Announcements

- SCS response rate is currently at: 69.31%
- Homework 4 is assigned, and is due next Tuesday
- Quiz 7 (on Topic 8) will be next Tuesday
- We will be allowing one project resubmission! See the D2L announcement for details

Topic 9: Algorithm Families

Algorithm Families

- What is an “algorithm family?”
  - A set of algorithms that share a common approach
- 5 common algorithm families:
  1. Divide and Conquer
  2. Dynamic Programming
  3. Greedy Algorithms
  4. Backtracking
  5. Approximation Algorithms

1. Divide and Conquer

- Divide and Conquer algorithms have three parts:
  1. Divide: Split the problem into 2 (or more) subproblems
  2. Solve: Compute solutions to the subproblems
  3. Conquer: combine the subproblem solutions into a solution to the original problem

Divide and Conquer: Example Algorithms

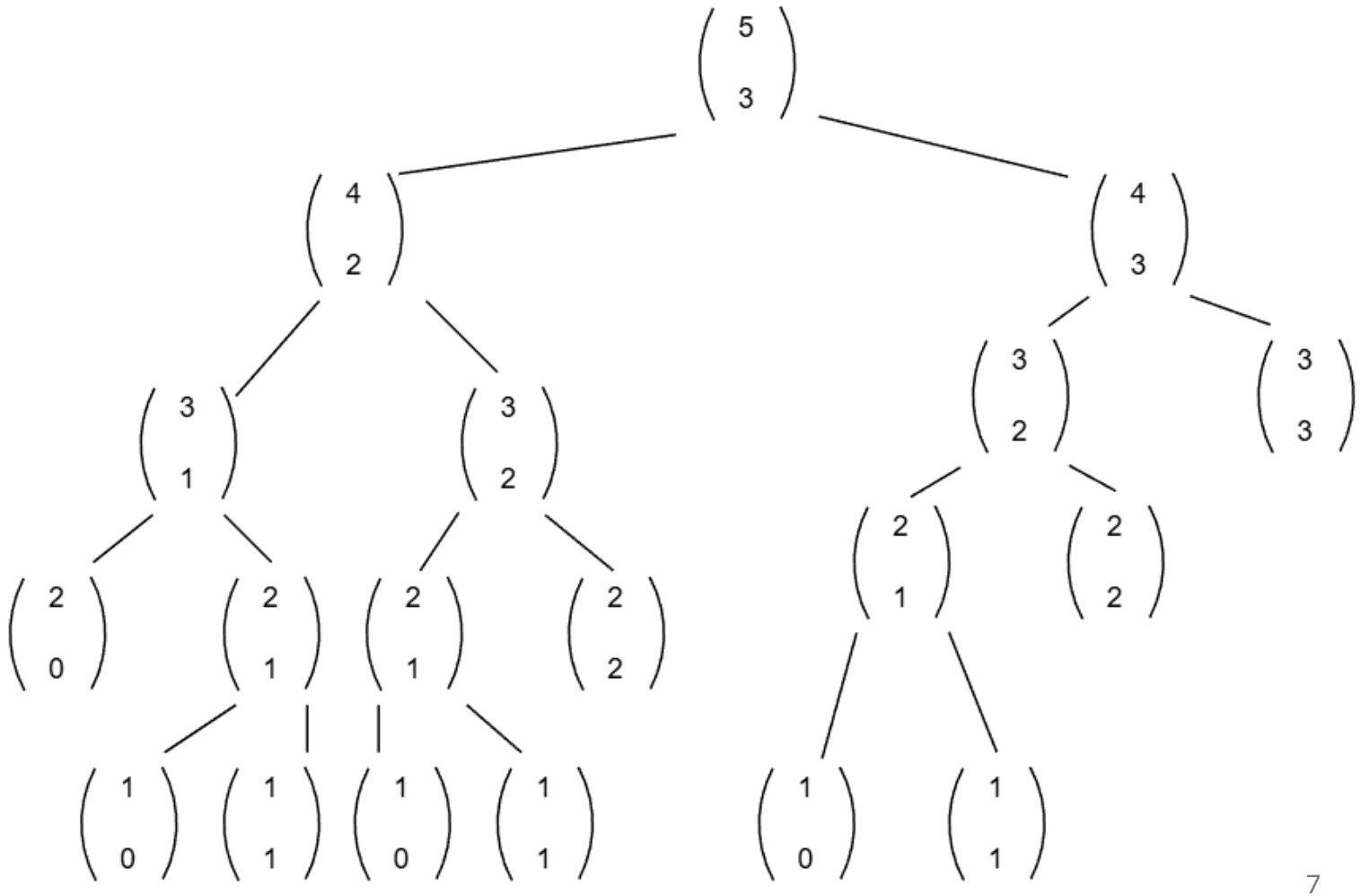
- Examples of Divide and Conquer algorithms:
  1. Mergesort
  2. Quicksort
  3. Binary Search?
     - “Decrease” and conquer

2. Dynamic Programming

- Divide and Conquer is great! … But it can potentially solve the same subproblem over and over again
  - Dynamic Programming helps address this issue… but how?

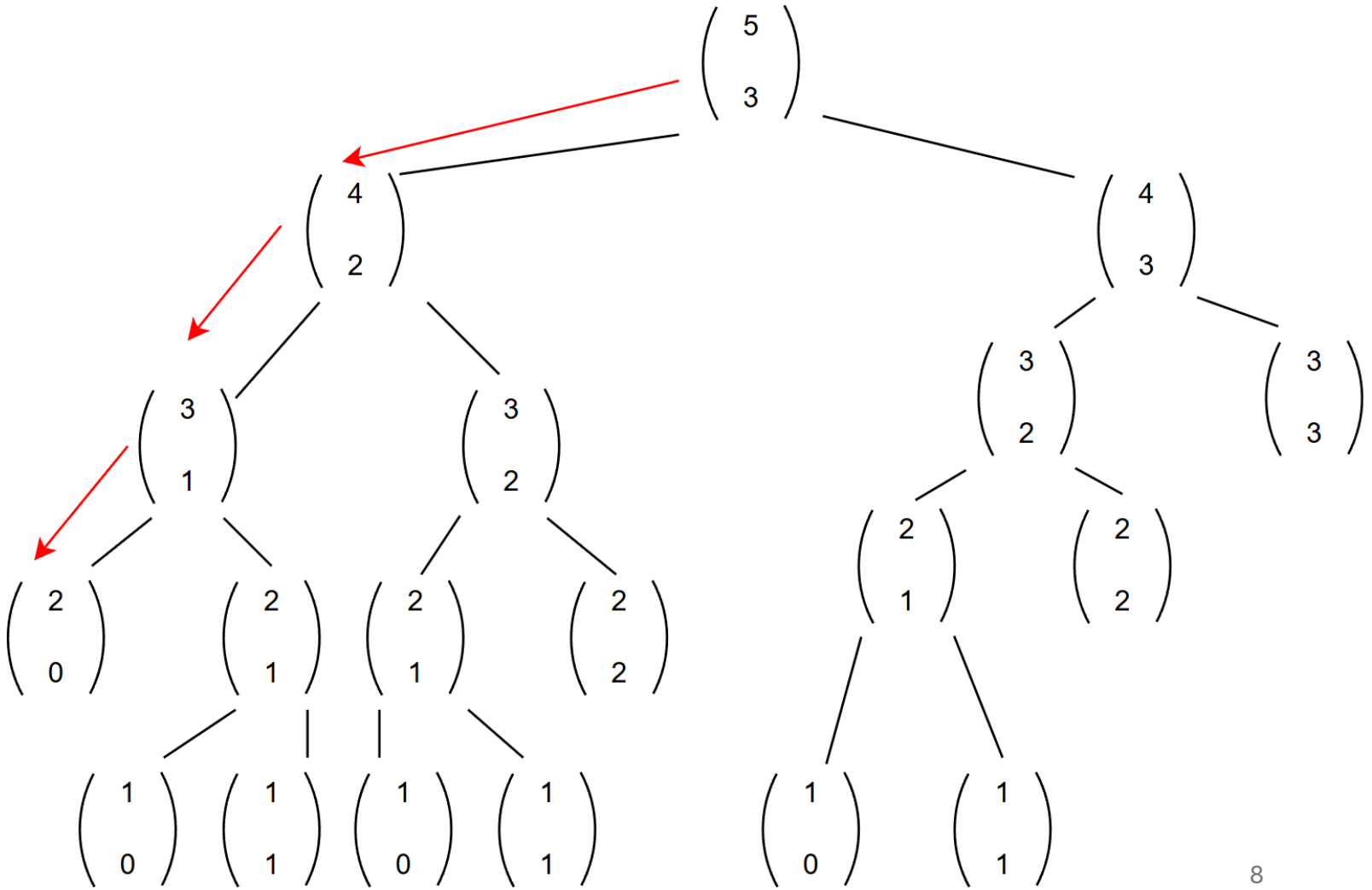
Dynamic Programming

- Recall Pascal's Identity: $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$
  - Here's an example call tree for $\binom{5}{3}$



Dynamic Programming

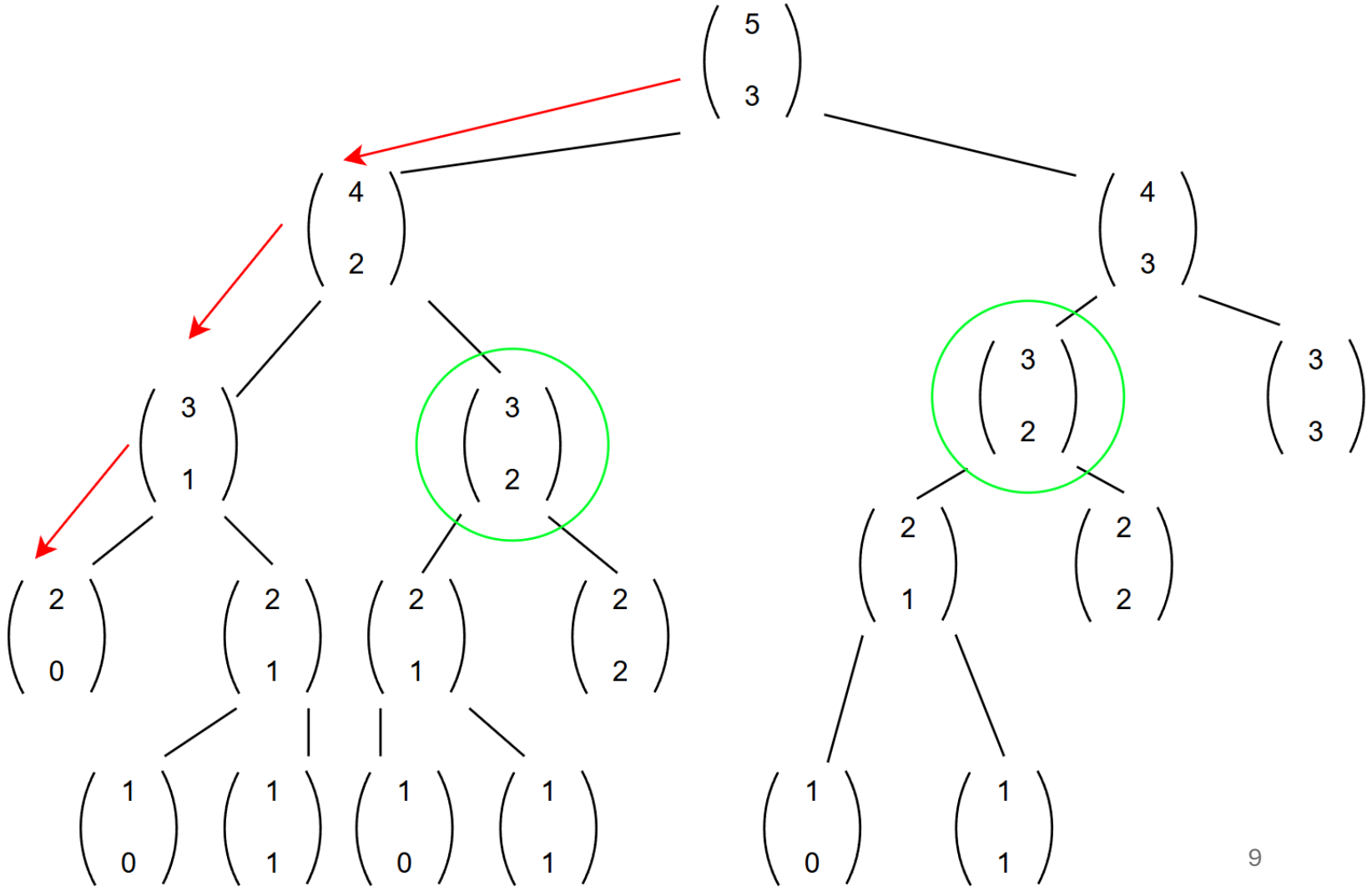
- Recall Pascal's Identity: $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$
  - Here's an example call tree for $\binom{5}{3}$



Dynamic Programming

- Recall Pascal's Identity: $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$
  - Here's an example call tree for $\binom{5}{3}$

N \ R	0	1	2
1	1	1	
2	1	2	1
3		3	3
4			6



Dynamic Programming Definition

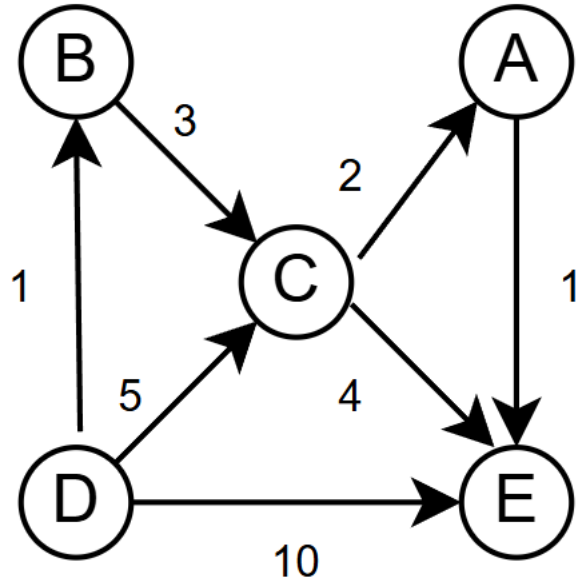
- **Dynamic Programming**: designing recursive algorithms to solve simpler sub-problems early, and to save those solutions so that we can look them up later instead of (repeatedly!) re-computing them

The All-Pairs Shortest Path (APSP) Problem

- APSP has a Dynamic Programming solution!
- Recall:
  - Dijkstra's Algorithm
  - Bellman-Ford Algorithm
- What's a "simple" way to find all the shortest paths between all vertices in a graph?

A Brute Force APSP Solution

- |V| calls to Dijkstra's Algorithm!
- But lots of repeated work…

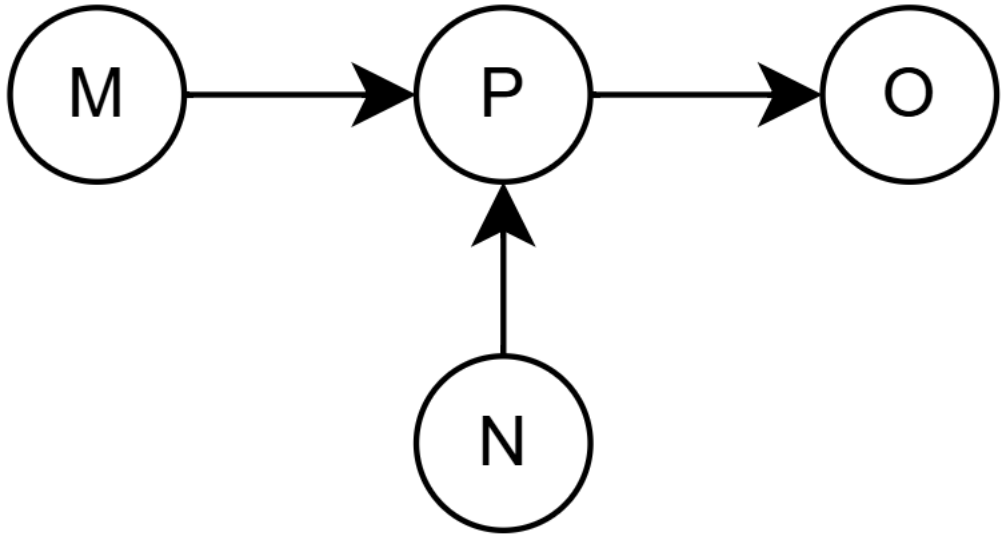


Detour: Transitive Closure of a Digraph

- **Transitive Closure**: the transitive closure of a graph G=(V,E) is the graph G*=(V,E*) such that E*={ (i,j) | G has a path from i to j}
- Why bother with this? Suppose you want to know: “Can we get from vertex A to vertex B?” E* has the answer!

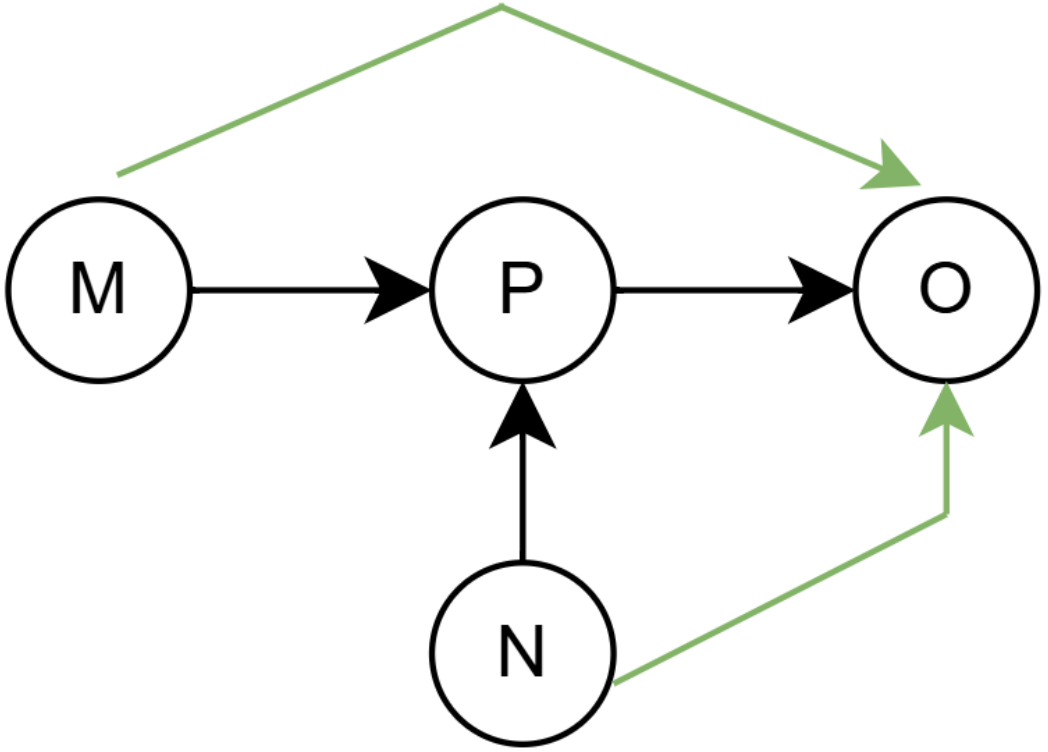
Example: Transitive Closure of a Digraph

- V = {M, N, O, P} E = { (M, P), (N, P), (P, O) }
- Question: What would E* look like?



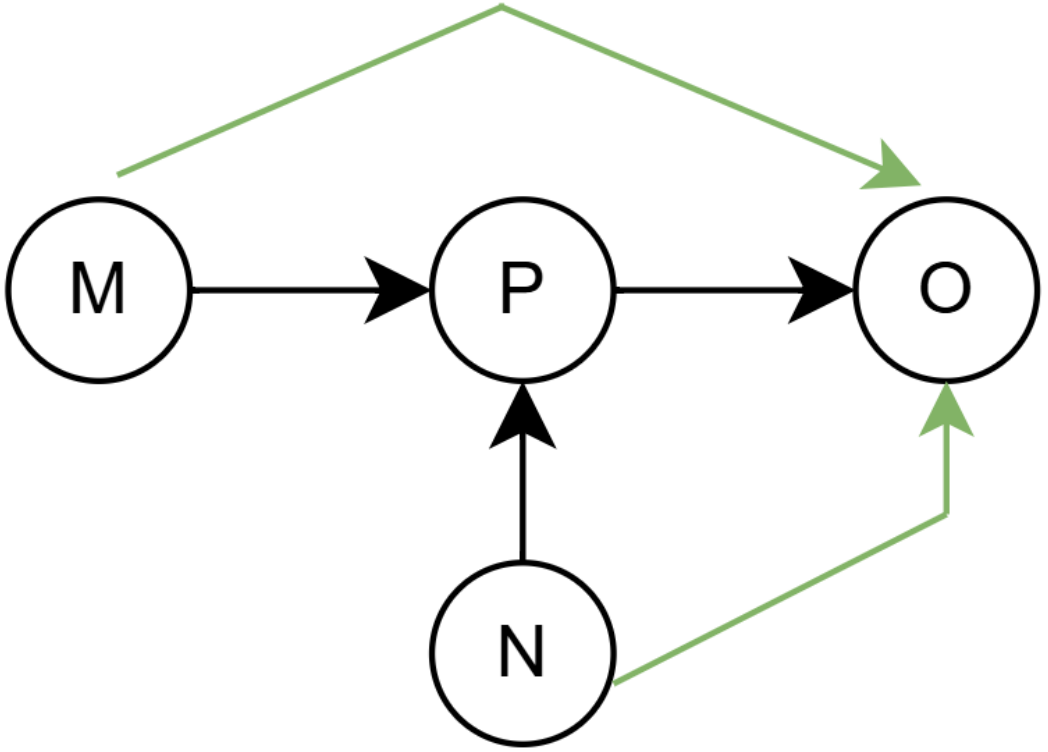
Example: Transitive Closure of a Digraph

- V = {M, N, O, P} E = { (M, P), (N, P), (P, O) }
- Question: What would E* look like?
  - E* = { (M, P), (N, P), (P, O), (M,O), (N,O) }



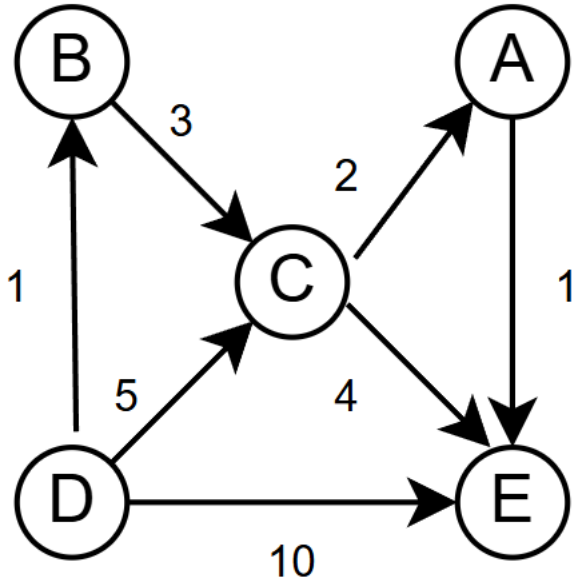
Example: Transitive Closure of a Digraph

- Okay, but what's transitive closure go to do with APSP?
  - Think of 'reversing' the transitive closure
- We know we can get from M to O… but how?



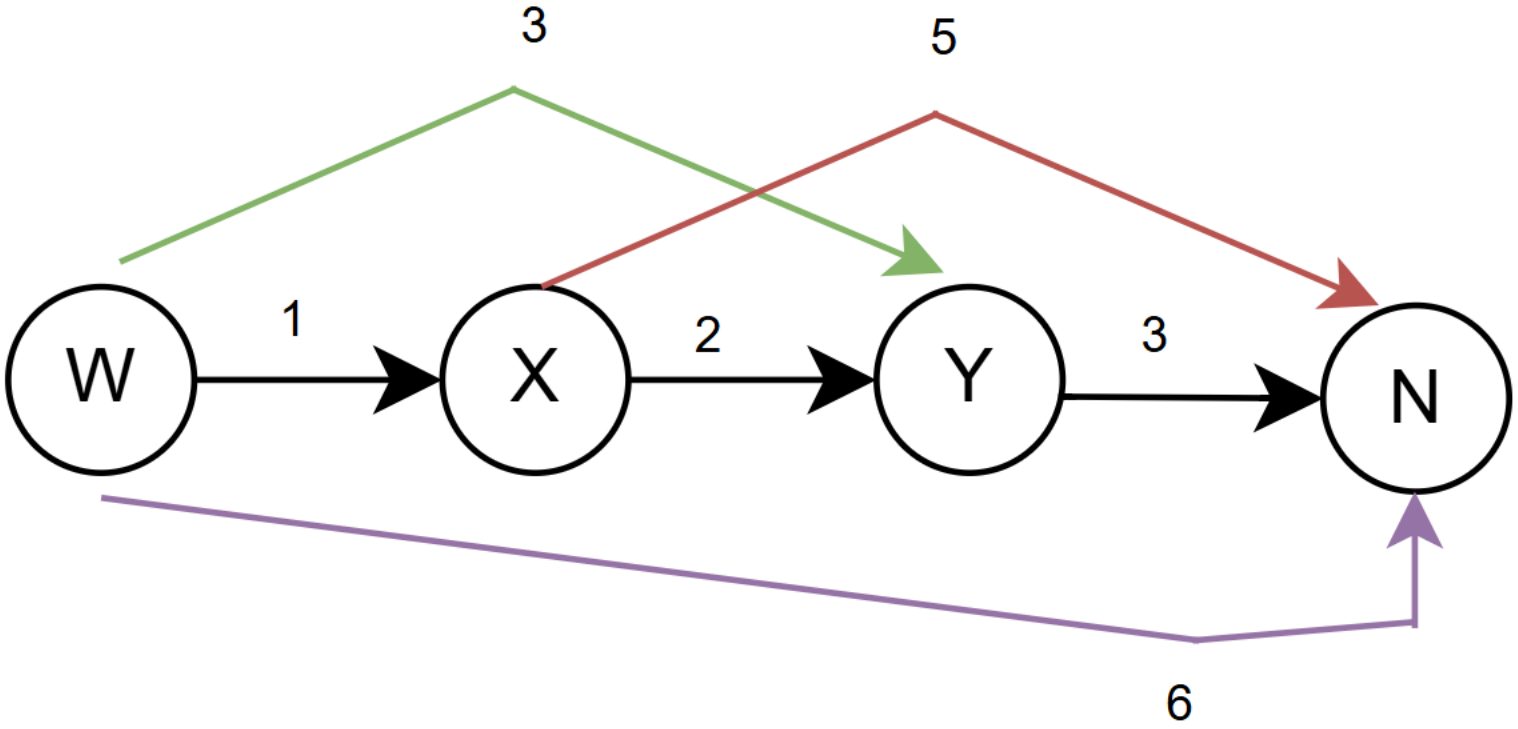
Example: Benefit of Computing the Transitive Closure

- Going back to our digraph example, there are 5 paths from D to E:
  1. D -> E
  2. D -> C -> E
  3. D -> B -> C -> E
  4. D -> C -> A -> E
  5. D -> B -> C -> A -> E
- Suppose we wanted the cheapest route to E
  - How can we find all of these options?



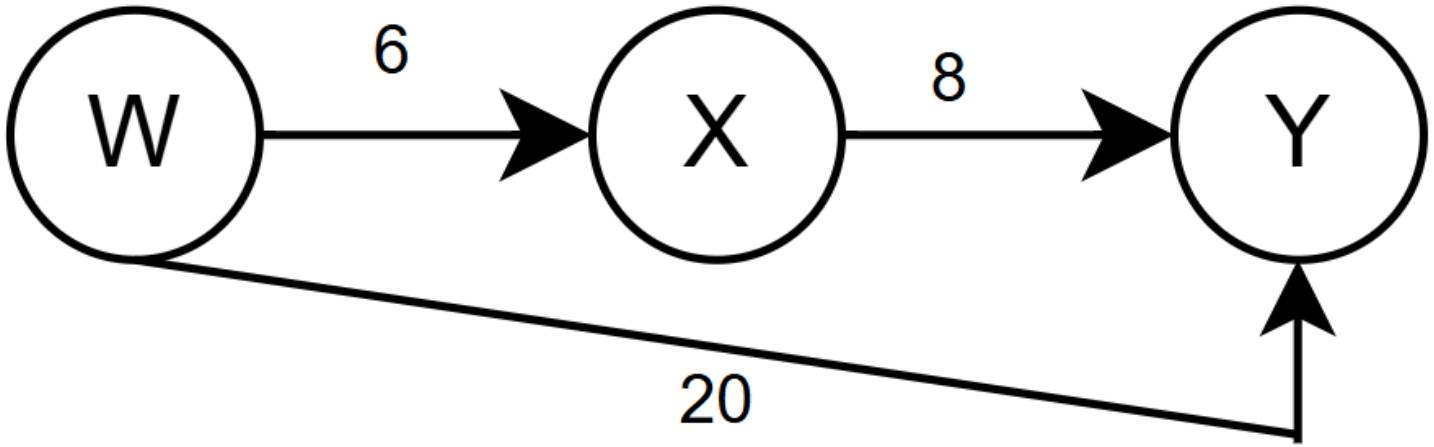
The Floyd-Warshall APSP Algorithm

- 1962, Robert Floyd, based on Stephen Warshall's 1962 transitive algorithm… which had been already published in 1959 by Bernard Roy… in French



Floyd-Warshall: Just One Data Structure!

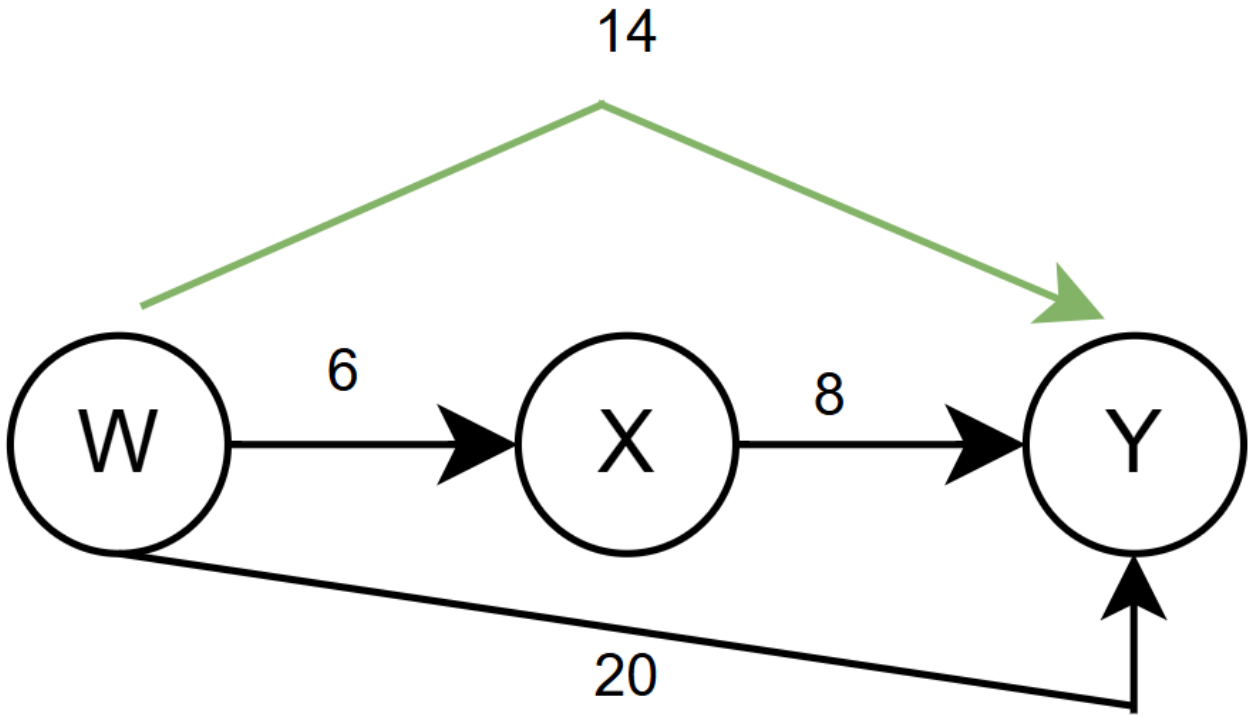
- As the algorithm computes updated path lengths, we store them in a ‘transitive adjacency matrix’, d[][].



d[][]	W	X	Y
W	0	6	20
X	inf	0	8
Y	inf	inf	0

Floyd-Warshall: Just One Data Structure!

- As the algorithm computes updated path lengths, we store them in a ‘transitive adjacency matrix’, d[][].



d[][]	W	X	Y
W	0	6	20->14
X	inf	0	8
Y	inf	inf	0

Floyd-Warshall: The Approach

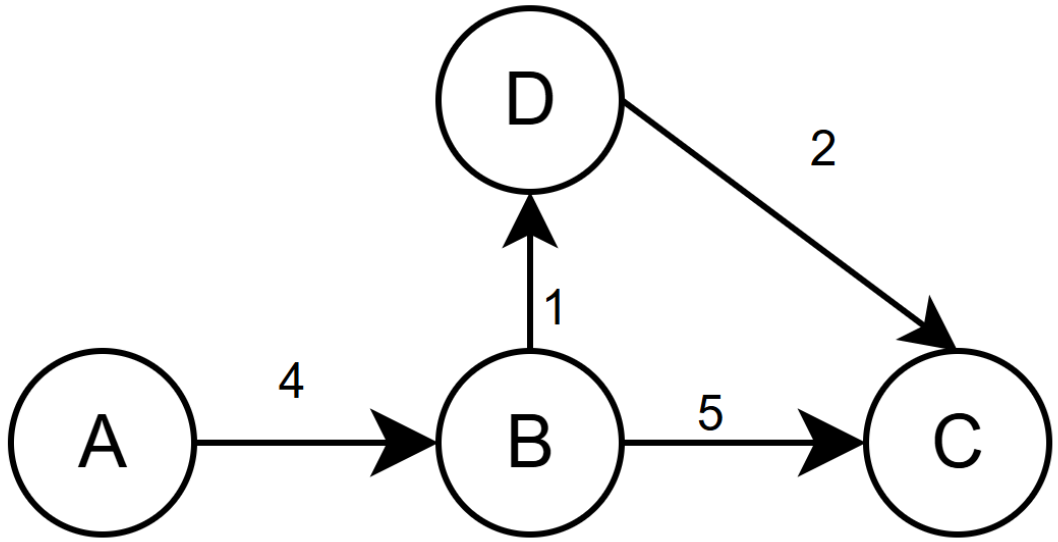
- For each (i, j) pair of vertices, see if the sum of the distances of the path (i, **k**) and (**k**, j) is less than the (i, j) distance—if such a **k** exists!
- We need the minimum of all possible options: (i, j) and the available (i, **k**) + (**k**, j) options. That is:
- d[i][j] = weight[i][j] (initially)
- d[i][j] = min(d[i][j], d[i][k] + d[k][j]), for all i,k,j

Floyd-Warshall: The Algorithm

- Assume: A digraph G=(V, E) with no negative-cost cycles
- Given: G, and an edge cost matrix weight[i][j]
- Produces: d[i][j], the ‘transitive adjacency matrix’ holding the least cost path from i to j.

```
d = weight
for k = 1 through |V|:
    for i = 1 through |V|:
        for j = 1 through |V|:
            d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

Example: Floyd-Warshall

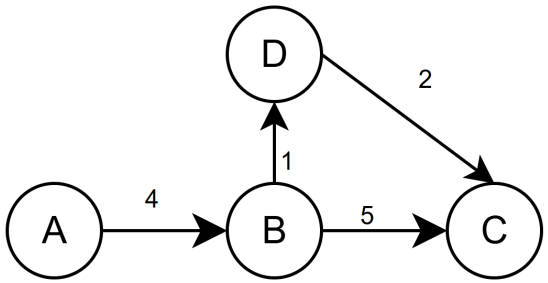


d[][] matrix	A	B	C	D
A	0	4	Inf	Inf
B	Inf	0	5	1
C	Inf	Inf	0	Inf
D	Inf	Inf	2	0

- Observations that will save work:
  1. A only has an out-edge
  2. C only has in-edges
  3. Can always ignore all i=k, j=k, and i=j cases

Floyd-Warshall (2): k=B

i	j	current	(i,B) + (B,j)	result
A	C	inf	(A,B) + (B,C)	9 – update!
A	D	inf	(A,B) + (B,D)	5 – update!
C	A	inf	(C,B) + (B,A)	inf
C	D	inf	(C,B) + (B,D)	inf
D	A	inf	(D,B) + (B,A)	inf
D	C	2	(D,B) + (B,C)	inf

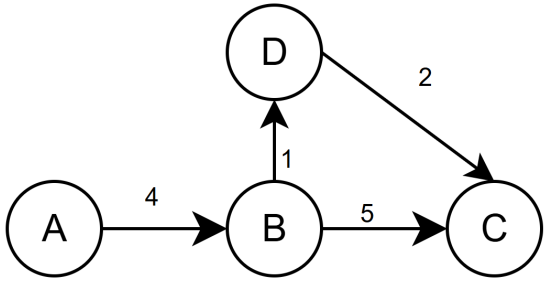


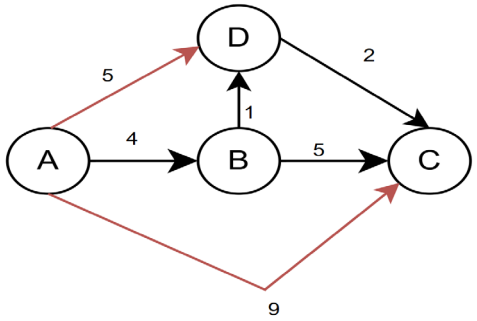
d[][] matrix	A	B	C	D
A	0	4	Inf -> 9	Inf -> 5
B	Inf	0	5	1
C	Inf	Inf	0	Inf
D	Inf	Inf	2	0

Floyd-Warshall (2): k=D

i	j	current	(i,D) + (D,j)	result
A	B	4	(A,D) + (D,B)	inf
A	C	9	(A,D) + (D,C)	7 – update!
B	A	inf	(B,D) + (D,A)	inf
B	C	5	(B,D) + (D,C)	3 – update!
C	A	inf	(C,D) + (D,A)	inf
C	B	inf	(C,D) + (D,B)	inf

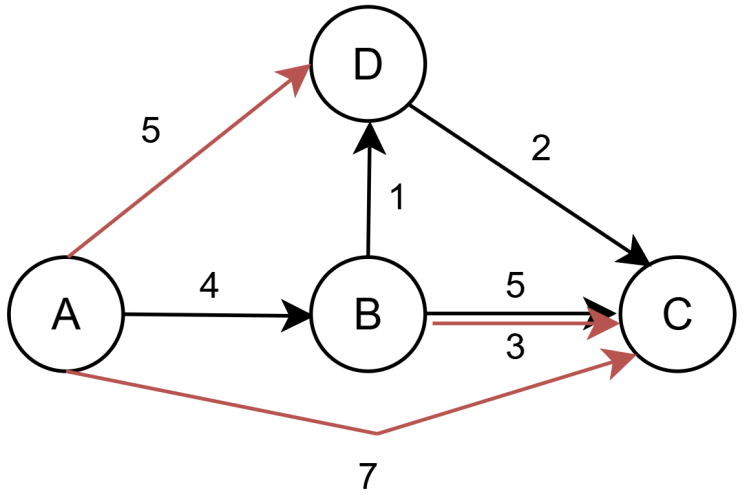
d[][] matrix	A	B	C	D
A	0	4	9 -> 7	5
B	Inf	0	5 -> 3	1
C	Inf	Inf	0	Inf
D	Inf	Inf	2	0





Floyd-Warshall (3)

- Note: It doesn't matter in which order you consider the k's! (starting with D, for this example, would give you the same answer



d[][] matrix	A	B	C	D
A	0	4	7	5
B	Inf	0	3	1
C	Inf	Inf	0	Inf
D	Inf	Inf	2	0

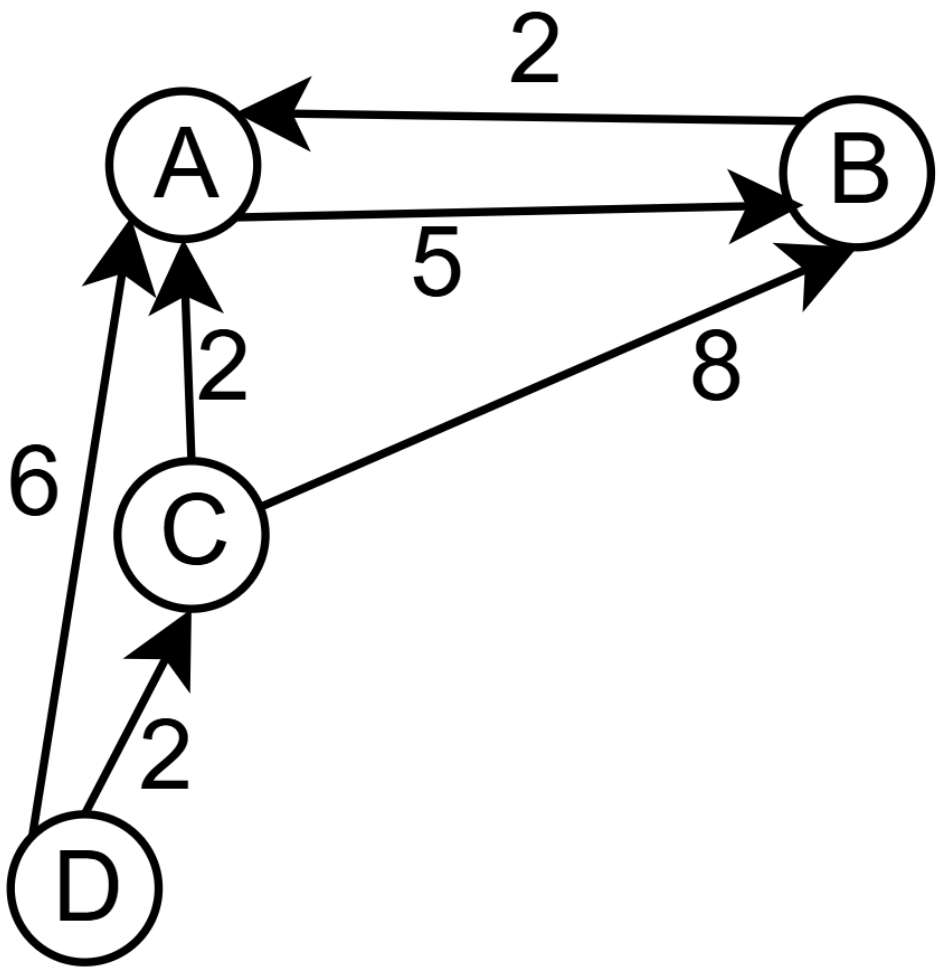
Floyd-Warshall (4)

- The d[][] holds our previously-discovered results, which are used to construct longer paths
  - This is an example of what?
  - Dynamic Programming!
- Efficiency of Floyd-Warshall: $\Theta(|V|^3)$
- Floyd-Warshall is often presented with a sequence of d[][]'s
- You can adjust the algorithm to also report the paths and costs discovered

Floyd-Warshall (5)

- A bit more on Floyd-Warshall:
- It won't work directly with negative-edge cycles, but can be modified to detect/report them. It does work with negative edge weights
- It works on any directed weighted graph. Given that, would it work on:
- An undirected graph?
- Yes, as you can easily make an undirected graph into a directed graph
- An unweighted graph?
- Yes, just set each edge's weight to 1 (or some arbitrary number)

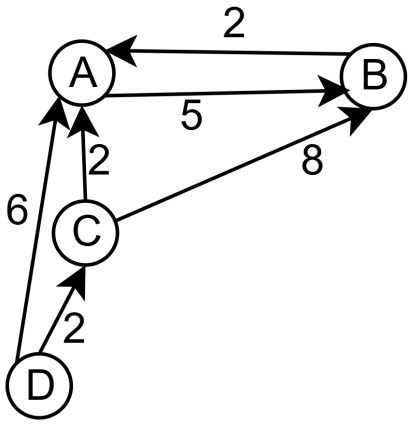
Example 2: Floyd-Warshall



d[][] matrix	A	B	C	D
A	0	5	Inf	Inf
B	2	0	Inf	Inf
C	2	8	0	Inf
D	6	Inf	2	0

Floyd-Warshall (6): k=A

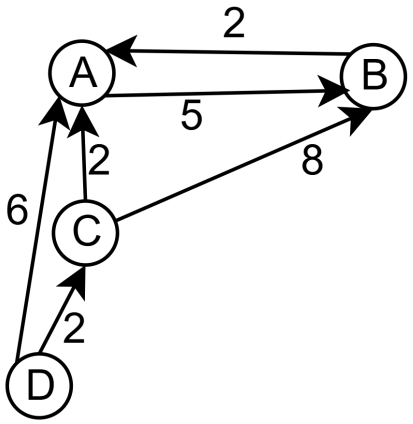
i	j	current	(i,A) + (A,j)	result
B	C	inf	(B,A) + (A,C)	inf
B	D	inf	(B,A) + (A,D)	inf
C	B	8	(C,A) + (A,B)	7->update
C	D	inf	(C,A) + (A,D)	inf
D	B	inf	(D,A) + (A,B)	11->update
D	C	2	(D,A) + (A,C)	2



d[][] matrix	A	B	C	D
A	0	5	Inf	Inf
B	2	0	Inf	Inf
C	2	8->7	0	Inf
D	6	Inf->11	2	0

Floyd-Warshall (6): k=B

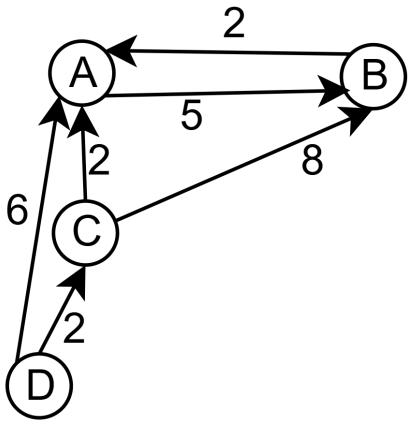
i	j	current	(i,B) + (B,j)	result
A	C	inf	(A,B) + (B,C)	inf
A	D	inf	(A,B) + (B,D)	inf
C	A	2	(C,B) + (B,A)	2
C	D	inf	(C,B) + (B,D)	inf
D	A	6	(D,B) + (B,A)	6
D	C	2	(D,B) + (B,C)	2



d[][] matrix	A	B	C	D
A	0	5	Inf	Inf
B	2	0	Inf	Inf
C	2	7	0	Inf
D	6	11	2	0

Floyd-Warshall (6): k=C

i	j	current	(i,C) + (C,j)	result
A	B	inf	(A,C) + (C,B)	inf
A	D	inf	(A,C) + (C,D)	inf
B	A	2	(B,C) + (C,A)	2
B	D	inf	(B,C) + (C,D)	inf
D	A	6	(D,C) + (C,A)	4->update
D	B	11	(D,C) + (C,B)	9->update



d[][] matrix	A	B	C	D
A	0	5	Inf	Inf
B	2	0	Inf	Inf
C	2	7	0	Inf
D	6->4	11->9	2	0

Back to: Dynamic Programming

- Dynamic Programming's major application areas are:
  1. A recursive solution with common subproblems (like our Pascal's Identity example)
  2. When many solutions exist, but you want the optimal solution
- Speaking of optimality…

The Principle of Optimality

- In an optimal sequence of decisions/choices, **every subsequence must also be optimal**

3. Greedy Algorithms

- The idea: At each step, we can select the best available choice
- We hope: A locally-optimal choice will turn out to be the globally-optimal choice
- In general: if you find a greedy solution to a problem, you can also find a dynamic programming solution—but the reverse isn't true as often

Greedy Algorithms: When does a Greedy Algorithm Apply?

1. **Greedy Choice Property**: It must be possible for a globally optimal solution to follow from a locally optimal one
   - Example: Climbing up Mount Lemmon in the fog
2. **Optimal Substructure**: It must be the case that the optimal solution uses optimal subproblem solutions

- So, an optimal solution = a greedy choice + the optimal solutions to the remaining subproblems

Greedy Algorithms: Characteristics

- We need a group of solution components:

1. The group of solution components that have yet to be selected
   - It might help to order the choices
2. The group of selected but rejected components
3. The group of selected and retained components

Greedy Algorithms: Characteristics

- And we need some functions:

1. selection() : determines which available solution component to select
2. feasible() : will the addition of this component to the ones already selected form the subset of a potential solution?
3. solution() : does the current set of components form a solution?
4. (optionally) cost() : the ‘value’ of the solution

Example: Making Change

- You buy snacks for $4.08 and pay with a $5 bill. How does the clerk make $0.92 of change?
- Greedily! Quarters, then dimes, nickels, and pennies.
- Suppose we had a 12-cent coin. Could we make 16 cents of change efficiently by being greedy?
- 12-cent coin + 4 pennies = 5 coins
- A dime (10 cent) + a nickel (5 cent) + a penny (1 cent) = 3 coins!

Example: Making Change (2)

- Back to paying $4.08 with a $5 bill… But now our clerk is a computer! How would we set up the solution components and functions to make the computer optimally return the change?
- What are our solution components?
  - Coins! Quarters, dimes, nickels, pennies
- What are our functions? What would they do in this context?
  1. selection() : Select the largest coin available for selection
  2. feasible() : Check if adding the coin will give too much change back; if so, reject and add coin to selected but rejected group
  3. solution() : Do these coins add up to the amount of change?
  4. cost() : Add up the number of coins required for the solution

Other Greedy Algorithms

- Our MCST examples (Prim's and Kruskal's). Why?
- Kruskal's chooses the cheapest edge that doesn't cause a cycle, and Prim's chooses the cheapest edge in the fringe set
- Dijkstra's Algorithm?
- Yes, it picks the node with the smallest distance from the source every time
- Huffman Coding
- It's hard to prove that greedy approaches will always work

Example: Kruskal's Algorithm

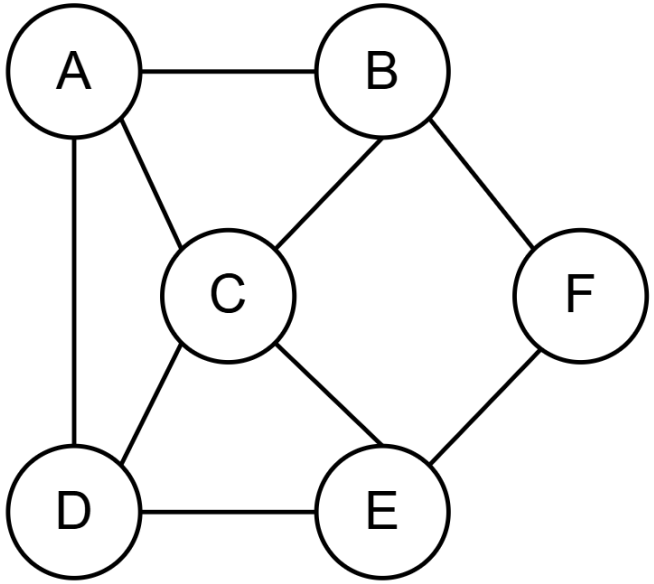
- Let's see if we can "implement" Kruskal's using our greedy algorithm functions (selection(), feasible(), solution(), cost())
- What are our solution components?
  - Edges!
- What would our functions do in this context?
  1. selection() : Select the cheapest edge available.
  2. feasible() : Check if adding the edge would cause a cycle given the selected and retained edges; if so, move that edge to selected but rejected.
  3. solution() : Does the number of edges = |V| - 1?
  4. cost() : Add up the weights of the selected and retained edges.

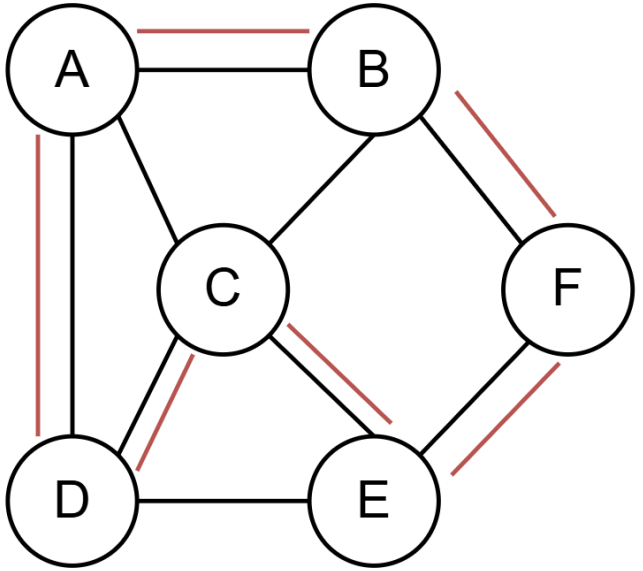
4. Backtracking

- The idea: Be ‘flexibly greedy’: Make a choice, but be prepared to reconsider it if necessary
- Simple example: Solving a maze

Example: Hamiltonian Cycles

- Definition: A simple cycle containing each vertex of a graph is called a **Hamiltonian Cycle** of the graph





Other Backtracking Algorithms

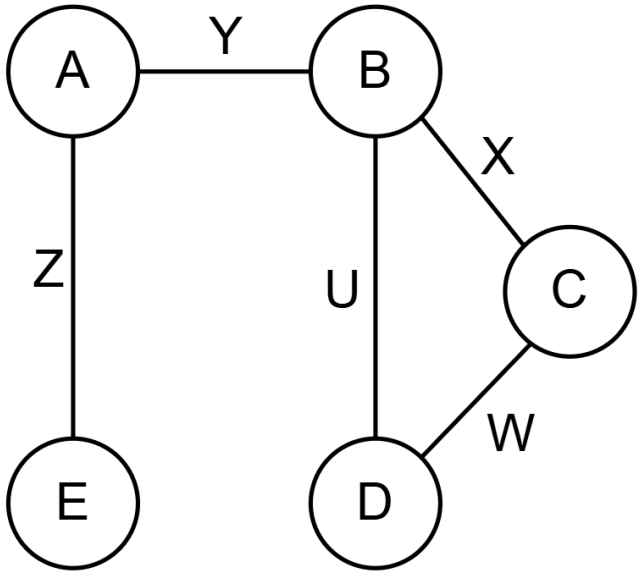
- N-Queen Puzzle (and crossword solvers, sudoku solvers…)
- Subset Sum problem (is there a subset of a set that sums up to some value?)
- Graph Coloring (assigning “colors” to vertices such that no pair of adjacent vertices share the same “color”)

5. Approximation Algorithms

- Idea: When no efficient solution exists to an optimization problem, would you settle for an approximately optimal one?
  - Sometimes! For example, Hamiltonian Cycles can’t be approximated

Example: The Vertex Cover Problem

- Definition: A **vertex cover** for an undirected graph G = (V, E) is a set of vertices C $\subseteq$ V such that each edge in E is incident on at least one vertex in C
- Take a look at this graph:
- Is there a 2-vertex solution?



A 2-Approximation Vertex Cover Algorithm

- “2-Approximation” means the # of vertices <= twice the minimum

```
F = a copy of E (the graph’s edge set)
C = Null (C is the vertex cover set)
While F != Null:
    Select an arbitrary edge e from F
    C = C ∪ the vertices incident on e
    F = F – all the edges incident on at least one vertex of e
Return C
```

- Efficiency?
- O(|V| + |E|)

Other Approximation Algorithms

- Subset Problem (Polynomial Time Approximation Scheme)
- Traveling Salesman approximation