Dynamic Programming

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

Richard Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Dynamic Programming – Paradigm and Floyd's Algorithm

Textbook Reading:

- Section 8.1 Optimization Problems and the Principle of Optimality, pp. 331-333.
- Section 8.5 Floyd's Algorithm, pp. 355-359.

Dynamic Programming Book by Richard Bellman

The book Dynamic Programming by Richard Bellman is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading "Exercises and Research Problems," with easy questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied: "If you can solve it, it is an exercise; otherwise it's a research problem."

Problems we will Solve in this Course Using Dynamic Programming

- All-Pairs Shortest-Paths in Weighted Digraphs
  - Floyd's Algorithm
  - The unweighted case of Floyd's algorithm was discovered independently by Warshall, so Floyd's algorithm is sometimes called the Floyd-Warshall algorithm
- Optimal Chained Matrix Product Parenthesization
- Optimal Binary Search Trees
- Edit Distance

Comparison Algorithmic Paradigms

Greedy Method. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem. Top-Down Recursion

Dynamic programming. Break up a problem into a series of overlapping, reusable sub-problems, and build up solutions to larger and larger sub-problems.
Bottom-up Recursion

Dynamic Programming is a bottom-up recursive approach so avoids redundancy

- Top-down recursion may involve a lot of redundant calculations.
- For example consider the problem of computing the n^{th} Fibonacci number *Fib*(*n*) that we discussed earlier in this course.
- The top-down recursive approach involves returning
  `Fib(n - 1) + Fib(n - 2).`
- This entails an exponential amount of redundancy, i.e., the tree of recursive calls has more than $(1.5)^n$ vertices.
- On the other hand, the bottom up approach using the same recurrence relation involves no redundant calculations.
- It involves looping *n* – 1 times updating `Previous` and `Current` initially set to `Previous = 0; Current = 1` as follows:
  `Temp = Current; Current = Current + Previous;`
  `Previous = Temp;` The final value of `Current` is *Fib*(*n*).

Sometimes redundancy is a good thing! But not redundant calculations!

Dynamic Programming History

Richard Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Robotics – Bearcat Robot
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Floyd's algorithm for all-pairs shortest paths.
- CKY algorithm for parsing context free grammars.

Dynamic Programming Strategy

- Dynamic programming is a design strategy that involves constructing a solution *S* to a given problem by building it up dynamically from solutions to **smaller (or simpler) problems $S_1, S_2, \ldots, S_m$** of the same type.
- The solution to any given smaller problem S_i is itself built up from the solutions to even smaller (simpler) subproblems, and so forth. We start with the known solutions to the smallest (simplest) problem instances and build from there in a **bottom-up fashion.**
- To be able to reconstruct *S* from $S_1, S_2, \ldots, S_m$, some additional information is usually required.
- Let *Combine* denote the function that combines $S_1, S_2, \ldots, S_m$, using the additional information to obtain *S*, so that

$$S = Combine(S_1, S_2, \ldots, S_m)$$

Optimization Problems and the Principle of Optimality (POO )

Dynamic Programming is often used to solve **optimization problems**.

Principle of Optimality holds if the following is always true:

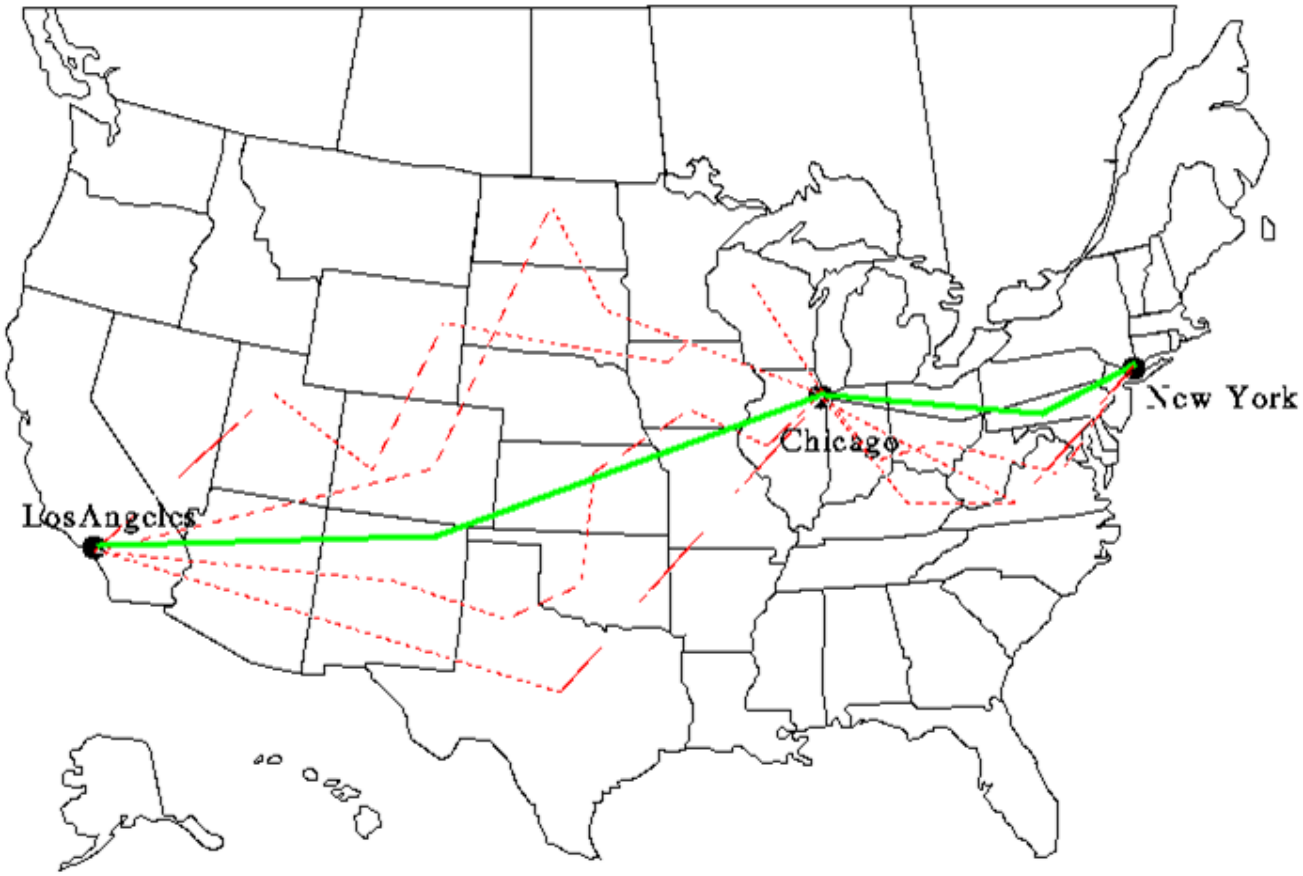
If $S = Combine(S_1, S_2, \ldots, S_m)$ is an *optimal* solution to the problem, then $S_1, S_2, \ldots, S_m$ are *optimal* solutions to their respective subproblems.

Building Block Analogy

If you think of the solutions to the subproblems as building blocks, then the Principle of Optimality says that an optimal solution to the problem is obtained by combining **only optimal building blocks**.

This significantly restricts the number of building blocks that need to be considered, which makes the Dynamic Programming algorithm much faster.

Example Principle of Optimality



If the green path is a shortest path from LA to NY then the green subpath from LA to Chicago and the green subpath from Chicago to NY must both be shortest paths.

Shortest Paths Satisfies Principle of Optimality

PSN. Consider a digraph $D = (V, E)$ whose edges are weighted with real numbers such that there are **no negative cycles**, i.e., no directed cycles so that the sum of the weights over the edges of the cycle is strictly negative. Then the Principle of Optimality holds for shortest paths.

Floyd's Algorithm for All-Pairs Shortest Paths

- Floyd's algorithms applies to a weighted digraph *D*, where we will allow negative weights, but no negative cycles.
- A negative cycle is a directed cycle, i.e., closed path, such that the sum of the weights are negative.
- Floyd's algorithms solves the problem of finding shortest paths between all pairs of vertices in the digraph *D*.

Floyd's Algorithm – Definitions

$D = (V,E)$ a digraph where $V = \{0, ..., n - 1\}$

Let $V_k = \{0, ..., k - 1\}$, $k = 1, \ldots, n - 1$ and let V_0 be the empty set.

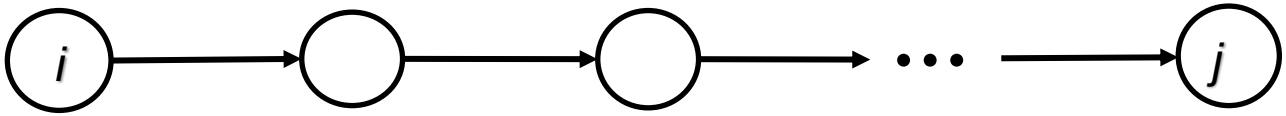
Let $S_k(i,j)$ denote the length (i.e., weight) of a shortest path P from i to j, whose **interior** vertices (if any) **all belong to in V_k .** If no such path P exists and let $S_k(i,j) = \infty$.

Since, by definition, $S_0(i,j)$ is the weight of a shortest path from i to j containing **no interior vertices**, S_0 is equal to the **weight matrix *W*** defined as follows:

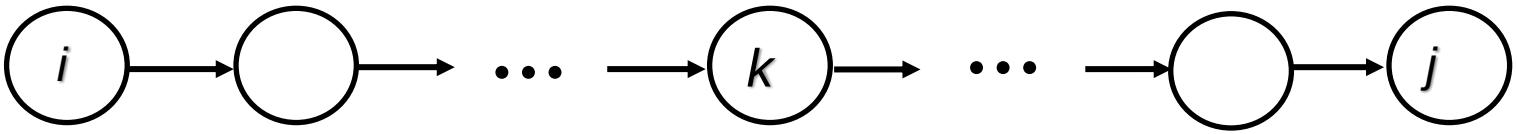
$W(i,j) = w(ij)$ if $ij \in E$, 0 if $i = j$ and ∞, otherwise.

Note that **$S_n(i,j)$ is the length of a shortest path in *D* from *i* to *j***.

Two Cases



Case 1. Shortest path from *i* to *j* does **not contain *k*** as an interior vertex. Then, $S_{k+1}(i,j) = S_k(i,j)$.



Case 2. Shortest path from *i* to *j* **contains *k*** as an interior vertex. Then, it follows from the **Principle of Optimality** that

$$S_{k+1}(i,j) = S_k(i,k) + S_k(k,j).$$

Recurrence Relation for Floyd's Algorithms

How do we know which case to choose? The answer is, **whichever is smallest**. This yields the recurrence relation:

$$S_{k+1}(i,j) = \min \{S_k(i,j), S_k(i,k) + S_k(k,j)\},$$

Initial condition $S_0(i,j) = W(i,j)$,

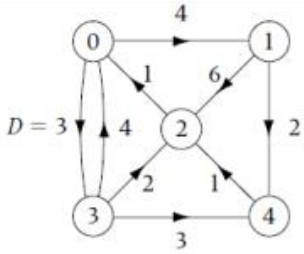
Keeping Track of Shortest Paths in Floyd's Algorithm

Recurrence relation for keeping track of shortest paths

$$P_{k+1}(i,j) = \begin{cases} P_k(i,j) & \text{if } S_{k+1}(i,j) = S_k\ (i,j) \\ k & \text{otherwise.} \end{cases}$$

Initial Condition $P_0(i,j) = -1$ for all $i, j \in V$.

Stages of Floyd's algorithm for a sample weighted digraph D



$D =$

$$S_0 = W = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & \infty & 0 & \infty & \infty \\ 4 & \infty & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix}$$

$$S_1 = \begin{pmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & \infty \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \qquad P_1 = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0 & 4 & 10 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_3 = \begin{pmatrix} -1 & -1 & 1 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_4 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

$$S_5 = \begin{pmatrix} 0 & 4 & 5 & 3 & 6 \\ 4 & 0 & 3 & 7 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 3 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{pmatrix} \qquad P_5 = \begin{pmatrix} -1 & -1 & 3 & -1 & 1 \\ 4 & -1 & 4 & 4 & -1 \\ -1 & 0 & -1 & 0 & 1 \\ 2 & 2 & -1 & -1 & -1 \\ 2 & 2 & -1 & 2 & -1 \end{pmatrix}$$

Pseudocode for Floyd's Algorithm

```
procedure Floyd(W[0:n – 1,0:n – 1],P[0:n – 1,0:n – 1],S[0:n – 1,0:n – 1])
Input:   W[0:n – 1,0:n – 1]   (weight matrix for a weighted digraph D)
Output:  P[0:n – 1,0:n – 1]   (matrix implementing shortest paths)
         S[0:n – 1,0:n – 1]   (distance matrix where S[u,v] is the weight of a
                                shortest path from u to v in G)
    for i ← 0 to n – 1  do                    // initialize P and S
            for j ← 0 to n – 1 do
                    P[i,j] ← –1
                    S[i,j] ← W[i,j]
            endfor
    endfor
    for k ← 0 to n – 1 do
        for i ← 0 to n – 1 do
            for j ← 0 to n – 1 do
                if S[i,j] > S[i,k] + S[k,j] then
                    P[i,j] ← k
                    S[i,j] ← S[i,k] + S[k,j]
                endif
            endfor
        endfor
    endfor
end Floyd
```

Complexity of Floyd's Algorithm

Floyd's algorithm involves three nested loops from 0 to $n - 1$. Therefore, its worst-case complexity is given by

$$W(n) \in \Theta(n^3)$$

Longest Paths in DAGs

- Consider a DAG *D* = (V,E) and a weighting w of the edges with any real numbers, positive or negative. Since *D* contains no directed cycles, in particular, it contains no negative directed cycles, so Floyd's algorithm applies.

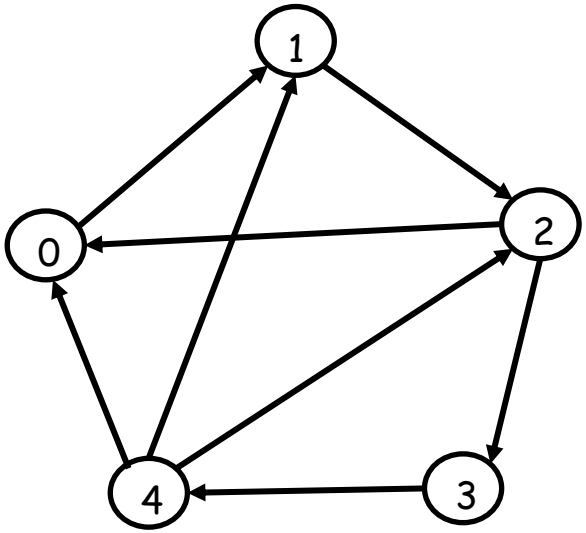
- Note that by replacing *w* with –*w*, i.e., replacing every weight w(e) with –w(e), and applying Floyd's algorithm we obtain **longest** paths between every pair of vertices in D with respect to the original weighting w, i.e., paths such that the sum of the w-weights on the edges of the path is maximum over all paths joining the same pair of vertices.

- As a special case, given an unweighted DAG, longest paths between every pair of vertices can be obtain by assigning each edge the weight -1 and then applying Floyd's algorithm.

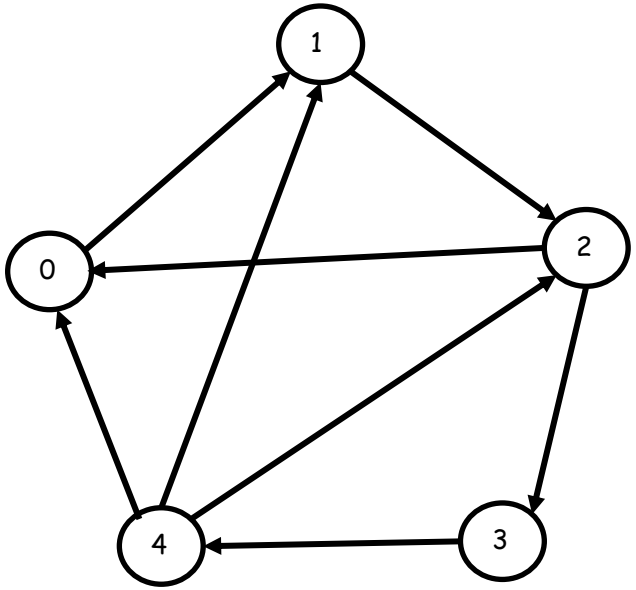
Warshall's Algorithm

The special case of Floyd's algorithm for an unweighted graph or digraph, i.e., all edges have unit weight was discovered independently by Warshall. Floyd's algorithm is often called the Floyd-Warshall algorithm.

Show the action of Warshall's algorithm for the following digraph:



$$S_0 = W = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & 1 & \infty & \infty \\ 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 1 & 1 & 1 & \infty & 0 \end{pmatrix}$$



$$S_1 = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & 1 & \infty & \infty \\ 1 & 2 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 1 & 1 & 1 & \infty & 0 \end{pmatrix} \quad P_1 = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_2 = \begin{pmatrix} 0 & 1 & 2 & \infty & \infty \\ \infty & 0 & 1 & \infty & \infty \\ 1 & 2 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 1 & 1 & 1 & \infty & 0 \end{pmatrix} \quad P_2 = \begin{pmatrix} -1 & -1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$$S_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & \infty \\ 2 & 0 & 1 & 2 & \infty \\ 1 & 2 & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ 1 & 1 & 1 & 2 & 0 \end{pmatrix} \quad P_3 = \begin{pmatrix} -1 & -1 & 1 & 2 & -1 \\ 2 & -1 & -1 & 2 & -1 \\ -1 & 0 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 2 & -1 \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 0 & 1 & 2 & 3 \\ 1 & 2 & 0 & 1 & 2 \\ \infty & \infty & \infty & 0 & 1 \\ 1 & 1 & 1 & 2 & 0 \end{pmatrix} \quad P_4 = \begin{pmatrix} -1 & -1 & 1 & 2 & 3 \\ 2 & -1 & -1 & 2 & 3 \\ -1 & 0 & -1 & -1 & 3 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 2 & -1 \end{pmatrix}$$

$$S_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 0 & 1 & 2 & 3 \\ 1 & 2 & 0 & 1 & 2 \\ 2 & 2 & 2 & 0 & 1 \\ 1 & 1 & 1 & 2 & 0 \end{pmatrix} \quad P_4 = \begin{pmatrix} -1 & -1 & 1 & 2 & 3 \\ 2 & -1 & -1 & 2 & 3 \\ -1 & 0 & -1 & -1 & 3 \\ 4 & 4 & 4 & -1 & -1 \\ -1 & -1 & -1 & 2 & -1 \end{pmatrix}$$

Floyd-Warshall Algorithm vs. Dijkstra's Algorithm

- We can compute all pairs shortest paths by applying Dijkstra's algorithm at each vertex *u* to get shortest paths from u to all other vertices *v*.
- If we implement using the adjacency matrix and an array for priority queue then computing time is

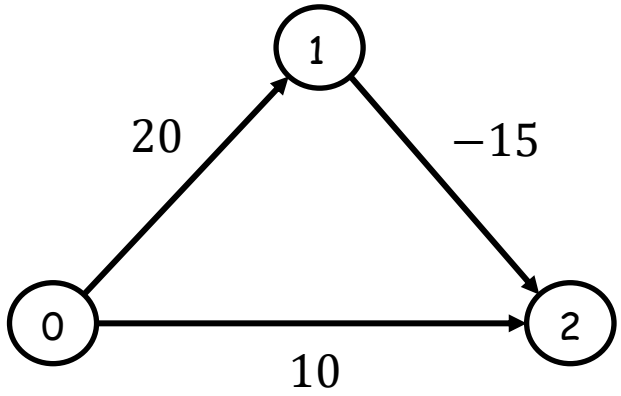
$$O(n^3)$$

  which is the same as Floyd-Warshall algorithm
- If we implement using adjacency lists and a min-heap for the priority queue then computing time is

$$O(mn \log n)$$

  which is better than the Floyd-Warshall algorithm for sparce graphs and digraphs.
- Advantage of the Floyd-Warshall algorithm is that it works for weighted digraphs where negative weights are allowed provided there is no negative cycle.

PSN. Show that Dijkstra fails for the following weighted DAG but the Floyd-Warshall algorithm works.



The shortest distance between two jokes makes a great speech.