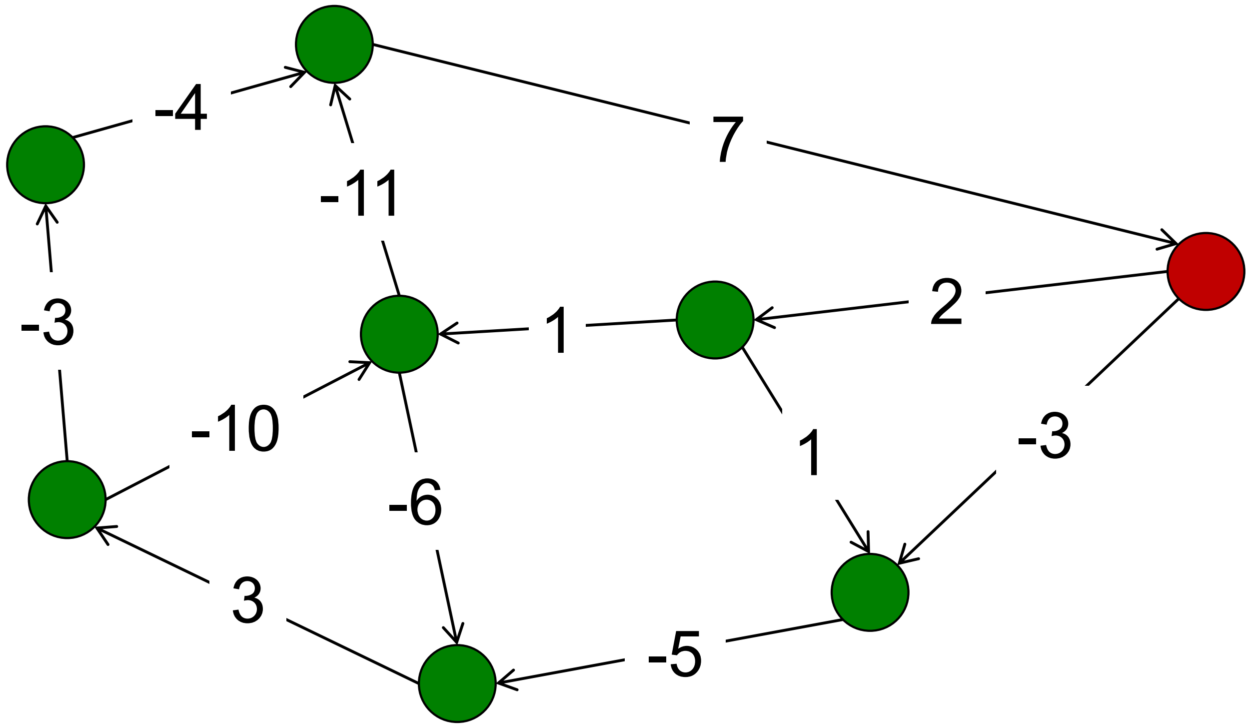
$$P[v, k] = \text{MAX}\{\ P[w_1, k-1] + w(v, w_1),\ P[w_2, k-1] + w(v, w_2),\ P[w_3, k-1] + w(v, w_3), \ldots\ \}$$

where $v.\text{nbrList}() = \{w_1, w_2, w_3, \ldots\}$

Lazy Prize Collecting

Dynamic Programming

P[v, 1] = max(0+2, 0-3) = 2

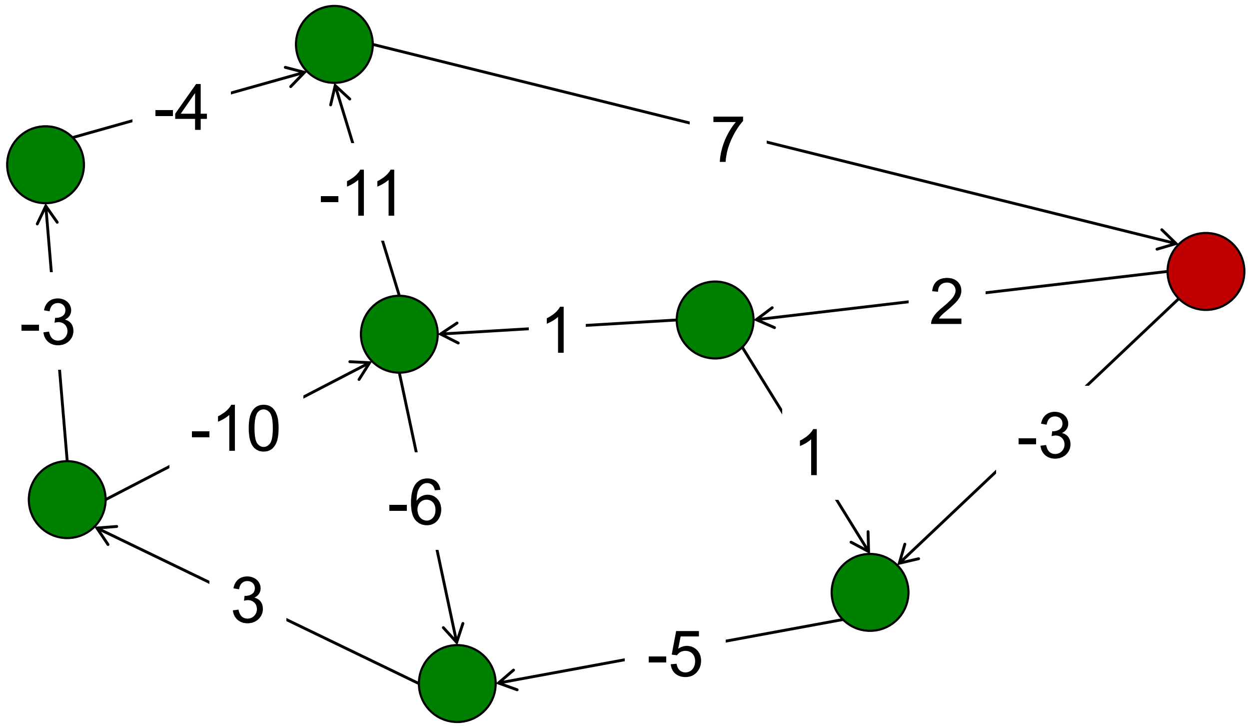


Lazy Prize Collecting

Dynamic Programming

$P[v, 1] = \max(0+2, 0-3) = 2$

$P[v, 2] = \max(1+2, -5-3) = 3$



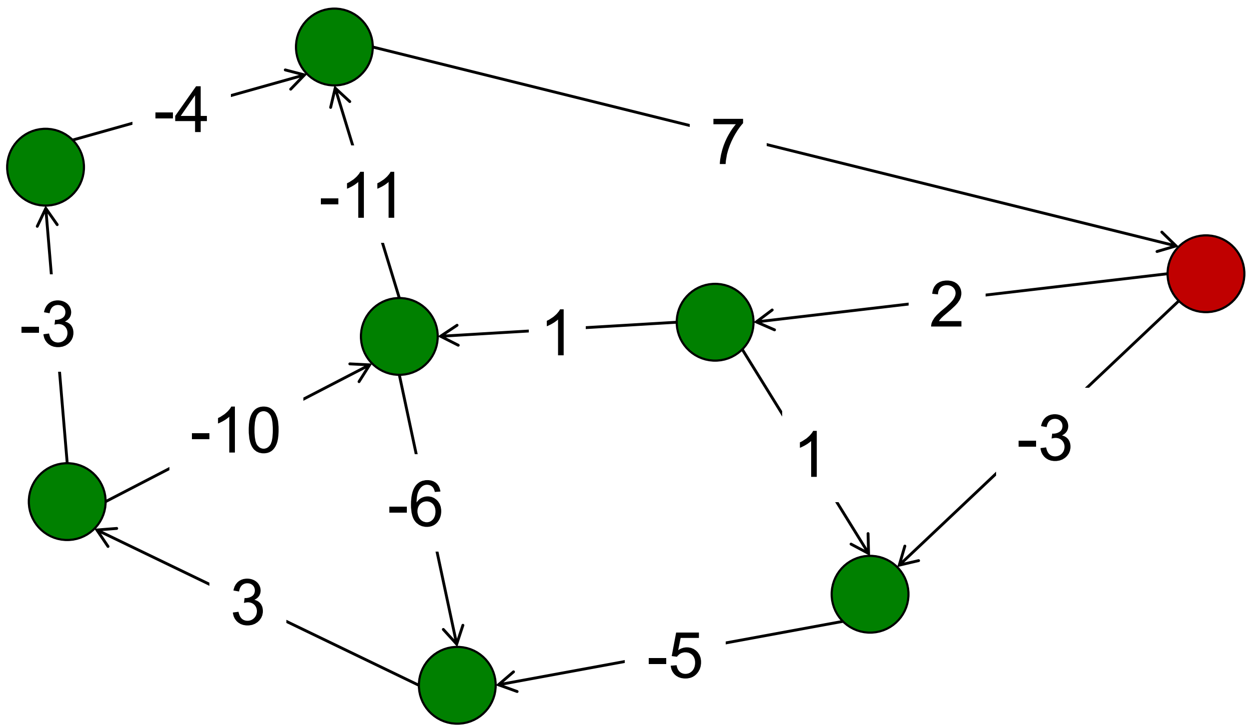
Lazy Prize Collecting

Dynamic Programming

$P[v, 1] = \max(0+2, 0-3) = 2$

$P[v, 2] = \max(1+2, -5-3) = 3$

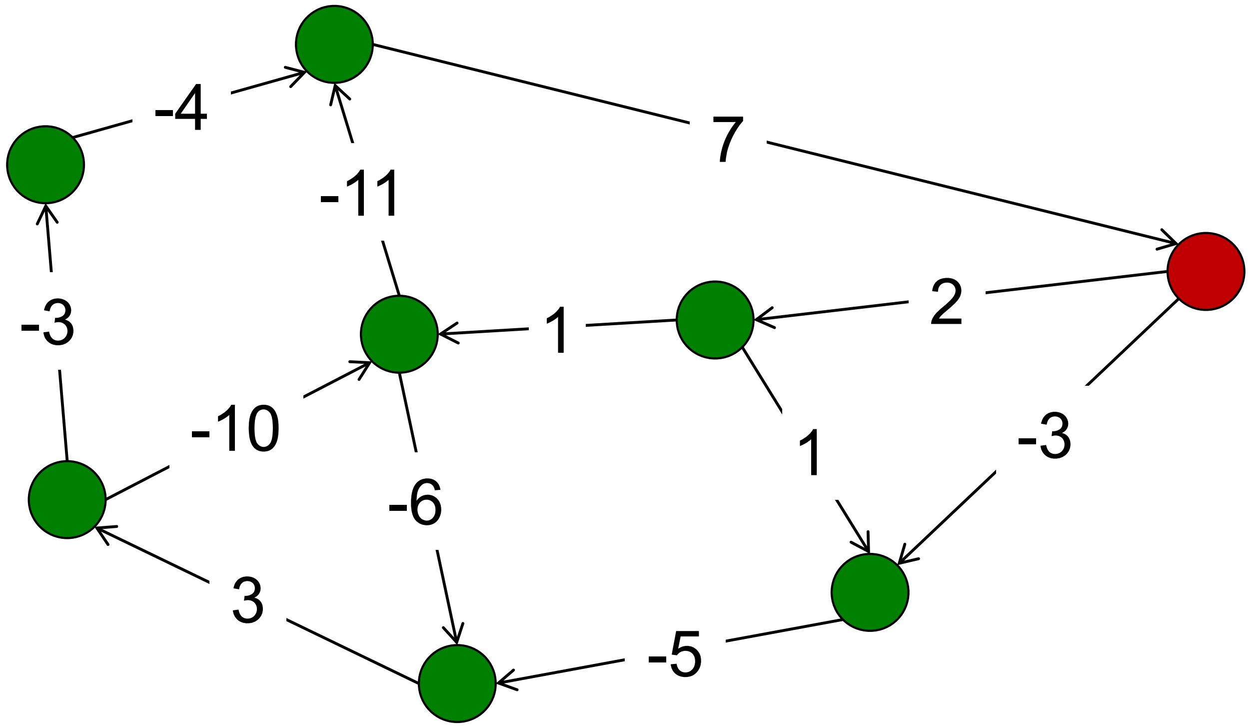
$P[v, 3] = \max(-4+2, -2-3) = -2$



Lazy Prize Collecting

Dynamic Programming

When is it worth crossing a negative edge?



Dynamic Programming

Table view: P[k, v]

k	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	17	22	14	19	8	4	9	12	15	7
2	15	12	13	13	7					
3										
4										
5										
6										
7										
8										
9										
10										
11										

```
int LazyPrizeCollecting(V, E, kMax) {

    int[][] P = new int[V.length][kMax+1]; // create memo table P
    for (int i=0; i<V.length; i++) // initialize P to zero
       for (int j=0; j<kMax+1; j++)
              P[i][j] = 0;

    for (int k=1; k<kMax+1; k++) { // Solve for every value of k
       for (int v = 0; v<V.length; v++) { // For every node…
              int max = -INFTY;
              // …find max prize in next step
              for (int w : V[v].nbrList()) {
                     if (P[w,k-1] + E[v,w] > max)
                            max = P[w,k-1] + E[v,w];
              }
              P[v, k] = max;
       }
    }
    return maxEntry(P); // returns largest entry in P
}
```

Lazy Prize Collecting

Dynamic Programming

P[v, k] = maximum prize that you can collect starting at v and taking exactly k steps.

Total Cost:

Two factors:

– Number of subproblems: kV

– Cost to solve each subproblem: |v.nbrList|

Total: $O(kV^2)$

Dynamic Programming

Table view: P[k, v]

k	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	17	22	14	19	8	4	9	12	15	7
2	15	12	13	13	7					
3										
4										
5										
6										
7										
8										
9										
10										
11										

Lazy Prize Collecting

Dynamic Programming

P[v, k] = maximum prize that you can collect starting at v and taking exactly k steps.

Total Cost:

Two factors:

– Number of rows: k

– Cost to solve all problems in a row: E

Total: O(kE)

Roadmap

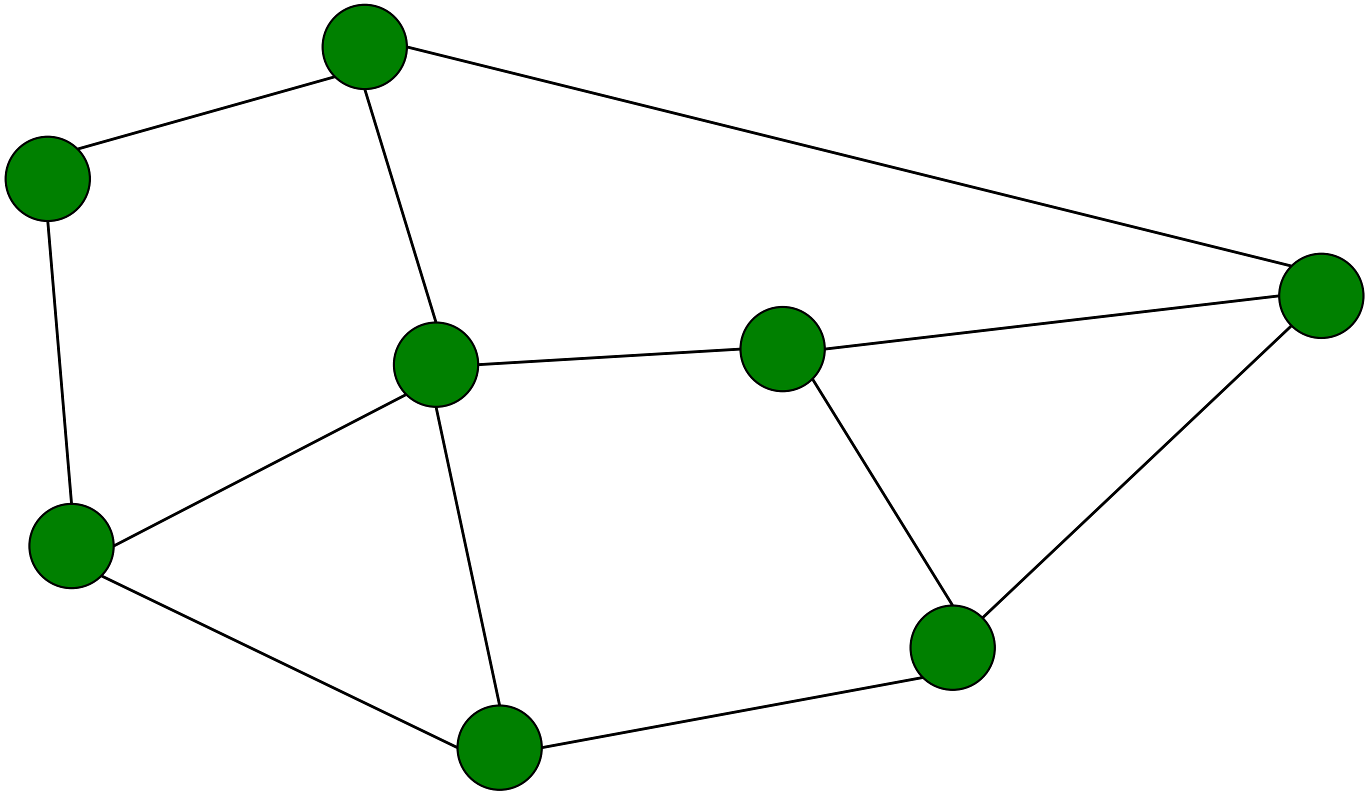
Dynamic Programming

- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths

Vertex Cover

Input:

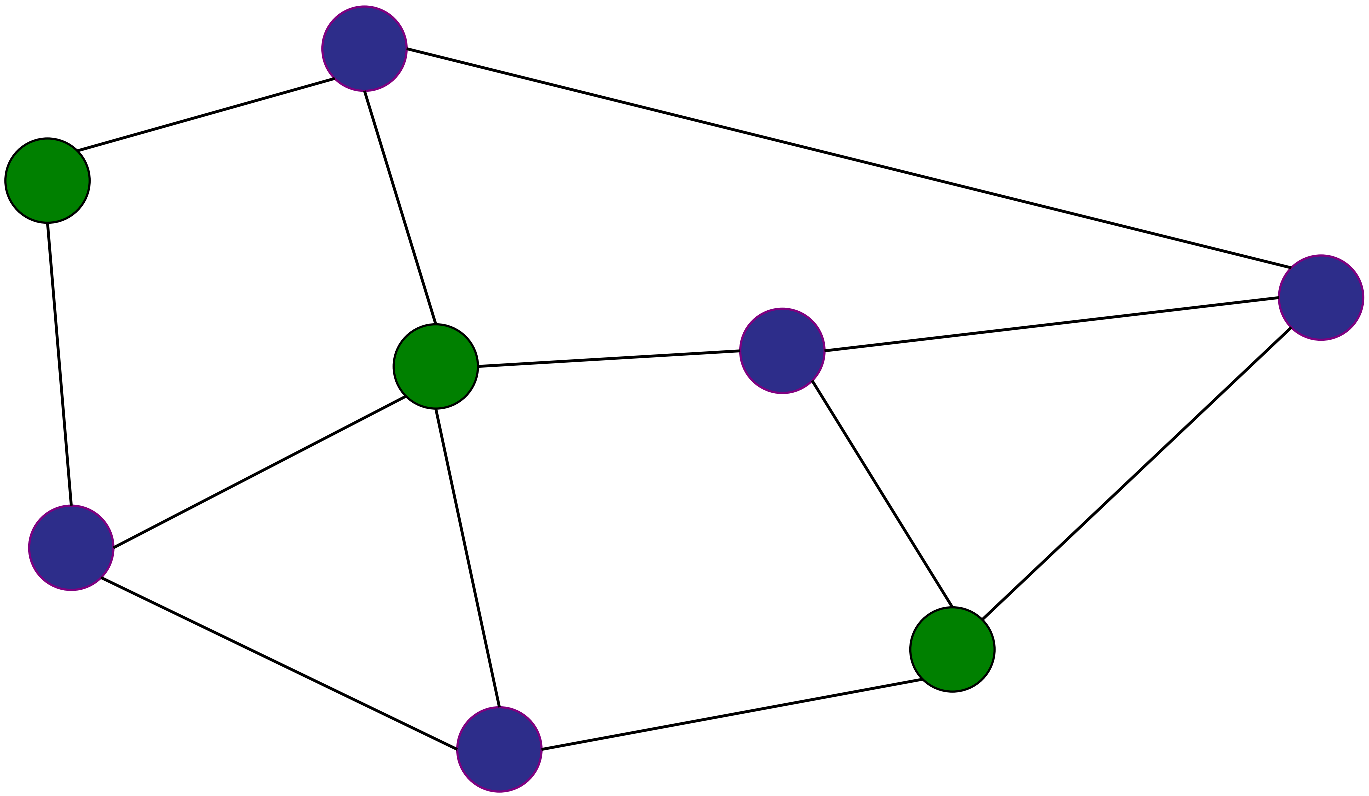
Undirected, unweighted graph G = (V,E)



Vertex Cover

Output:

Set of nodes C where every edge is adjacent to at least one node in C.



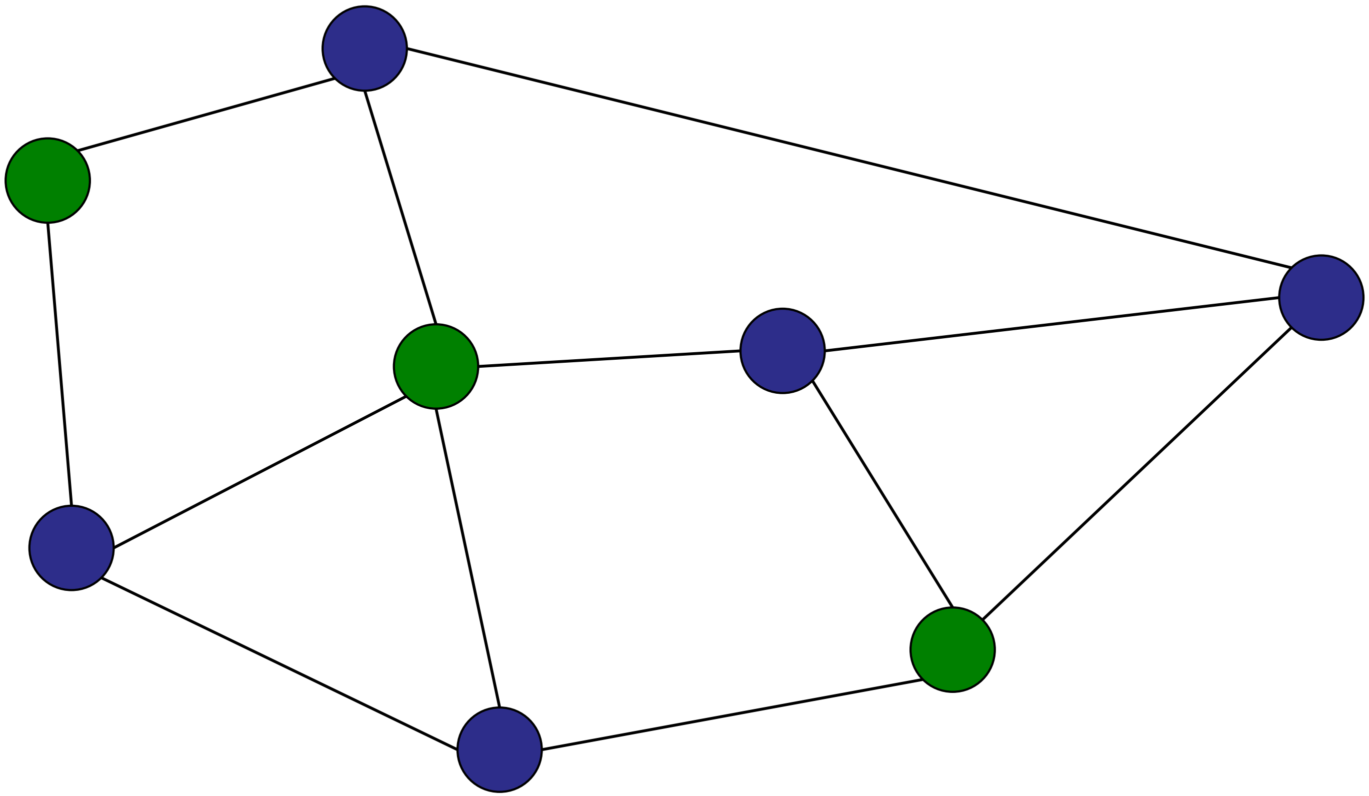
Minimum Vertex Cover

NP-complete:

No polynomial time algorithm (unless P=NP).

Easy 2-approximation (via matchings).

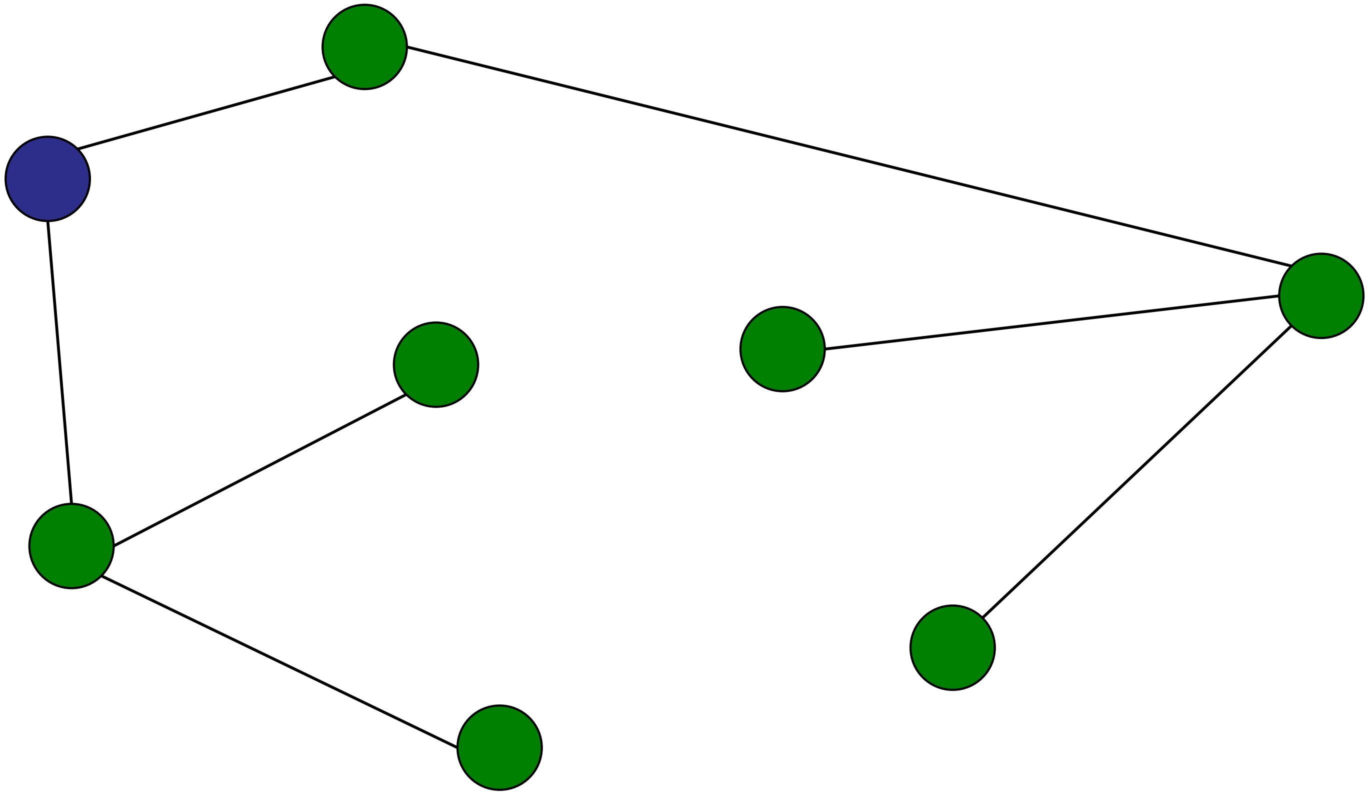
Nothing better known.



Vertex Cover on a Tree

Input:

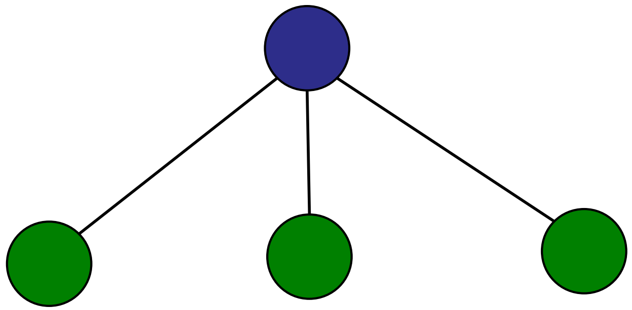
- Undirected, unweighted **tree** G = (V,E)
- Root of tree r

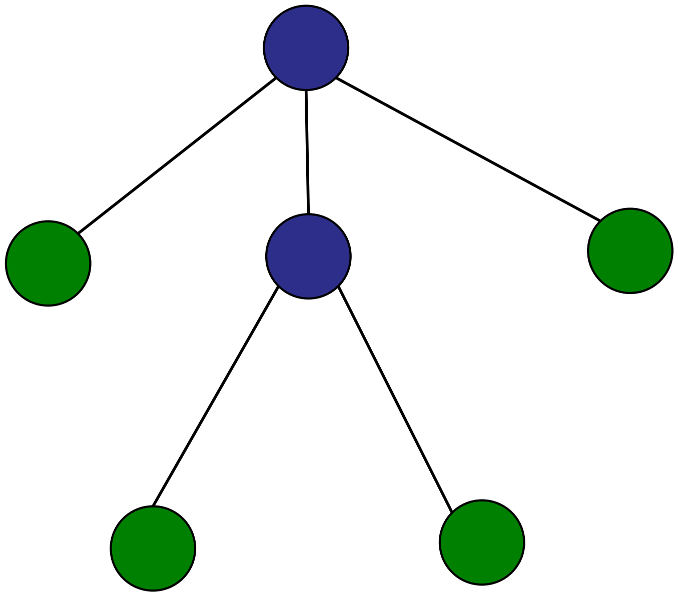


Vertex Cover on a Tree

Output:

- size of the minimum vertex cover





Dynamic Programming Recipe

Step 1: Identify optimal substructure

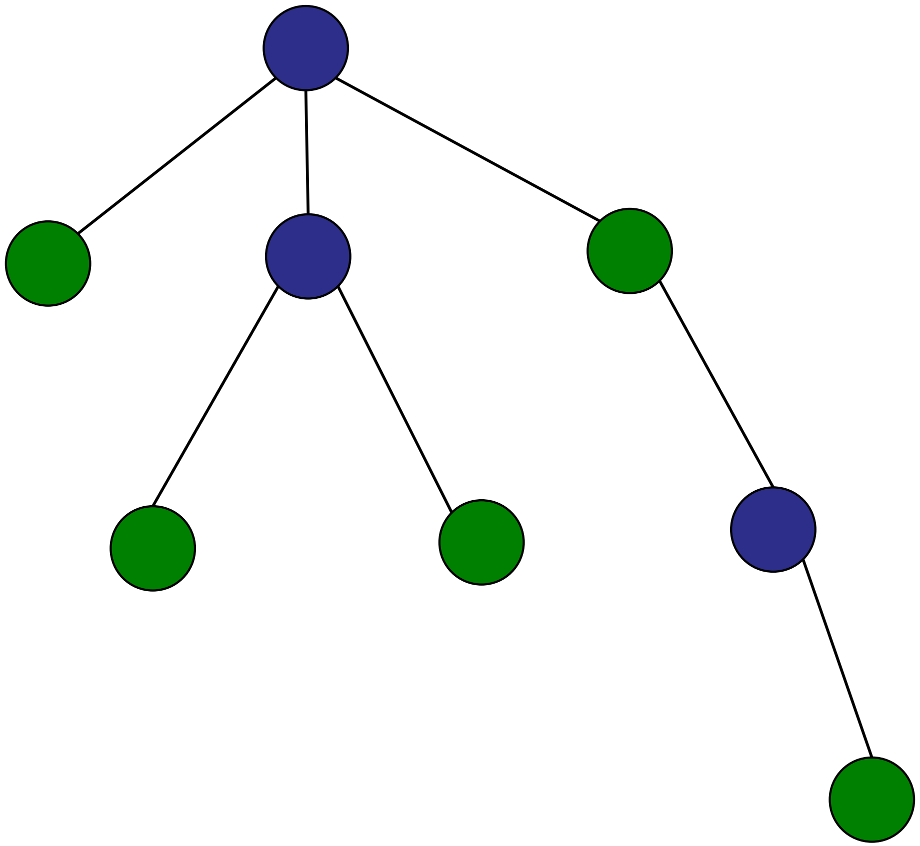
Step 2: Define sub-problems

Step 3: Solve problem using sub-problems

Step 4: Write (pseudo)code.

Vertex Cover on a Tree

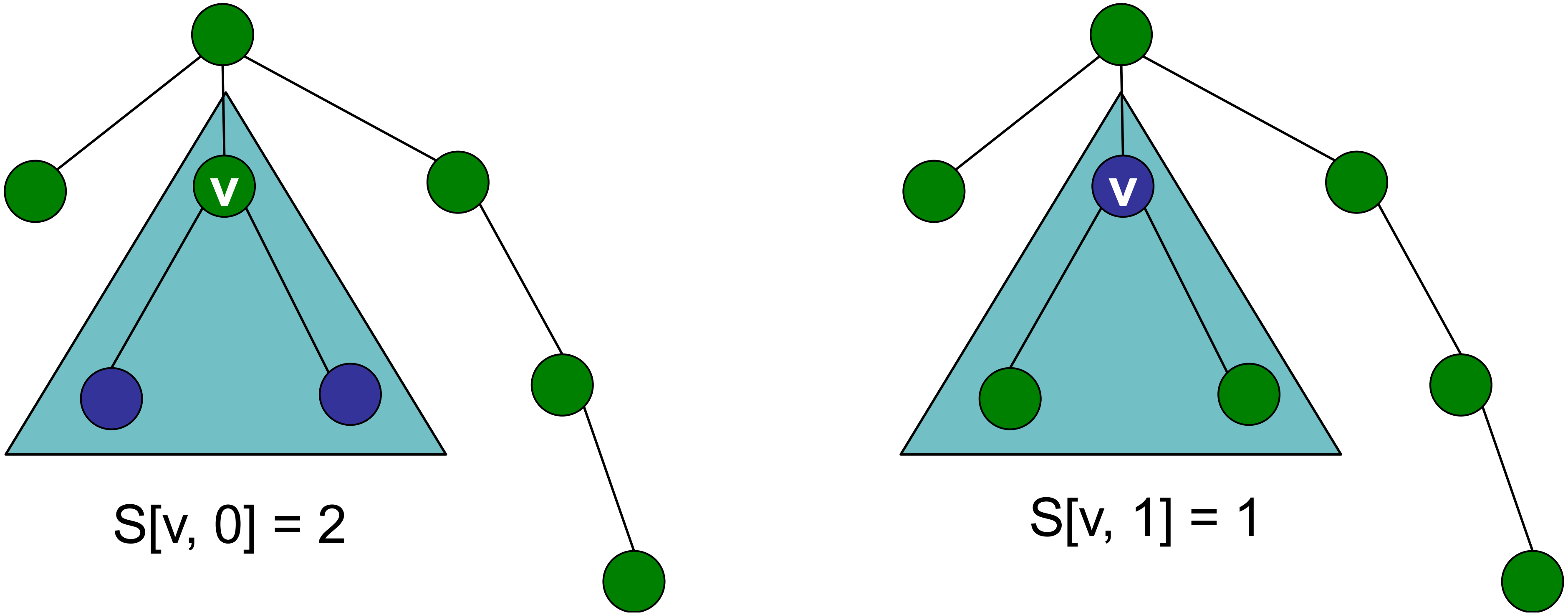
What are the subproblems?



Vertex Cover on a Tree

S[v, 0] = size of vertex cover in subtree rooted at node v, if v is NOT covered.

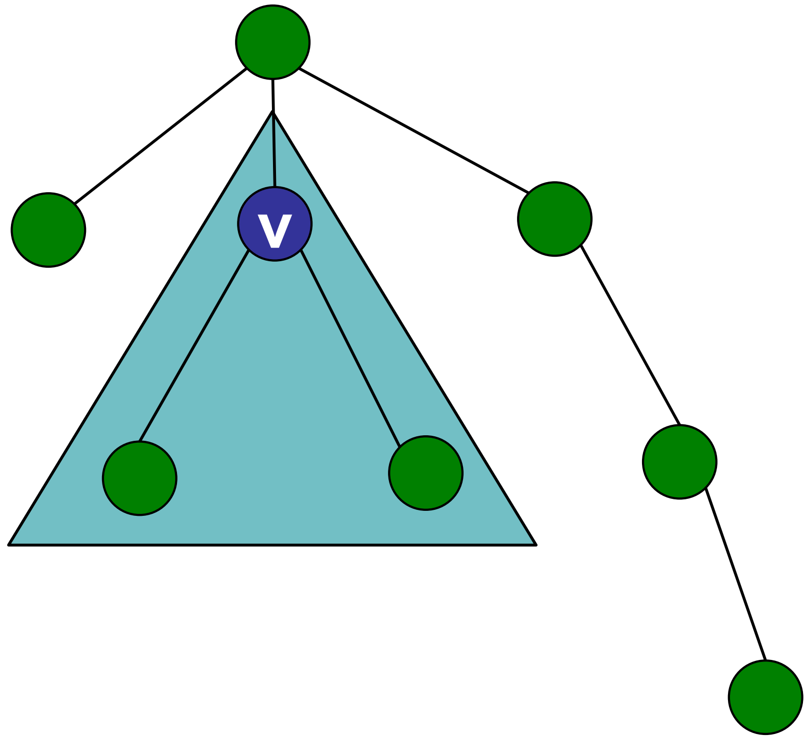
S[v, 1] = size of vertex cover in subtree rooted at node v, if v IS covered.



S[v, 0] = 2

S[v, 1] = 1

How many subproblems?



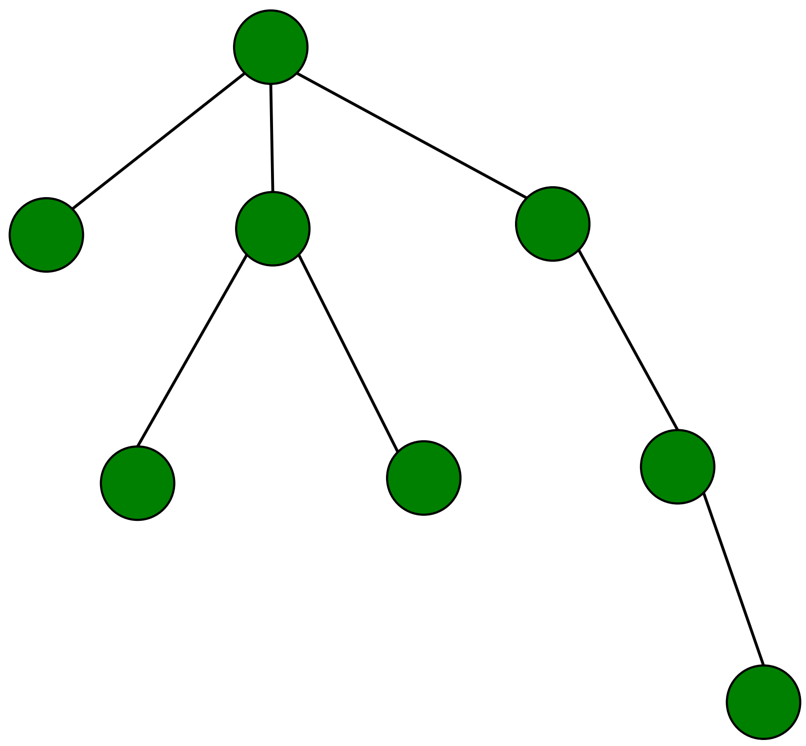
1. 2
2. V
3. 2V
4. E
5. 2E
6. VE

ARCHIPELAGO

is open

Vertex Cover on a Tree

What is the base case?



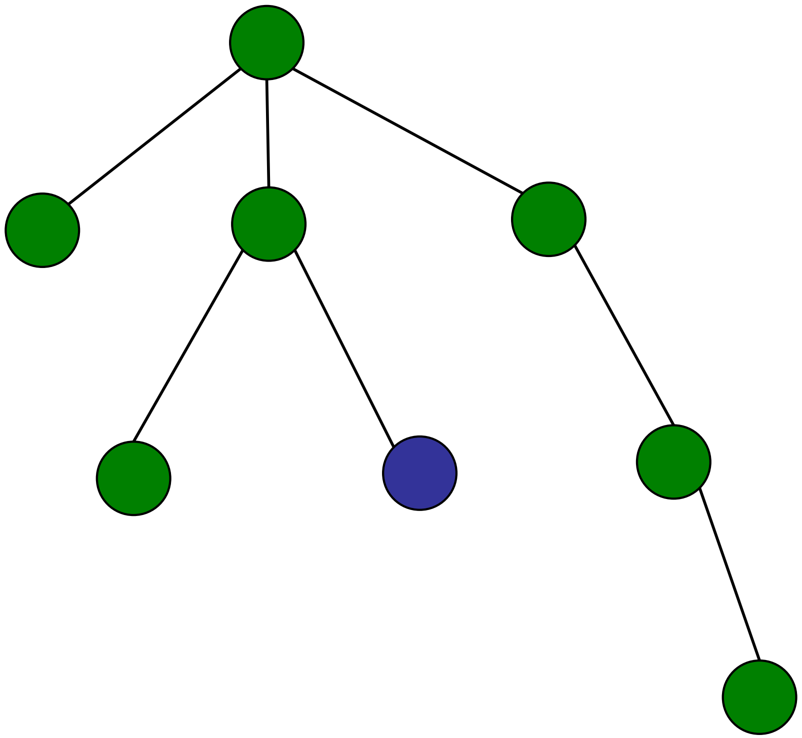
Vertex Cover on a Tree

What is the base case?

Start at the leaves!

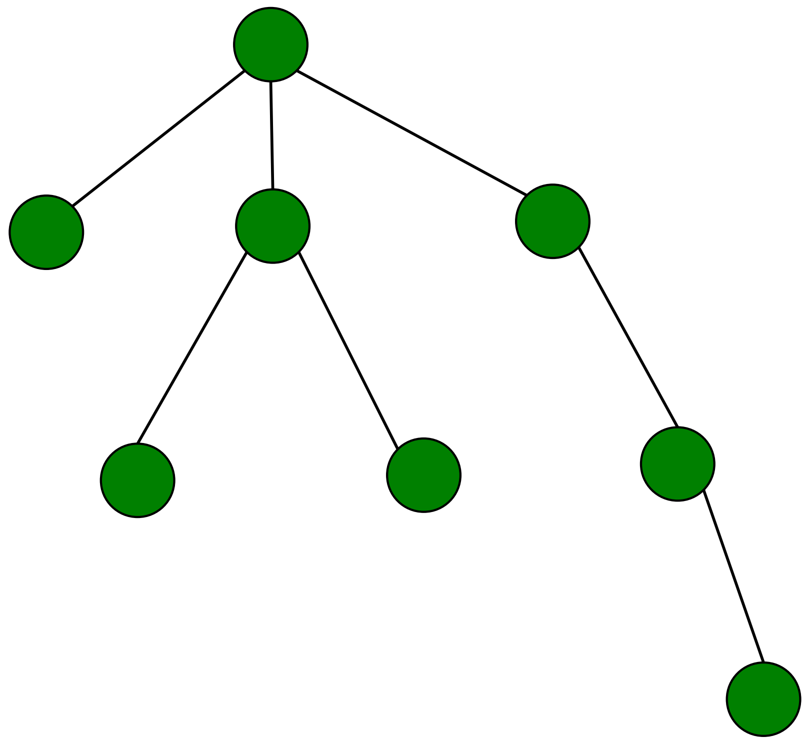
S[leaf, 0] = 0

S[leaf, 1] = 1



Vertex Cover on a Tree

How do we calculate S[v, 0]?

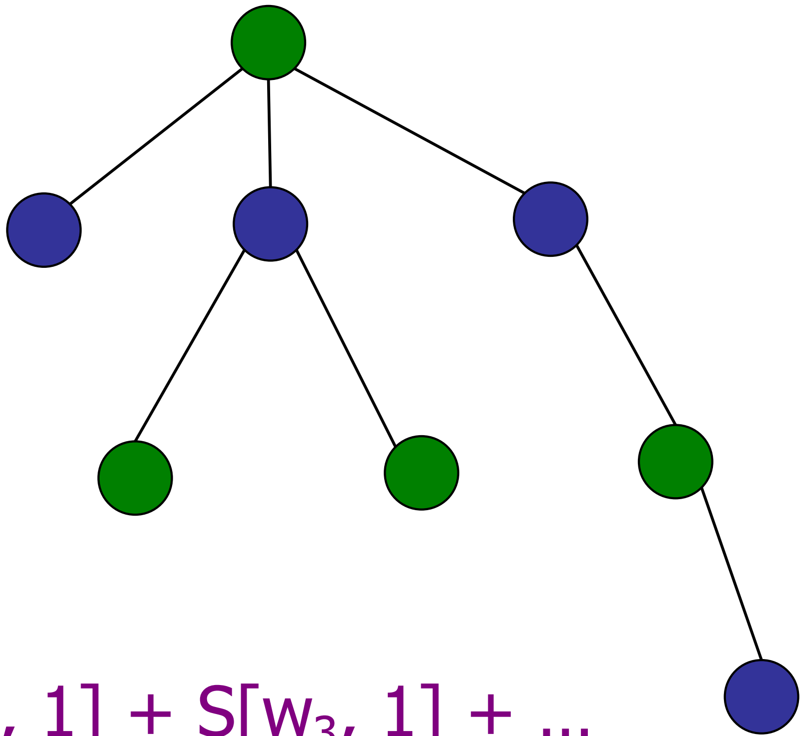


Vertex Cover on a Tree

How do we calculate S[v, 0]?

If we do not cover v, then we need to cover all of v's children.

Remember: we have already solved the subproblems!



$S[v, 0] = S[w_1, 1] + S[w_2, 1] + S[w_3, 1] + \ldots$

$v.\text{nbrList}() = \{w_1, w_2, w_3, \ldots\}$

Vertex Cover on a Tree

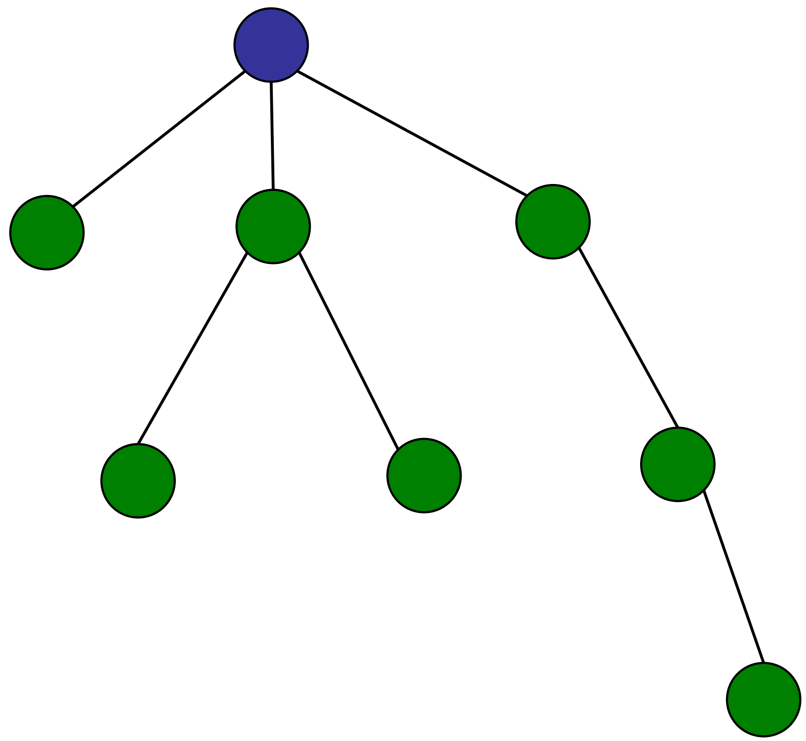
How do we calculate S[v, 1]?

We can either cover or uncover v's children.

$W_1 = \min(S[w_1, 0], S[w_1, 1])$

$W_2 = \min(S[w_2, 0], S[w_2, 1])$

$W_3 = \min(S[w_3, 0], S[w_3, 1])$



$S[v, 1] = 1 + W_1 + W_2 + W_3 + \ldots$

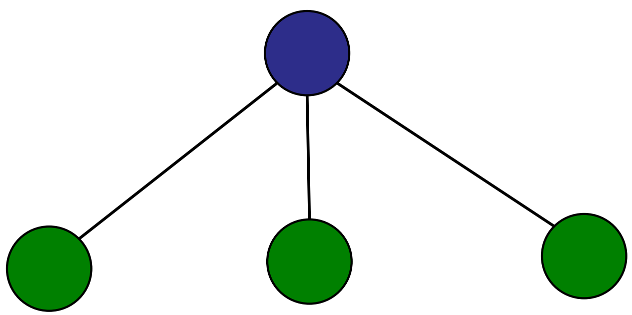
$v.\text{nbrList}() = \{w_1, w_2, w_3, \ldots\}$

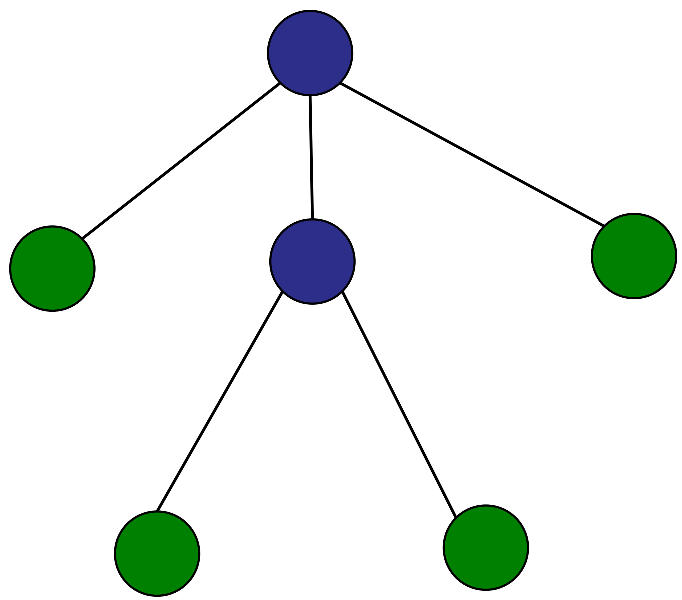
```
int treeVertexCover(V){//Assume tree is ordered from root-to-leaf

    int[][] S = new int[V.length][2]; // create memo table S

    for (int v=V.length-1; v>=0; v--){//From the leaf to the root
       if (v.childList().size()==0) { // If v is a leaf…
              S[v][0] = 0;
              S[v][1] = 1;
       }
       else{  // Calculate S from v’s children.
              int S[v][0] = 0;
              int S[v][1] = 1;
              for (int w : V[v].childList()) {
                     S[v][0] += S[w][1];
                     S[v][1] += Math.min(S[w][0], S[w][1]);
              }
       }
    }
    return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

Vertex Cover on a Tree

Running time:

- 2V sub-problems
- O(V) time to solve all sub-problems.
  - Each edge explored once.
  - Each sub-problem involves exploring children edges.





Roadmap

Dynamic Programming

- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- – Example: All-Pairs Shortest Paths

All Pairs Shortest Path

Input:

- Directed, weighted graph $G = (V,E)$

Goal:

- Preprocess G
- Answer queries: min-distance(v, w)?

Example:

- On-line map service

All Pairs Shortest Path

Simple solution:

- Run Dijkstra’s Algorithm on every query

Cost:

- Preprocessing: 0
- Responding to q queries: O(q*E*log V)

All Pairs Shortest Path

Simple solution++:

On query(v,w):

- Run Dijkstra's Algorithm from source v
- Set dist[v,*] = ...
- Next time, on query(v, ?) don't run Dijkstra's.

Cost:

- Preprocessing: 0
- Responding to q queries: O(VE*log V)

All Pairs Shortest Path

Preprocessing solution:

On preprocessing:

- For all (v,w): calculate distance(v,w)

On query:

- Return precalculated value.

Cost:

- Preprocessing: all-pairs-shortest-paths
- Responding to q queries: O(q)

Diameter of a Graph

Input:

Undirected, weighted graph G=(V, E)

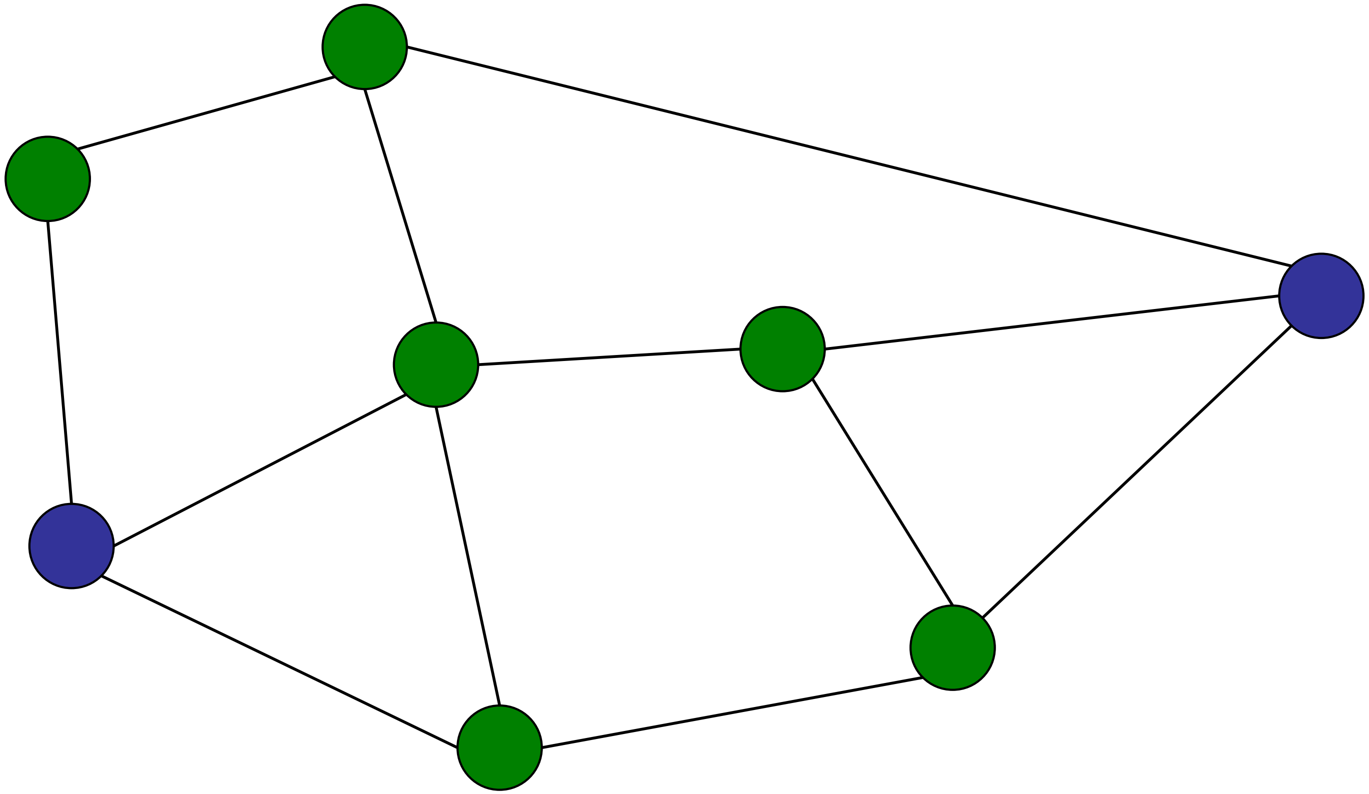
Output:

A pair of nodes (v,w) such that the shortest path from v to w is maximal.

Diameter of a Graph

Example:

diameter = 3



Diameter of a Graph

Examples:

In 1999, the diameter of the world-wide-web was (supposedly) 19.

Milgram claimed in the 1960’s that the diameter of the United Social social network was 6.

(“Six degrees of separation”)

Diameter of the Erdos collaboration graph is 23.

All Pairs Shortest Paths

Input:

- Weighted, directed graph G = (V,E)

Output:

- dist[v,w] : shortest distance from v to w, for all pairs of vertices (v,w)

All Pairs Shortest Paths

Input:

- Weighted, directed graph G = (V,E)

Output:

- dist[v,w] : shortest distance from v to w, for all pairs of vertices (v,w)

Solution:

- Run single-source-shortest paths once for every vertex v in the graph.

What is the running time of running SSSP for every vertex in V on a connected graph with positive weights?

1. O(VE)
2. O(V^2E)
3. O(V^2 + E^2)
4. O(E log V)
5. O(V^2log E)
6. O(VE log V)

ARCHIPELAGO
is open

All Pairs Shortest Paths

Solution:

- Run single-source-shortest paths once for every vertex v in the graph .
- Assume weights are all positive…

Note:

- In a sparse graph where E = O(V): $O(V^2 \log V)$
  - We don't know how to do any better.

What is the running time of running SSSP for every vertex in V on a connected graph with all identical weights?

1. $O(VE)$
2. $O(V^2E)$
3. $O(V^2 + E^2)$
4. $O(E \log V)$
5. $O(V^2 \log E)$
6. $O(VE \log V)$

ARCHIPELAGO is open

All Pairs Shortest Paths

Solution:

- Run single-source-shortest paths once for every vertex v in the graph .
- Assume weights are all positive…

Note:

- In a sparse graph where E = O(V): $O(V^2 \log V)$
  - We don't know how to do any better.
- Identical weights, use BFS: $O(V(E+V)) = O(VE)$
  - In dense graph: $O(V^3)$
  - In sparse graph: $O(V^2)$

Dynamic Programming Recipe

Step 1: Identify optimal substructure

Step 2: Define sub-problems

Step 3: Solve problem using sub-problems

Step 4: Write (pseudo)code.

Floyd-Warshall

Dynamic programming:

- Shortest paths have optimal sub-structure:
  - If P is the shortest path (u→v→w), then P contains the shortest path from (u→v) and from (v→w).

Floyd-Warshall

Dynamic programming:

- Shortest paths have optimal sub-structure:
  - If P is the shortest path (u→v→w), then P contains the shortest path from (u→v) and from (v→w).
- Shortest paths have overlapping subproblems
  - Many shortest path calculations depends on the same sub-pieces.

Floyd-Warshall

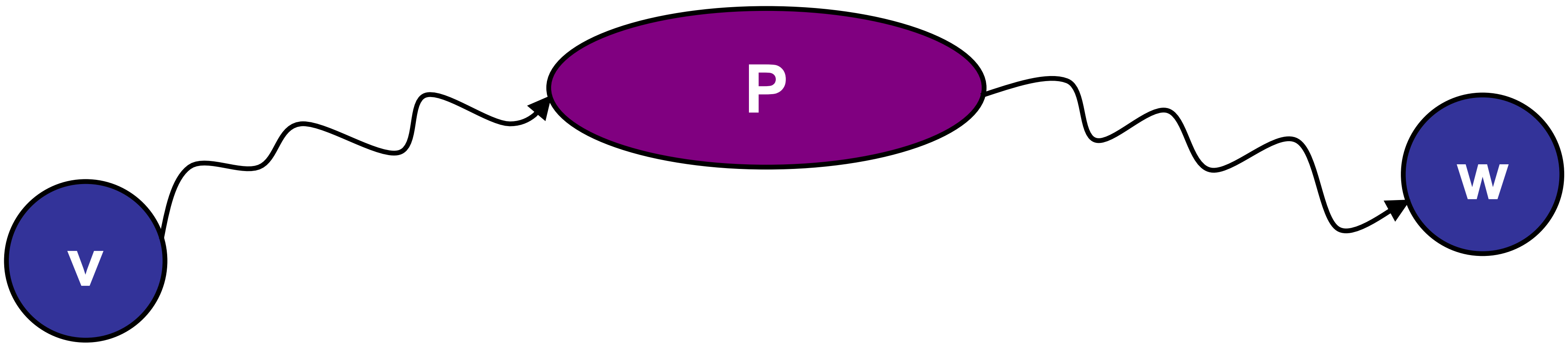
Dynamic programming:

- Shortest paths have optimal sub-structure:
  - If P is the shortest path (u→v→w), then P contains the shortest path from (u→v) and from (v→w).
- Shortest paths have overlapping subproblems
  - Many shortest path calculations depends on the same sub-pieces.
- Hard question: what are the right subproblems?

Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes in the set P.



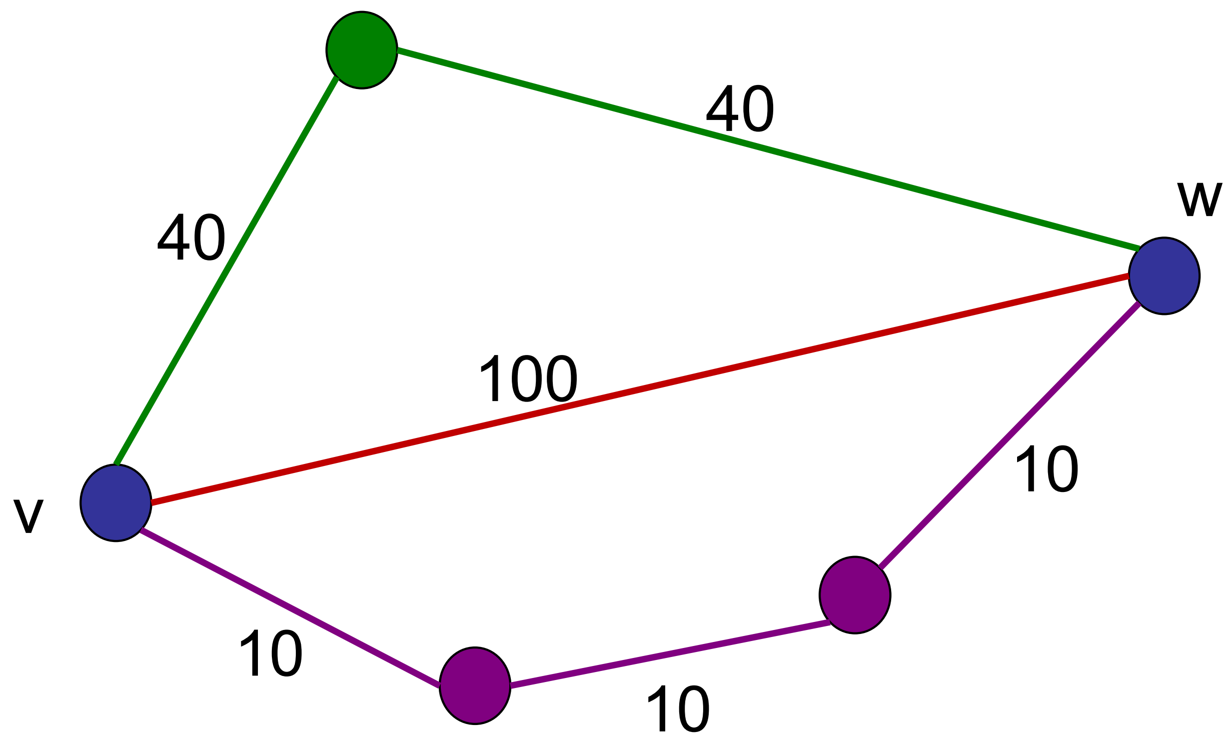
Floyd-Warshall

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes only in the set P.

P1 = no nodes (empty set)

P2 = green nodes

P3 = purple nodes



Floyd-Warshall

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes only in the set P.

P1 = no nodes (empty set)

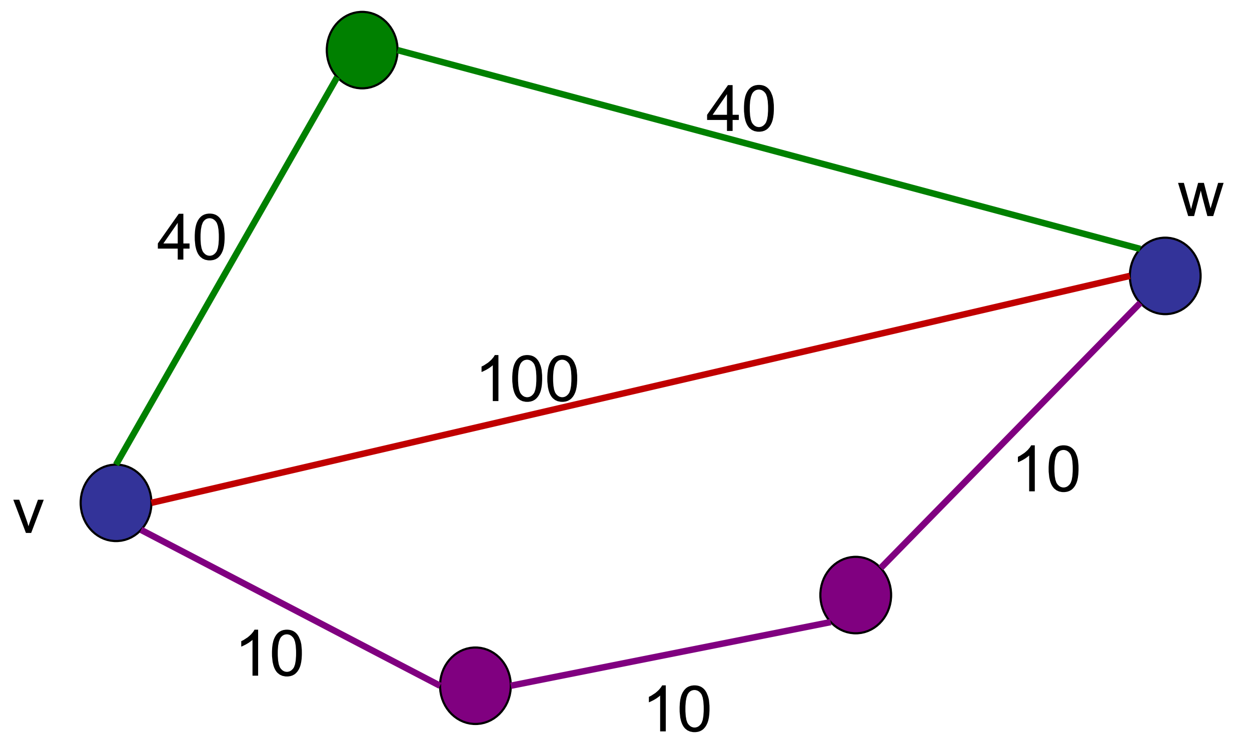
P2 = green nodes

P3 = purple nodes

S(v,w,P1) = 100

S(v,w,P2) = 80

S(v,w,P3) = 30



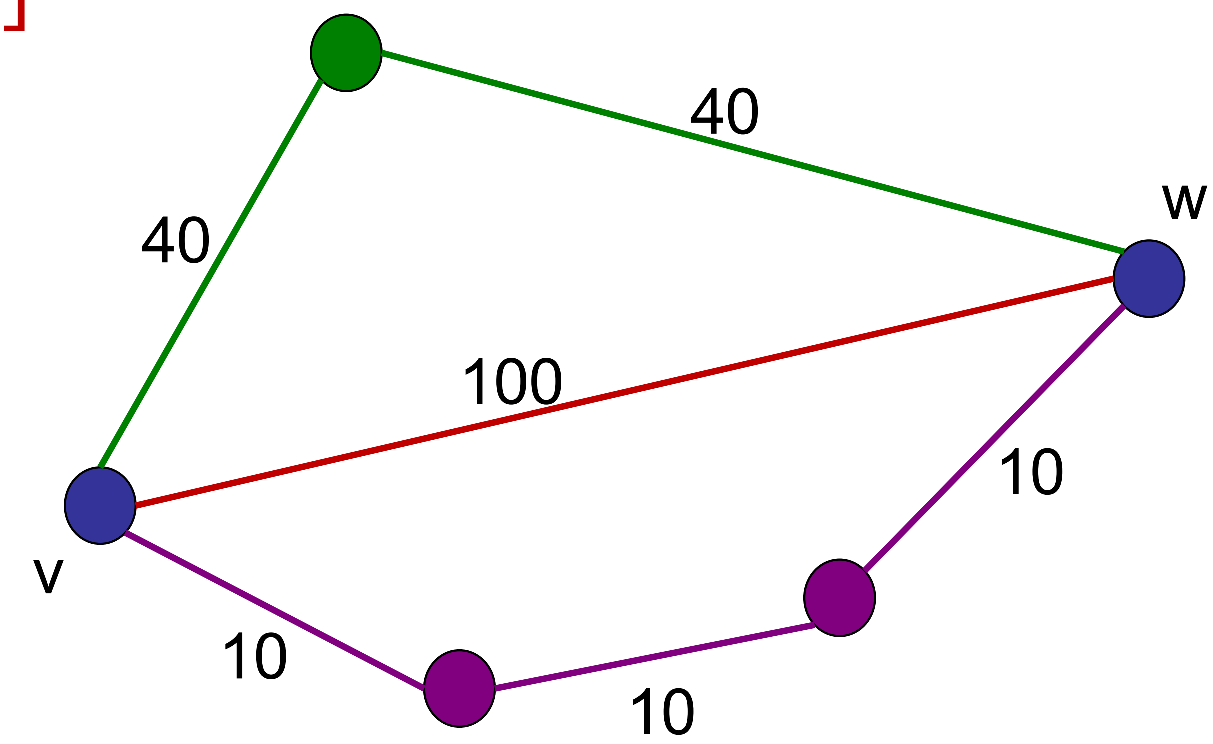
Floyd-Warshall

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes only in the set P.

Base case:

$S[v, w, \varnothing] = E[v,w]$

E[v,w] = weight of edge from v to w.



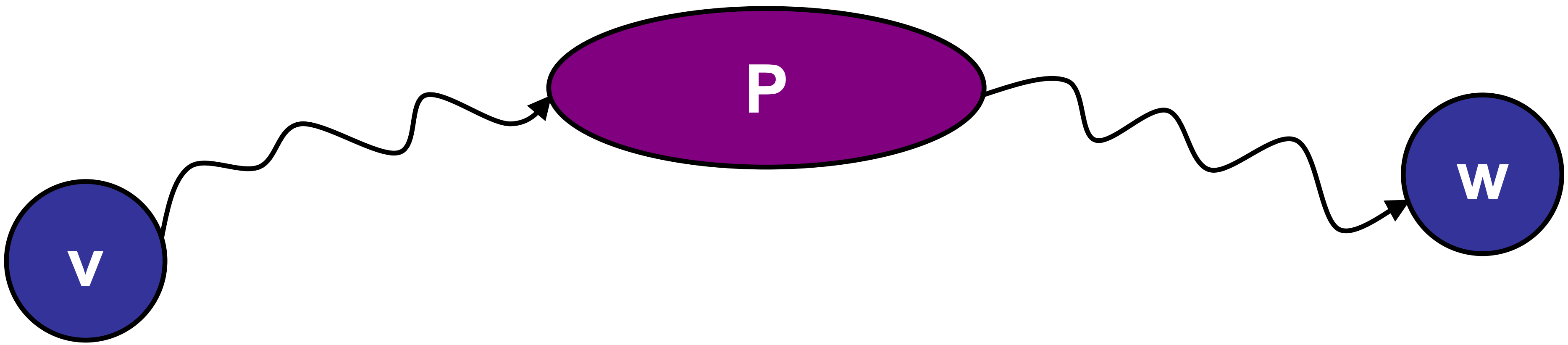
Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes in the set P.

Problem: 2^n possible sets P

➔ slow to solve *all* subproblems



Floyd-Warshall

Limit ourselves to n+1 different sets P:

$P_0 = \emptyset$

$P_1 = \{1\}$

$P_2 = \{1, 2\}$

$P_3 = \{1, 2, 3\}$

$P_4 = \{1, 2, 3, 4\}$

…

$P_n = \{1, 2, 3, 4, \ldots, n\}$

Dynamic Programming Recipe

Step 1: Identify optimal substructure

- Shortest paths are built out of shortest paths.

Step 2: Define sub-problems

- S(u,v,P) = shortest path from u to v using nodes in P.
- Consider only (n+1) sets P of increasing size.

Step 3: Solve problem using sub-problems

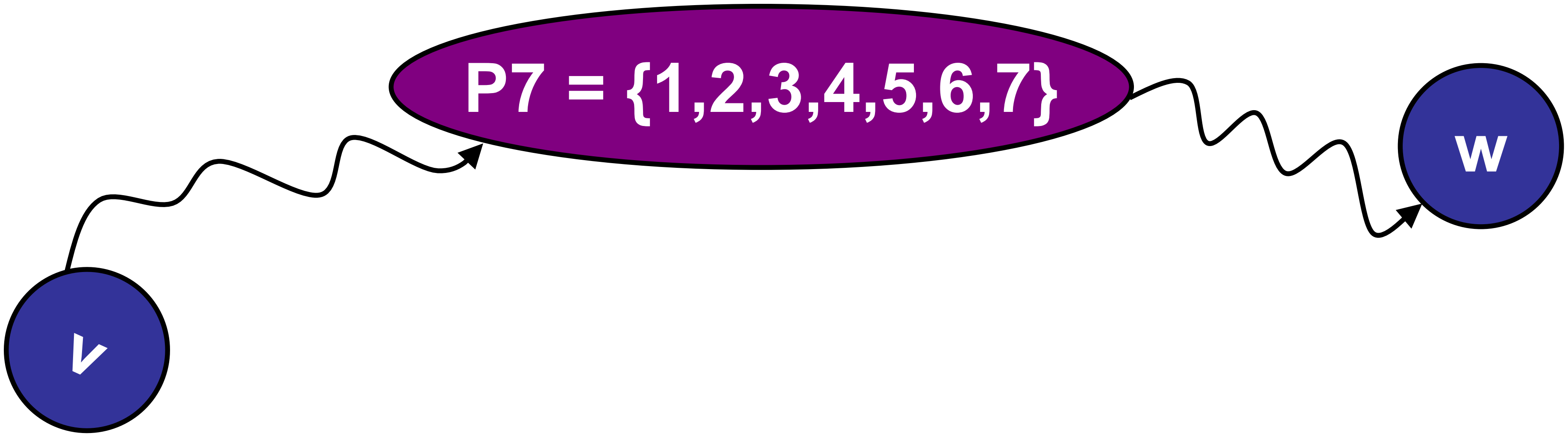
Step 4: Write (pseudo)code.

Floyd-Warshall

Use the precalculated subproblems:

Assume we have calculated $S[v,w,P_7] = 42$.

How do we calculate $S[v,w,P_8]$?



Floyd-Warshall

Use the precalculated subproblems:

Assume we have calculated $S[v,w,P_7] = 42$.

How do we calculate $S[v,w,P_8]$?

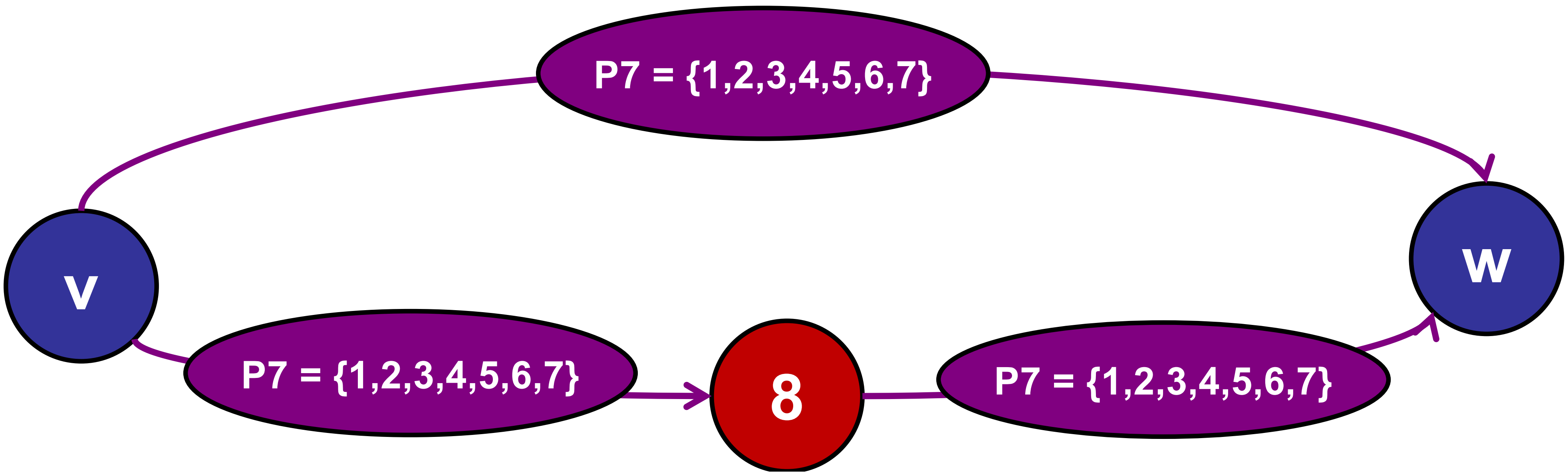
<u>Two possibilities:</u>

1. Shortest path using nodes P_8 includes node 8.
2. Shortest path using nodes P_8 does not include node 8.

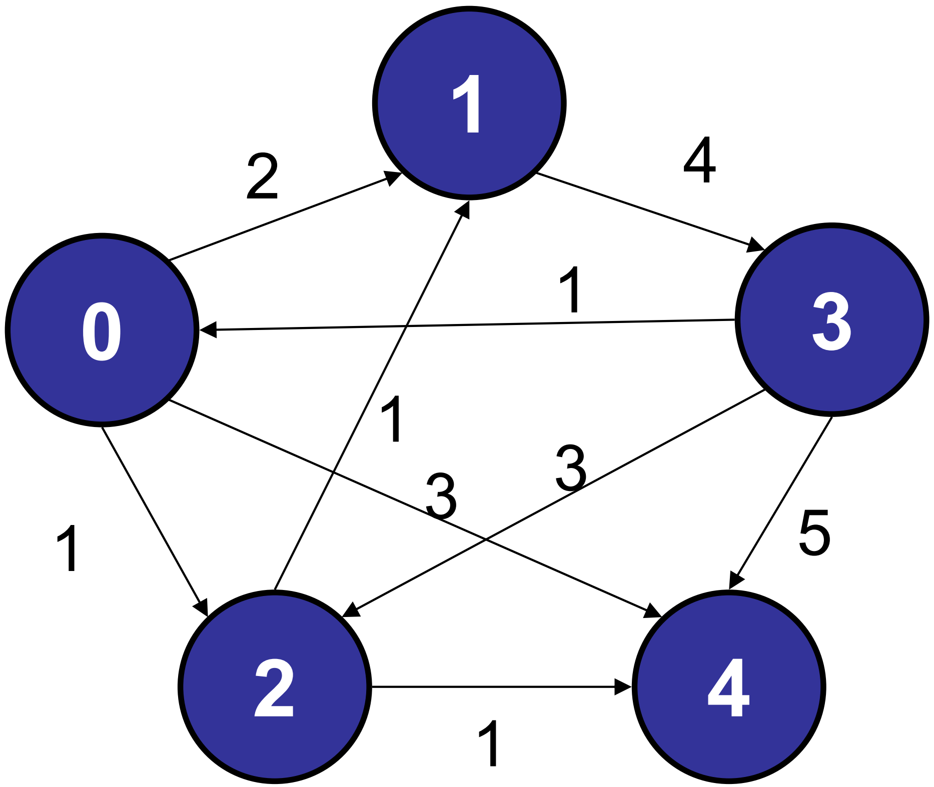
Floyd-Warshall

Use the precalculated subproblems:

$$S[v,w,P_8] = \min(\, S[v, w, P_7],\; S[v, 8, P_7] + S[8, w, P_7]$$

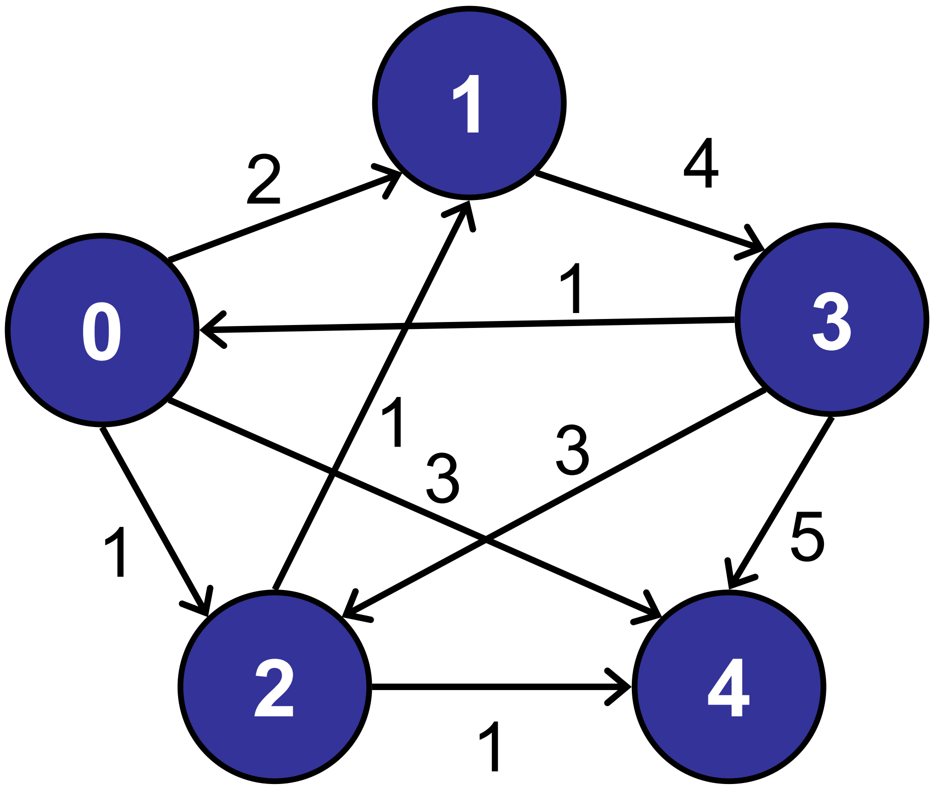


Example:

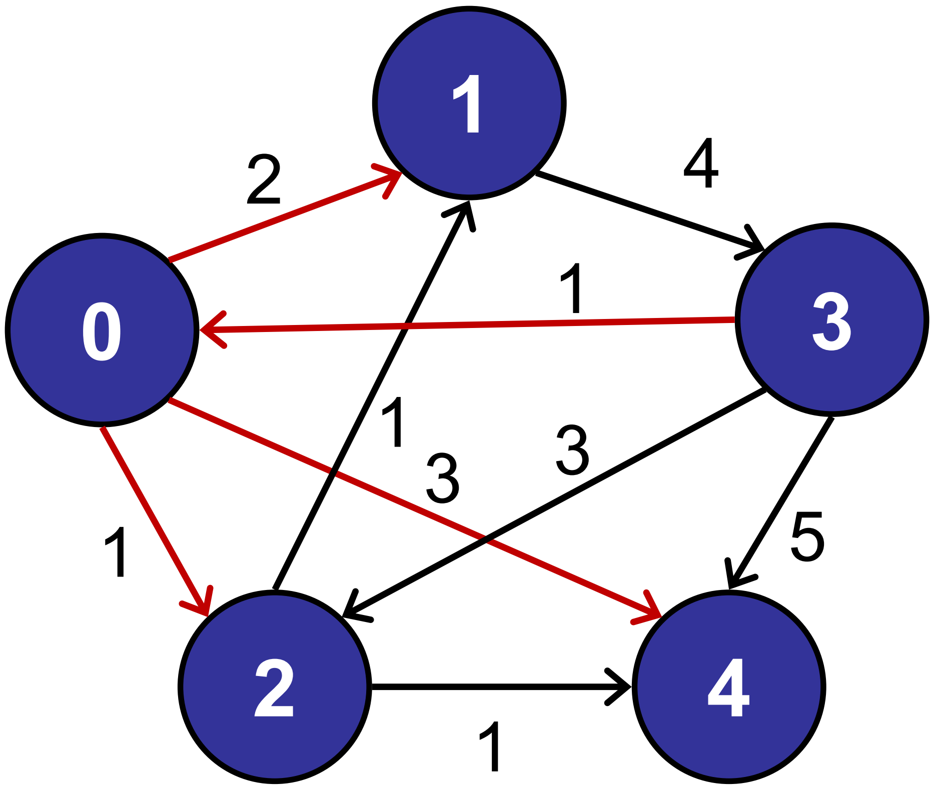


Initially:

	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0



Step: P = {0}

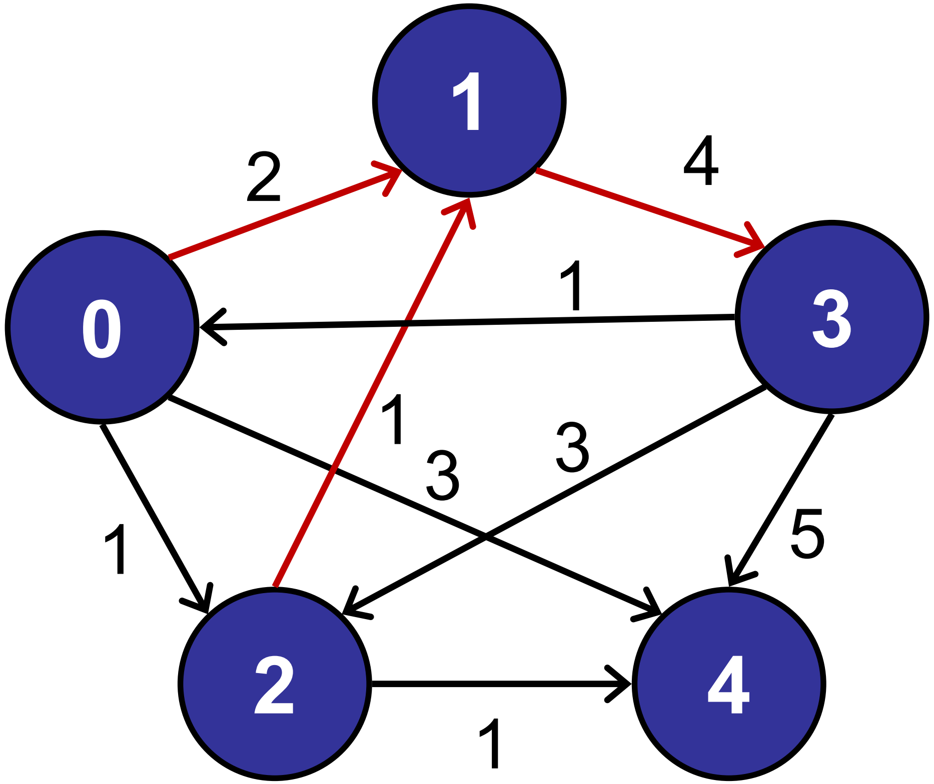


	0	1	2	3	4
0	0	**2**	**1**	∞	**3**
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	**1**	∞	3	0	5
4	∞	∞	∞	∞	0

⇓

	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	**3**	**2**	0	**4**
4	∞	∞	∞	∞	0

Step: P = {0, 1}

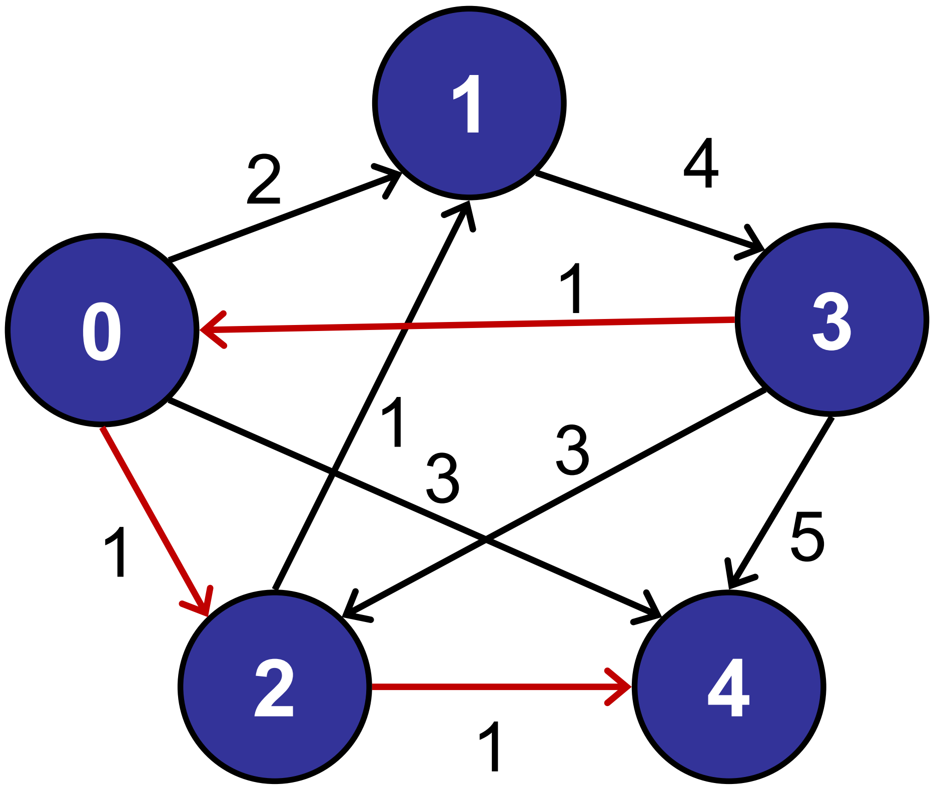


	0	1	2	3	4
0	0	**2**	1	∞	3
1	∞	0	∞	**4**	∞
2	∞	**1**	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

⇩

	0	1	2	3	4
0	0	2	1	**6**	3
1	∞	0	∞	4	∞
2	∞	1	0	**5**	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

Step: P = {0, 1, 2}

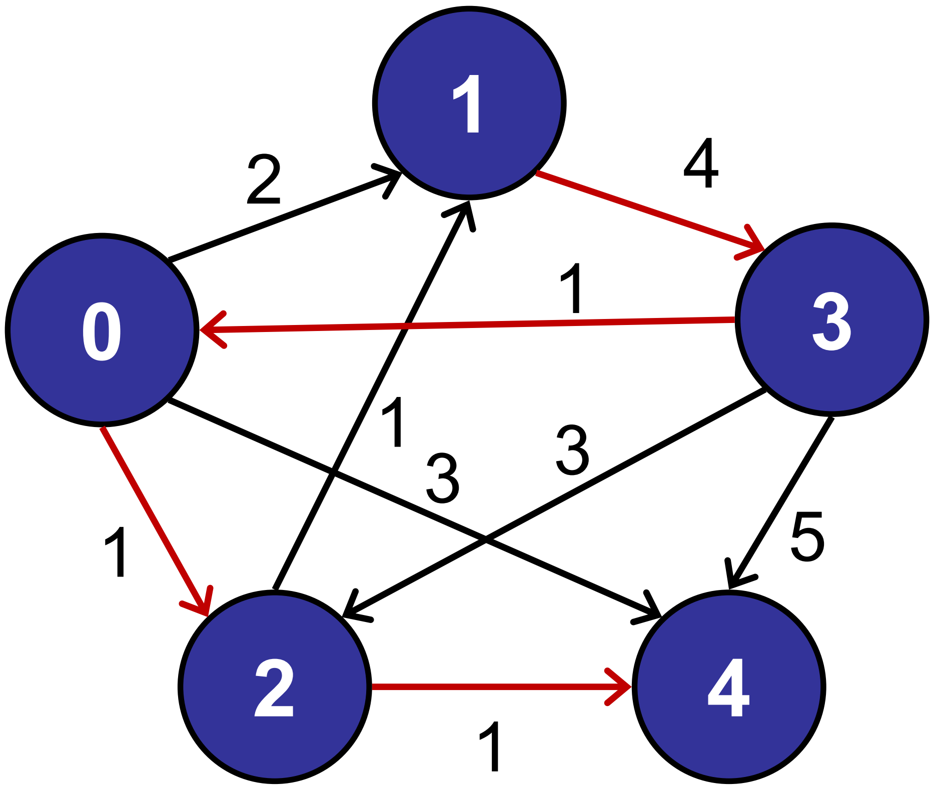


	0	1	2	3	4
0	0	2	**1**	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	**1**
3	1	3	**2**	0	4
4	∞	∞	∞	∞	0

⇩

	0	1	2	3	4
0	0	2	1	6	**2**
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**3**
4	∞	∞	∞	∞	0

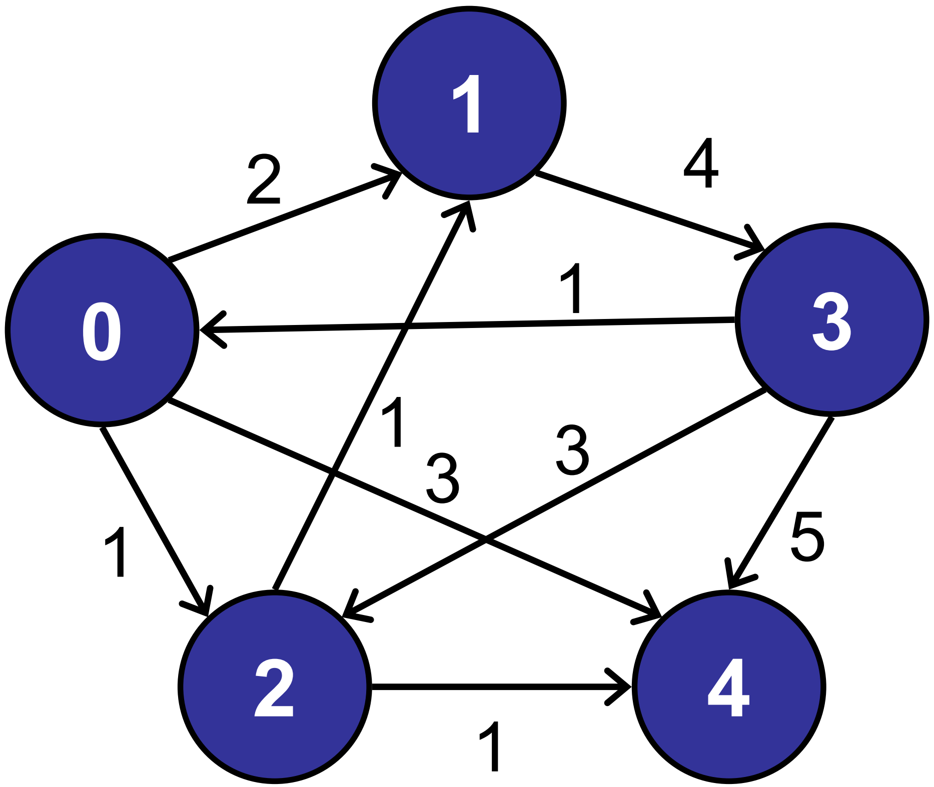
Step: P = {0, 1, 2, 3}



	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	**4**	∞
2	∞	1	0	**5**	1
3	**1**	3	**2**	0	**3**
4	∞	∞	∞	∞	0

	0	1	2	3	4
0	0	2	1	6	2
1	**5**	0	**6**	4	**7**
2	**6**	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Done: P = {0, 1, 2, 3, 4}

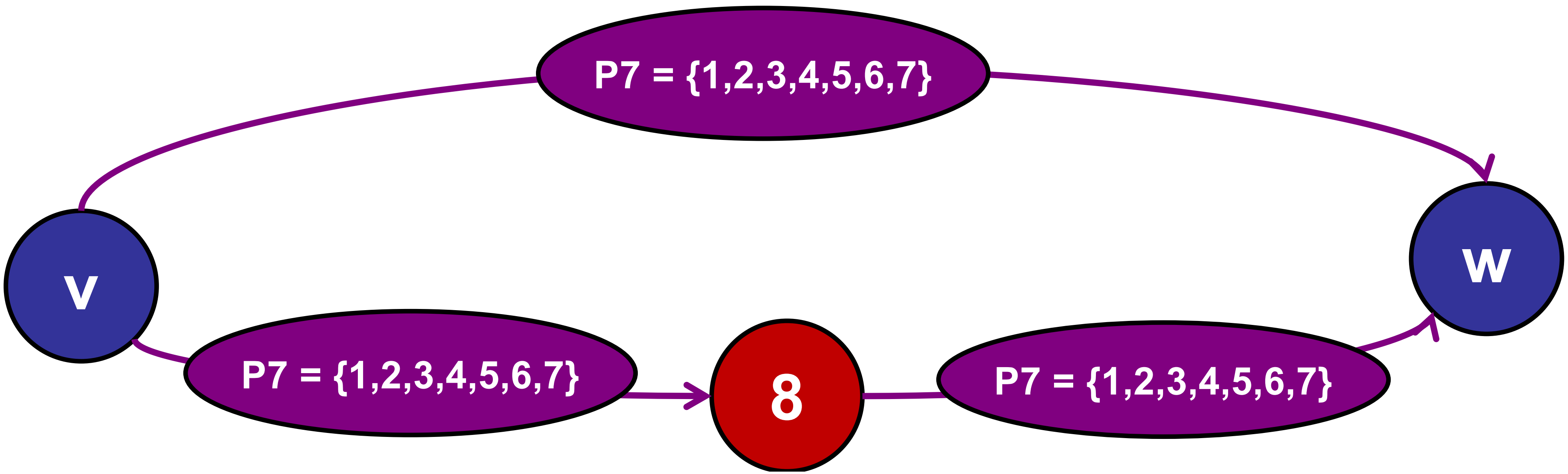


	0	1	2	3	4
0	0	2	1	6	2
1	5	0	6	4	7
2	6	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Floyd-Warshall

Use the precalculated subproblems:

$$S[v,w,P_8] = \min(\ S[v, w, P_7],\ S[v, 8, P_7] + S[8, w, P_7])$$



```
int[][] APSP(E){ // Adjacency matrix E
    int[][][] S = new int[V.length][V.length][V.length];

    // Initialize every pair of nodes for k=0
    for (int v=0; v<V.length; v++)
       for (int w=0; w<V.length; w++)
              S[0][v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …
    for (int k=0; k<V.length; k++)
       // For every pair of nodes
       for (int v=0; v<V.length; v++)
              for (int w=0; w<V.length; w++) {
                     int currD = S[k][v][w];
                     int toK = S[k][v][k];
                     int fromK = S[k][k][w];
                     S[k+1][v][w] = min(currD, toK+fromK);
              }
    return S;
}
```

```
int[][] APSP(E){ // Adjacency matrix E
    int[][] S = new int[V.length][V.length];//create memo table S

    // Initialize every pair of nodes
    for (int v=0; v<V.length; v++)
       for (int w=0; w<V.length; w++)
              S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …
    for (int k=0; k<V.length; k++)
       // For every pair of nodes
       for (int v=0; v<V.length; v++)
              for (int w=0; w<V.length; w++) {
                     int currD = S[v][w];
                     int toK = S[v][k];
                     int fromK = S[k][w];
                     S[v][w] = min(currD, toK+fromK);
              }
    return S;
}
```

```
int[][] APSP(E){ // Adjacency matrix E
    int[][] S = new int[V.length][V.length];//create memo table S

    // Initialize every pair of nodes
    for (int v=0; v<V.length; v++)
       for (int w=0; w<V.length; w++)
              S[v][w] = E[v][w];

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k=0; k<V.length; k++)
       for (int v=0; v<V.length; v++)
              for (int w=0; w<V.length; w++)
                     S[v][w] = min(S[v][w], S[v][k]+S[k][w]);
    return S;
}
```

What is the running time of Floyd Warshall?

1. $O(VE)$
2. $O(VE^2)$
3. $O(V^2E)$
4. $O(V^3)$
5. $O(V^3 \log E)$
6. $O(V^4)$

ARCHIPELAGO

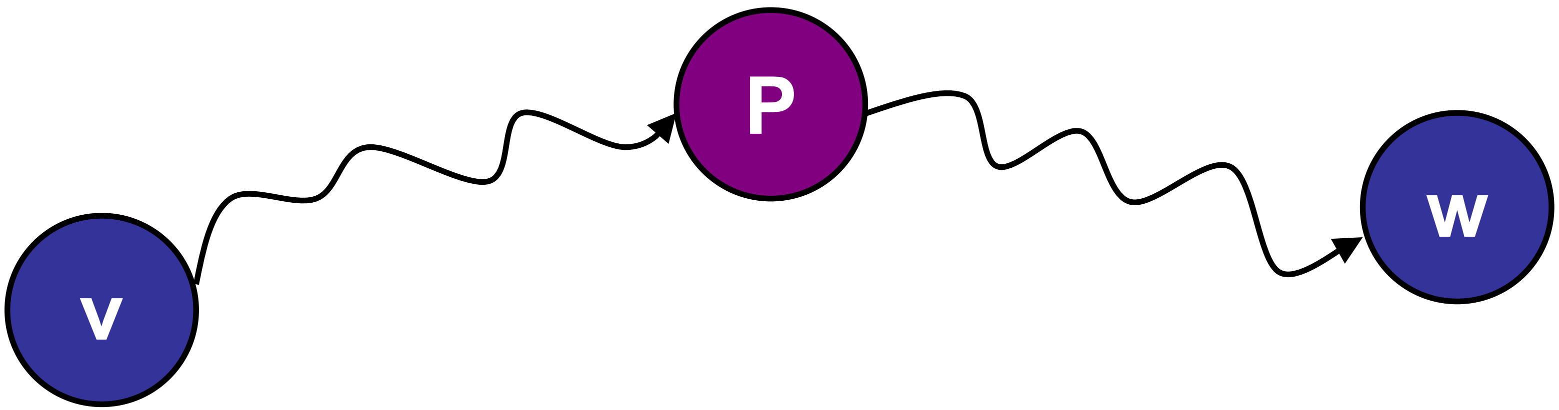
is open

```
int[][] APSP(E){ // Adjacency matrix E
    int[][] S = new int[V.length][V.length];//create memo table S

    // Initialize every pair of nodes
    for (int v=0; v<V.length; v++)
       for (int w=0; w<V.length; w++)
             S[v][w] = E[v][w]

    // For sets P0, P1, P2, P3, …, for every pair (v,w)
    for (int k=0; k<V.length; k++)
       for (int v=0; v<V.length; v++)
             for (int w=0; w<V.length; w++)
                   S[v][w] = min(S[v][w], S[v][k]+S[k][w]);
    return S;
}
```

Floyd-Warshall

Dynamic programming:

Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes only in the set P.



Dynamic Programming Recipe

Step 1: Identify optimal substructure

- Shortest paths are built out of shortest paths.

Step 2: Define sub-problems

- S(u,v,P) = shortest path from u to v using nodes in P.
- Consider only (n+1) sets P of increasing size.

Step 3: Solve problem using sub-problems

- $S(u,v,P_7) = \min(S[v,w,P_7], S[v, 8, P_7] + S[8, w, P_7])$.

Step 4: Write (pseudo)code.

Floyd-Warshall Variants

Path Reconstruction:

- Return the **actual** path from (v,w).
- Storing *all the shortest paths* requires (potentially) n^3 space!

  (n choose 2) pairs * n hops on the path

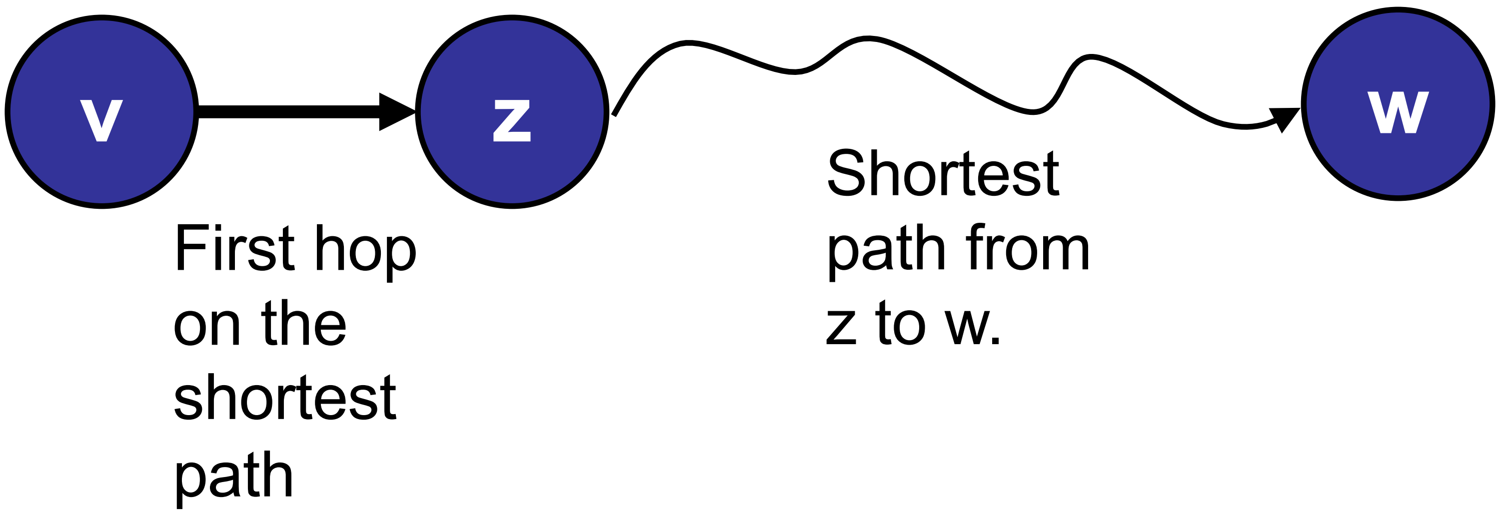
- How to represent it succinctly?
- How to store it efficiently?

ARCHIPELAGO

is open

Floyd-Warshall Variants

Optimal substructure:

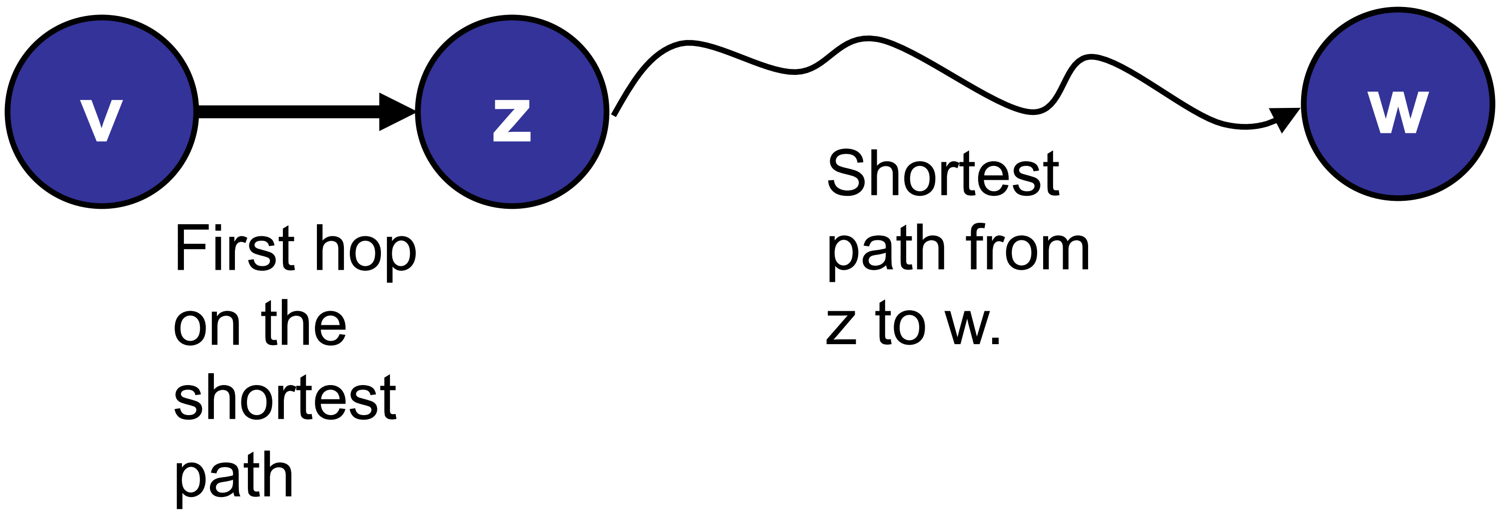


Shortest path from (v ➔ w) is:

(z + shortest path (z ➔ w)).

Floyd-Warshall Variants

Optimal substructure:



Only store first hop for each destination.

➔ routing table!

How much space to store all shortest paths in a routing table?

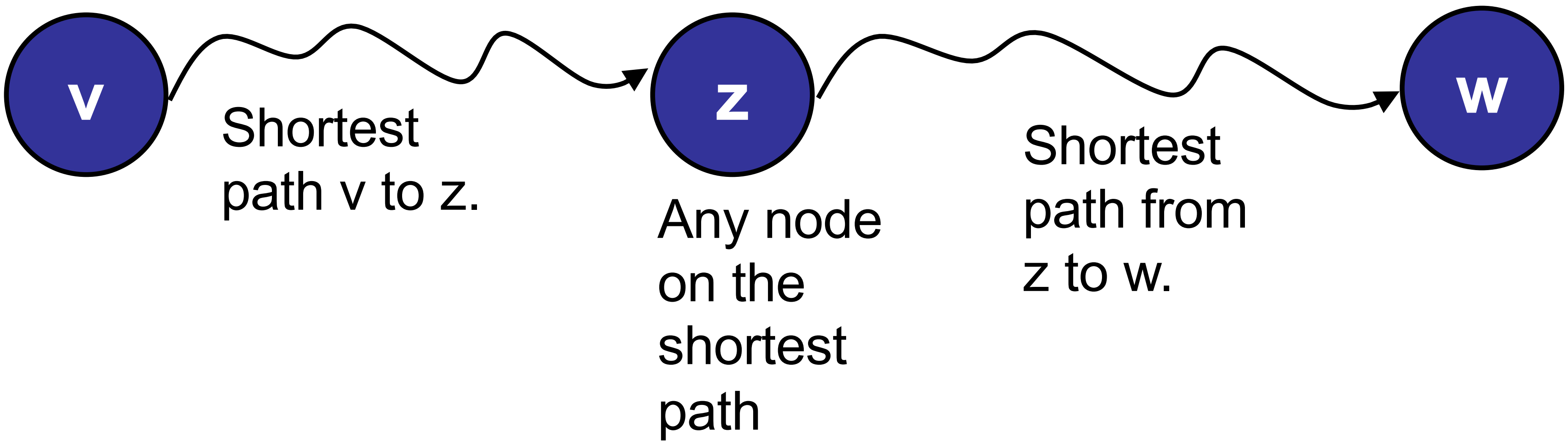
1. $O(V^2)$
2. $O(VE)$
3. $O(VE^2)$
4. $O(V^2E)$
5. $O(V^3)$
6. $O(V^3 \log E)$

ARCHIPELAGO

is open

Floyd-Warshall Variants

Optimal substructure:

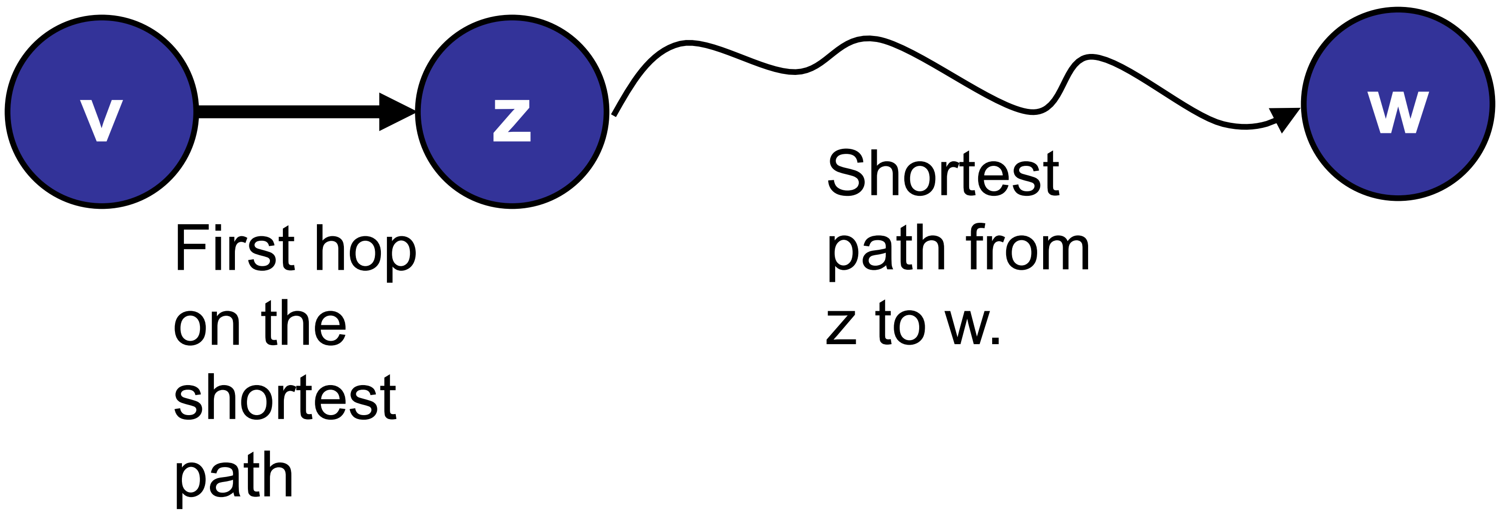


Store some node z on the shortest path from v to w.

Recursively find shortest path from v ➔ z and z ➔ w.

Floyd-Warshall Variants

Optimal substructure:



In Floyd-Warshall, store "intermediate node" whenever you modify/update the matrix entry for a pair.

Floyd-Warshall Variants

Transitive Closure:

Return a matrix M where:

- M[v,w] = 1 if there exists a path from v to w;
- M[v,w] = 0, otherwise.

Floyd-Warshall Variants

Minimum Bottleneck Edge:

- For (v,w), the bottleneck is the heaviest edge on a path between v and w.
- Return a matrix B where:

  B[v,w] = weight of the minimum bottleneck.

Roadmap

Dynamic Programming

- Basics of DP
- Example: Longest Increasing Subsequence
- Example: Bounded Prize Collecting
- Example: Vertex Cover on a Tree
- Example: All-Pairs Shortest Paths