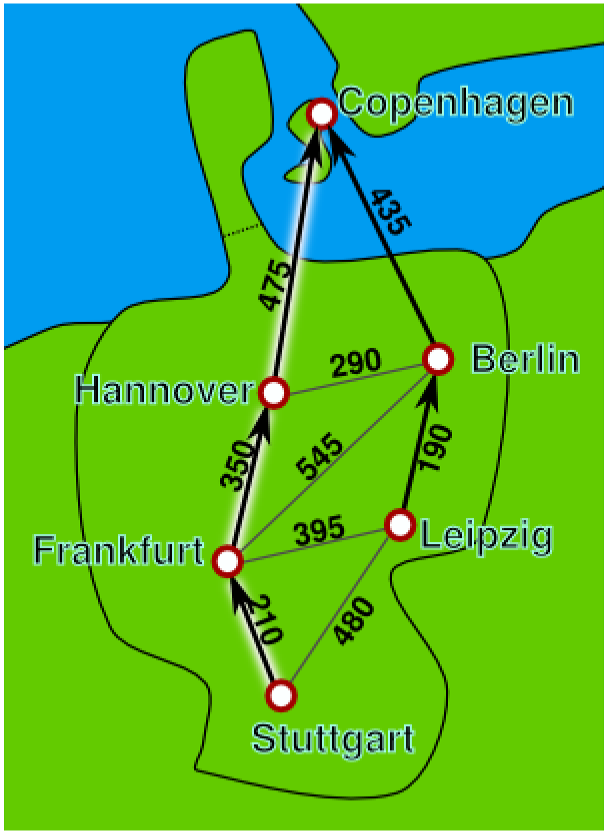
Algorithms and Data Structures

Graphs 2: Shortest Paths

Marius Kloft

Content of this Lecture

- Single-Source-Shortest-Paths: Dijkstra's Algorithm
- Single-Source-Single-Target
- All-Pairs Shortest Paths
  - Transitive closure & unweighted: Warshall's algorithm
  - Negative weights: Floyd's algorithm

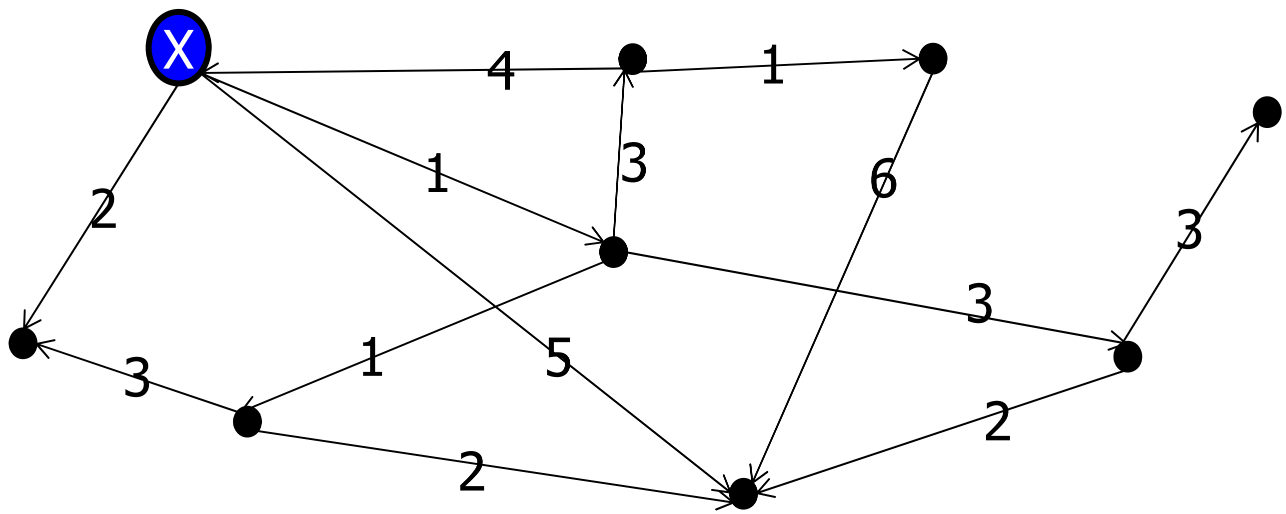


Source: http://beej.us/blog/data/dijkstras-shortest-path/images/dspmap.png

Distance in Graphs

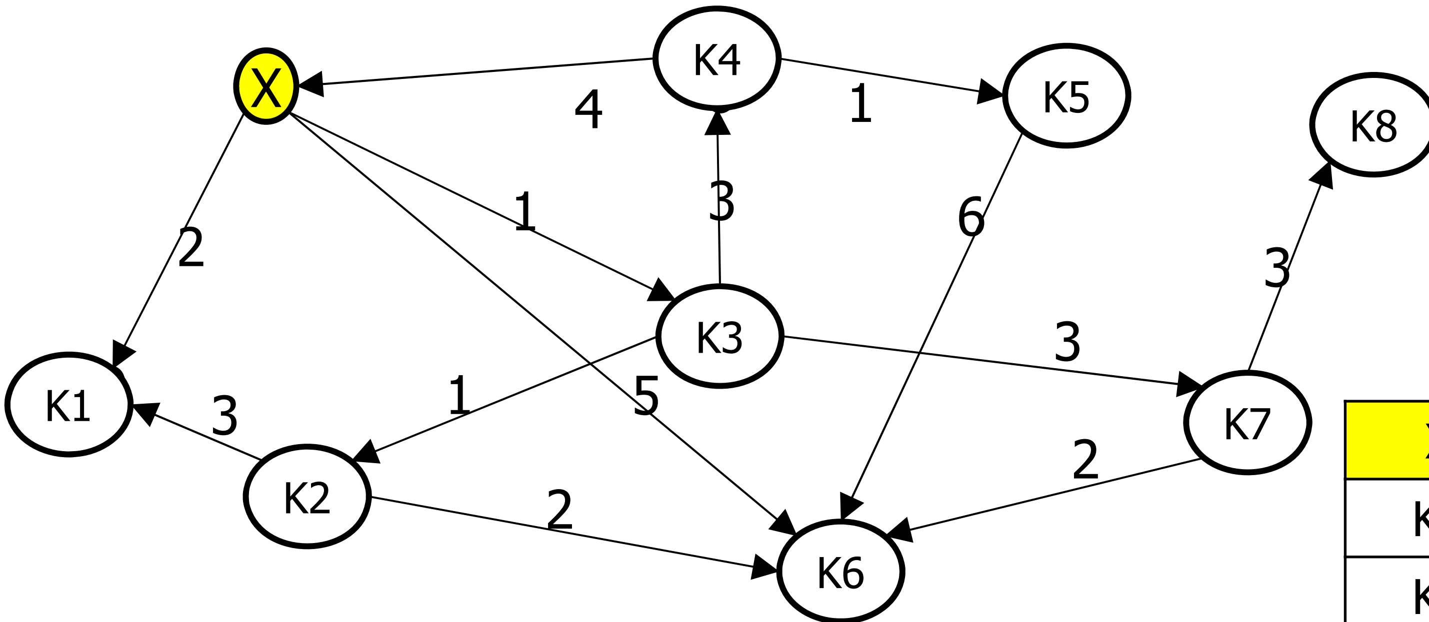
- Definition
  Let G=(V, E) be a graph. The distance d(u,v) between any two nodes u and v from V is defined as
  - *If G is un-weighted: The length of the shortest path from u to v, or ∞ if no path from u to v exists*
  - *If G is weighted: The minimal aggregated edge weight of all non-cyclic paths from u to v, or ∞ if no path from u to v exists*
- Remark
  - Distance in un-weighted graphs is the same as distance in weighted graphs with unit costs
  - Beware of negative cycles in directed graphs

Single-Source Shortest Paths in a Graph



- Task: Find the distance between X and all other nodes
  - Solution: Dijkstra's Algorithm (see Lecture 13 on priority queues)
- Only positive edge weights allowed
  - Bellman-Ford algorithm solves the general case in O(m*n)

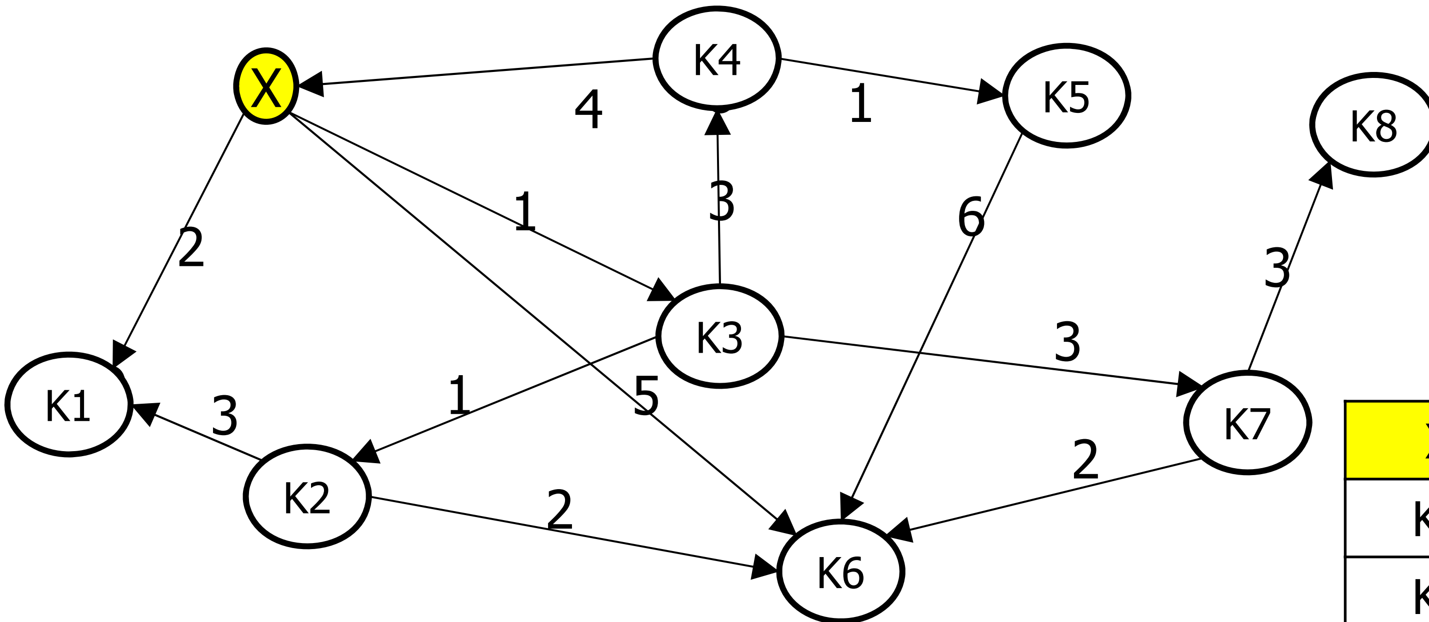
Example for Dijkstra’s Algorithm



- Pick x

X	0
K1	∞
K2	∞
K3	∞
K4	∞
K5	∞
K6	∞
K7	∞
K8	∞

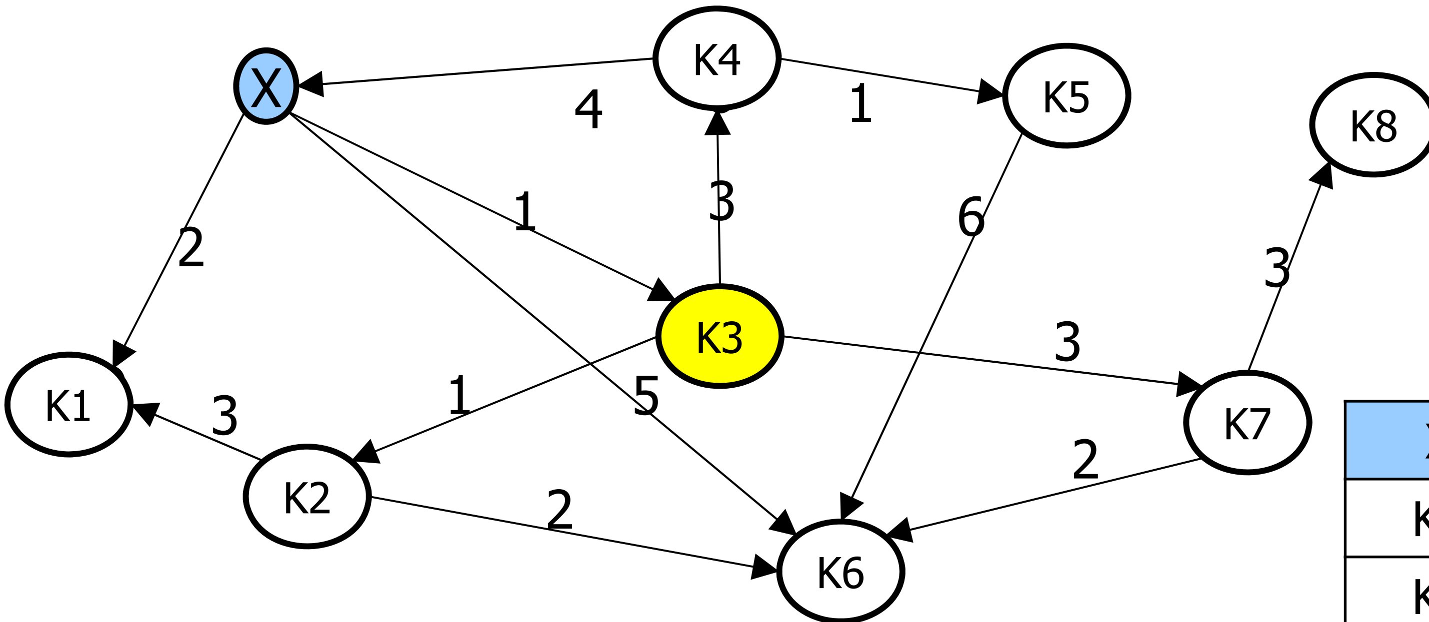
Example for Dijkstra's Algorithm



X	0
K1	2
K2	∞
K3	1
K4	∞
K5	∞
K6	5
K7	∞
K8	∞

- Pick x
- Adapt distances to all neighbors

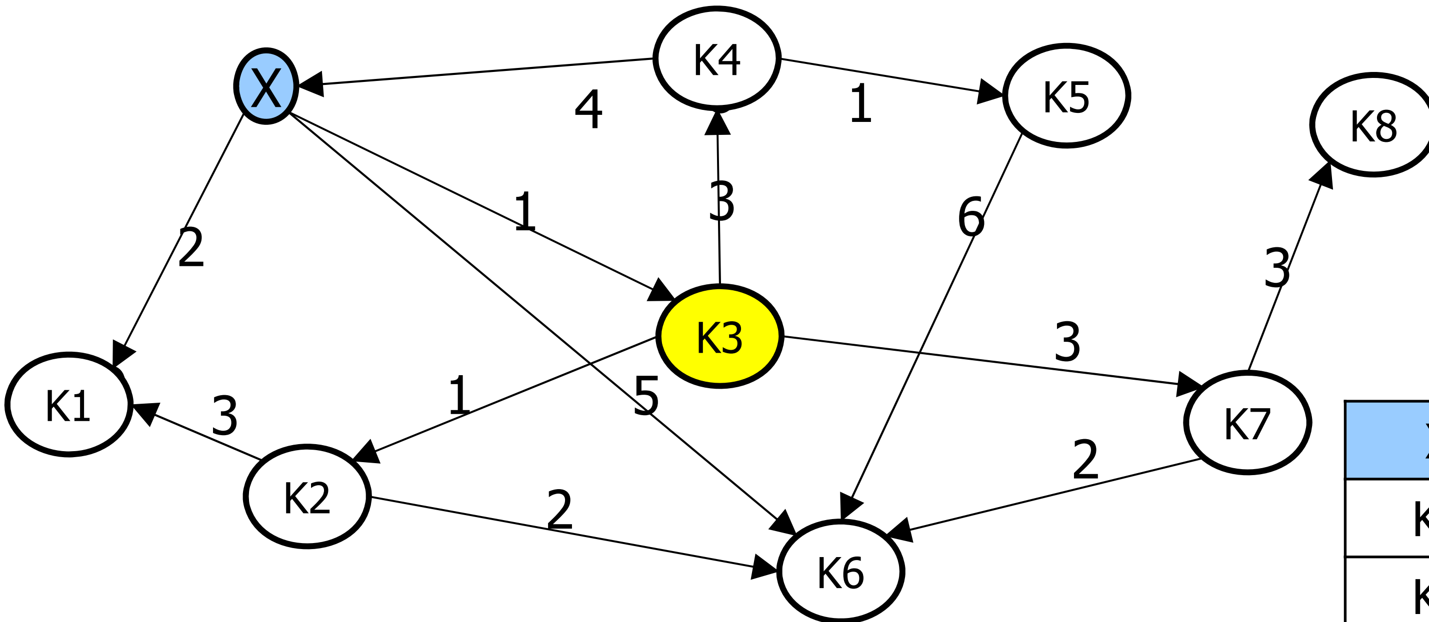
Example for Dijkstra's Algorithm



X	0
K1	2
K2	∞
K3	1
K4	∞
K5	∞
K6	5
K7	∞
K8	∞

- Pick K3 (closest to x)

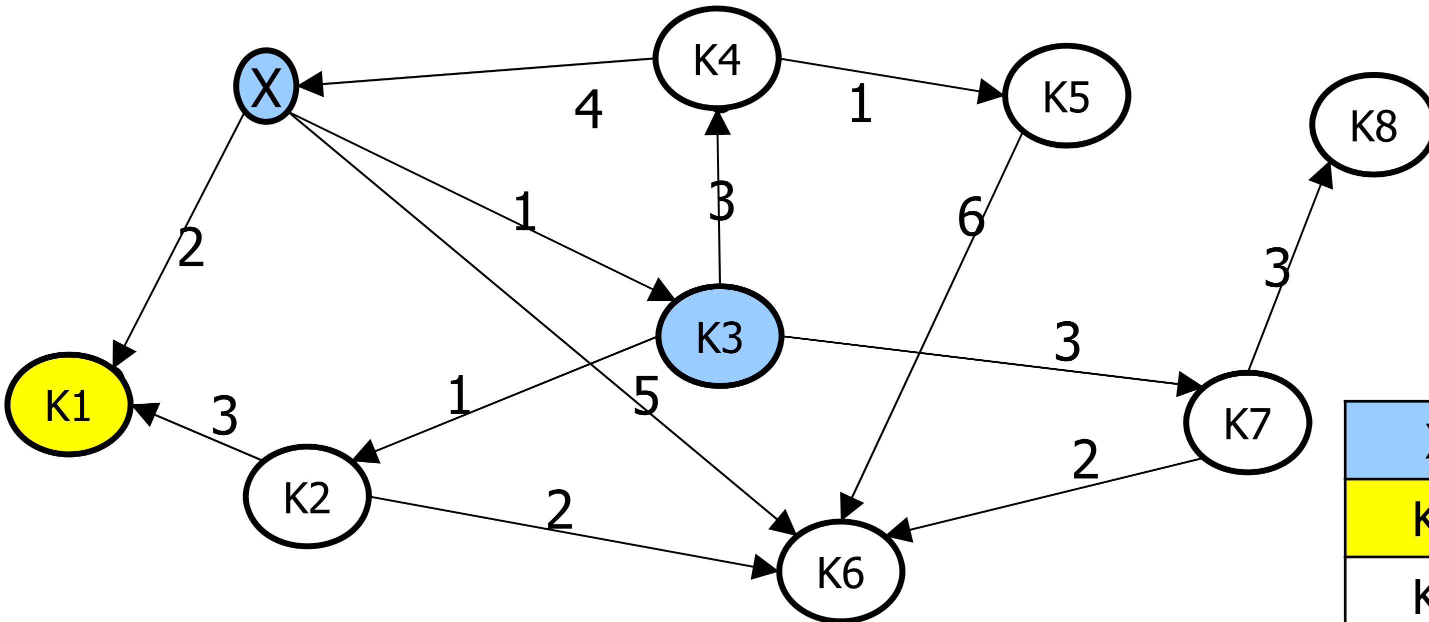
Example for Dijkstra’s Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

- Pick K3
- Adapt distances (from x) to all neighbors (of K3)

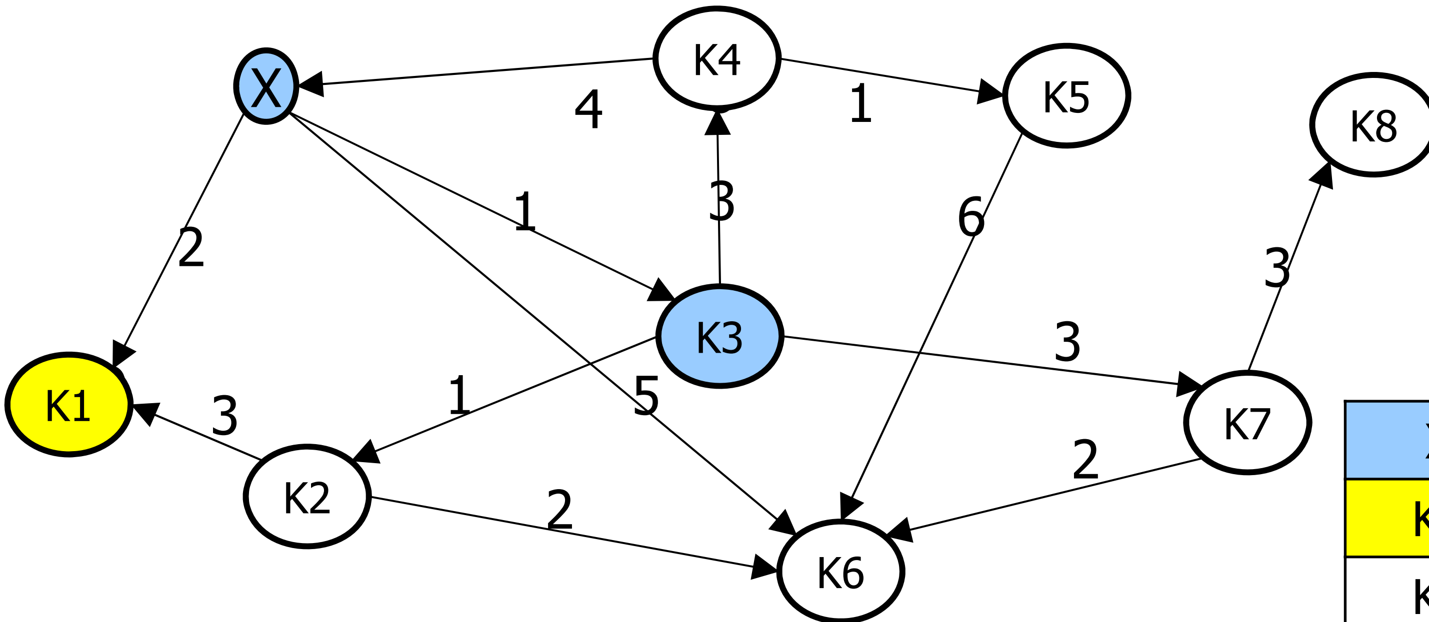
Example for Dijkstra’s Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

- Pick K1 (or K2)

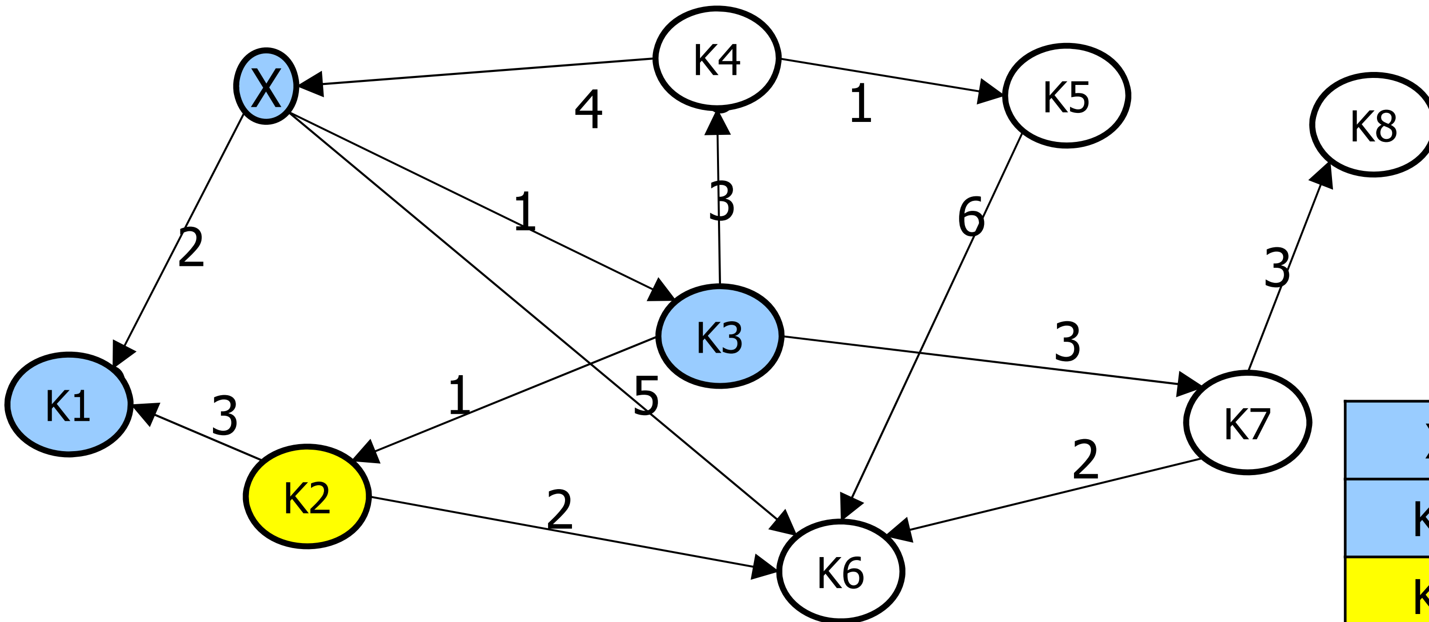
Example for Dijkstra's Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

- Pick K1
- Adapt distances to all neighbors
  - There are none

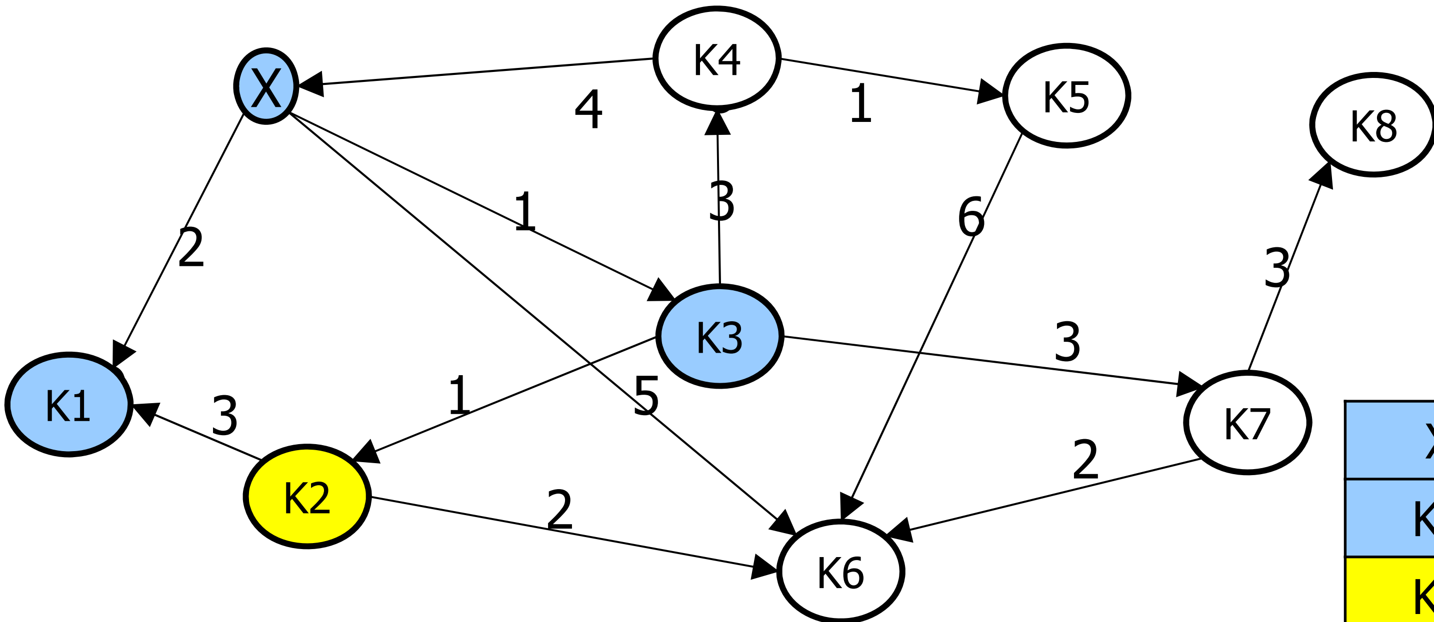
Example for Dijkstra's Algorithm



- Pick K2

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

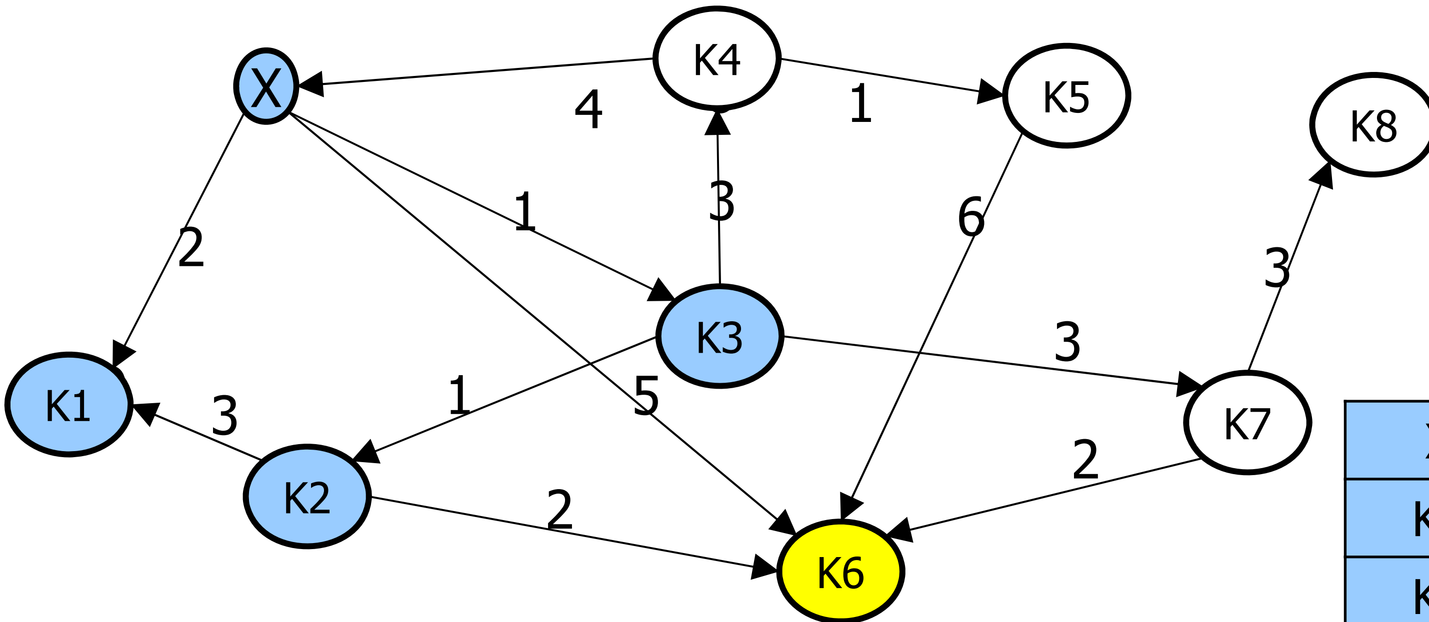
Example for Dijkstra's Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

- Pick K2
- Adapt distances to all neighbors
  - K1 was picked already – ignore
  - We found a shorter path to K6

Example for Dijkstra's Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

- And so on …

Dijkstra's Algorithm – Single Operations

```
1.  G = (V, E);
2.  x : start_node;     # x∈V
3.  A : array_of_distances_from_x;
4.  ∀i: A[i]:= ∞;
5.  L := V;        # organized as PQ
6.  A[x] := 0;
7.  while L≠∅
8.    k := L.get_closest_node();
9.    L := L \ k;
10.   forall (k,f,w)∈E do
11.     if f∈L then
12.       new_dist := A[k]+w;
13.       if new_dist < A[f] then
14.         A[f] := new_dist;
15.         update( L);
16.       end if;
17.     end if;
18.   end for;
19. end while;
```

- Assume a heap-based PQ L
- L holds at most all nodes (n)
- L4: O(n)
- L5: O(n*log(n)) (build PQ)
- L8: O(1) (getMin)
- L9: O(log(n)) (deleteMin)
- L10: with adjacency list O(k) per iteration, O(m) altogether
- L11: O(1)
  - Requires additional array of nodes
- L15: O(log(n)) (updatePQ)

Dijkstra's Algorithm - Loops

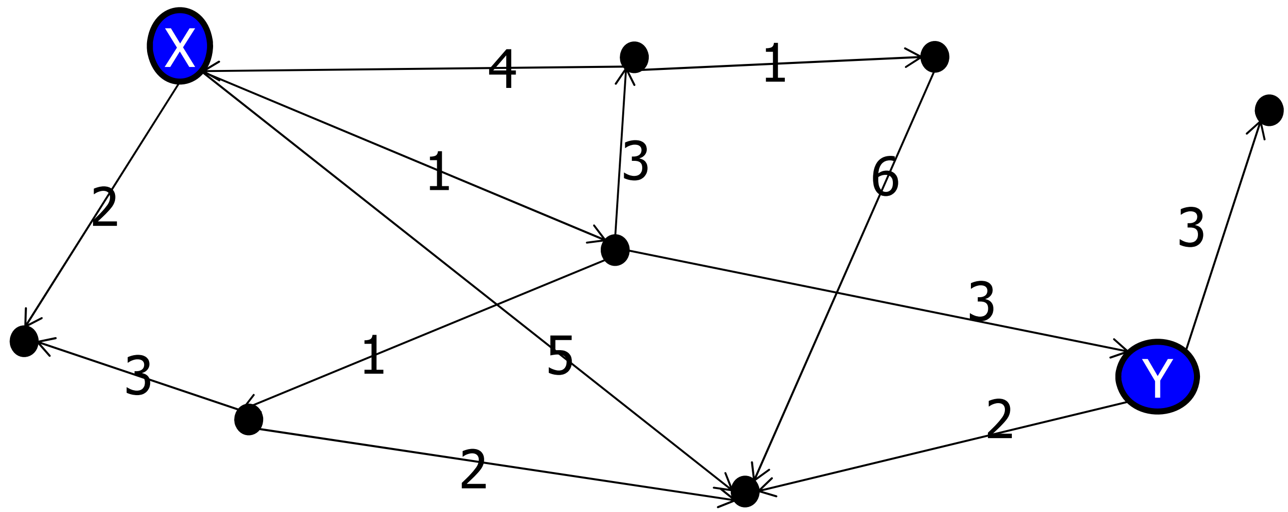
```
1.  G = (V, E);
2.  x : start_node;     # x∈V
3.  A : array_of_distances;
4.  ∀i: A[i]:= ∞;
5.  L := V;       # organized as PQ
6.  A[x] := 0;
7.  while L≠∅
8.    k := L.get_closest_node();
9.    L := L \ k;
10.   forall (k,f,w)∈E do
11.     if f∈L then
12.       new_dist := A[k]+w;
13.       if new_dist < A[f] then
14.         A[f] := new_dist;
15.         update( L);
16.       end if;
17.     end if;
18.   end for;
19. end while;
```

- Loops
  - Lines 7-19: O(n)
  - Line 10-18: All edges exactly once, O(m)
  - Together: O(m+n)
- Central costs
  - L9: O(log(n)) (deleteMin)
  - L15: O(log(n)) (del+ins)
- Altogether: O((n+m)*log(n))

Content of this Lecture

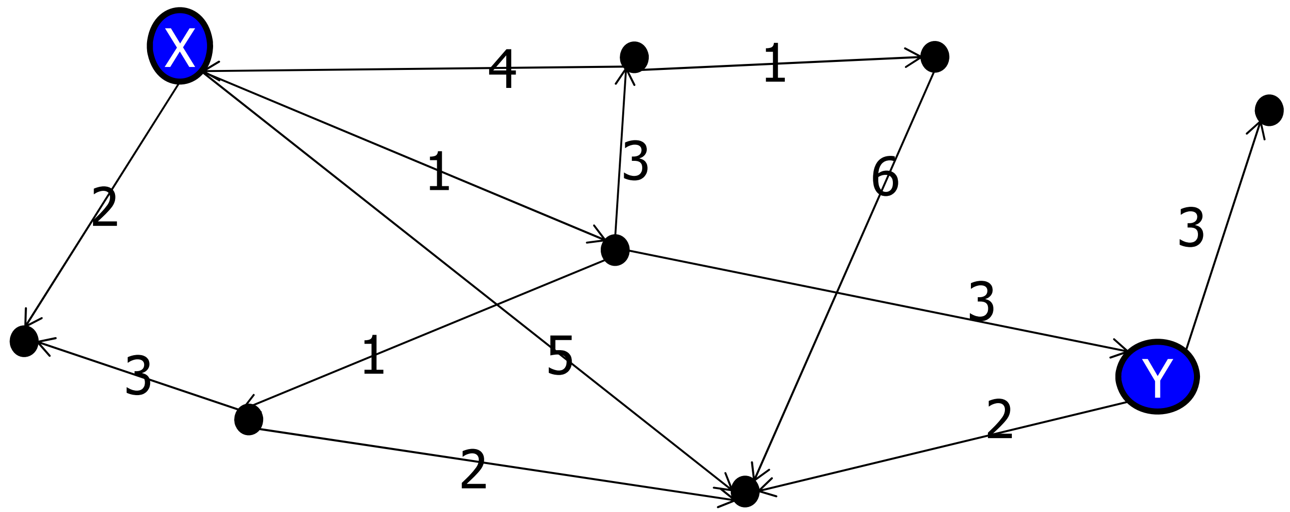
- Single-Source-Shortest-Paths: Dijkstra’s Algorithm
- Single-Source-Single-Target
- All-Pairs Shortest Paths
  - Transitive closure & unweighted: Warshall’s algorithm
  - Negative weights: Floyd’s algorithm

Single-Source, Single-Target



- Task: Find the distance between X and only Y
  - In general, there is no way to be WC-faster than Dijkstra / Bellman-Ford
  - We can stop as soon as Y appears at the min position of the PQ
    - We can visit edges in order of increasing weight
    - Worst-case complexity unchanged, average case is (slightly) better

Single-Source, Single-Target



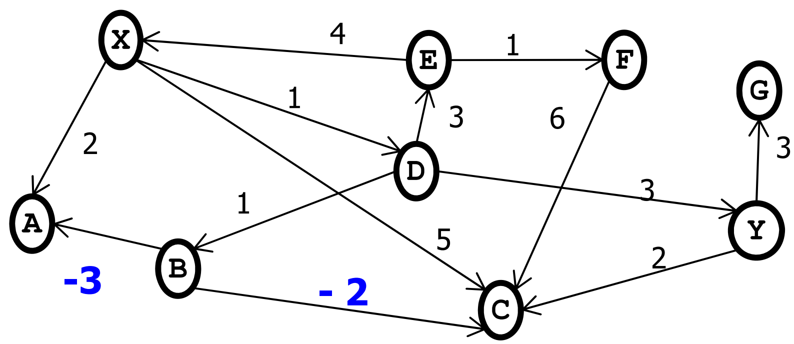
- Things are different in planar graphs: O(n)
  - Henzinger, Monika R., et al. "Faster shortest-path algorithms for planar graphs." *Journal of Computer and System Sciences* 55.1 (1997): 3-23.

Content of this Lecture

- Single-Source-Shortest-Paths: Dijkstra’s Algorithm
- Single-Source-Single-Target
- All-Pairs Shortest Paths
  - Transitive closure & unweighted: Warshall’s algorithm
  - Negative weights: Floyd’s algorithm

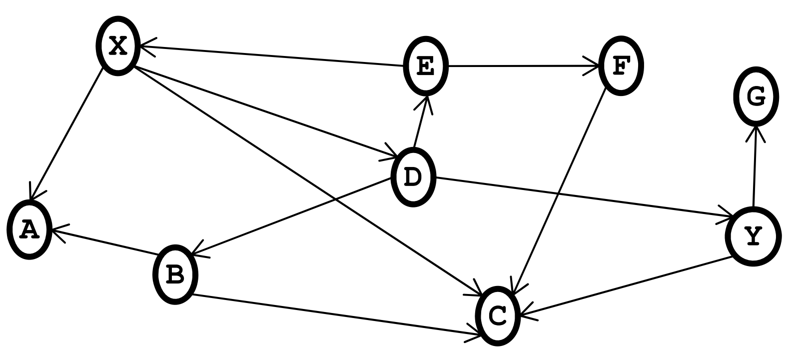
All-Pairs Shortest Paths: General Case

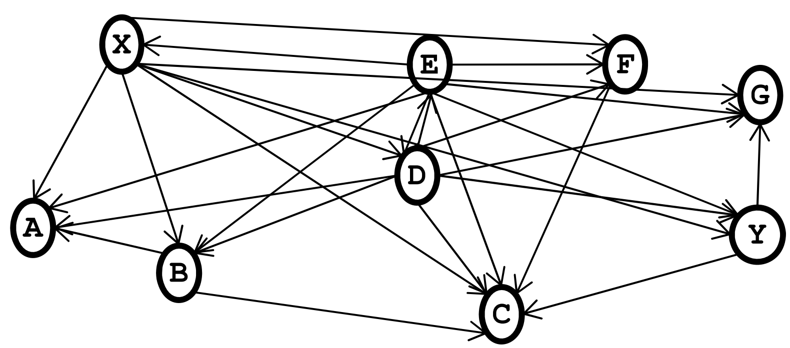
- Given a digraph G with positive or negative edge weights, find the distance between all pairs of nodes
  - Transitive closure with distances



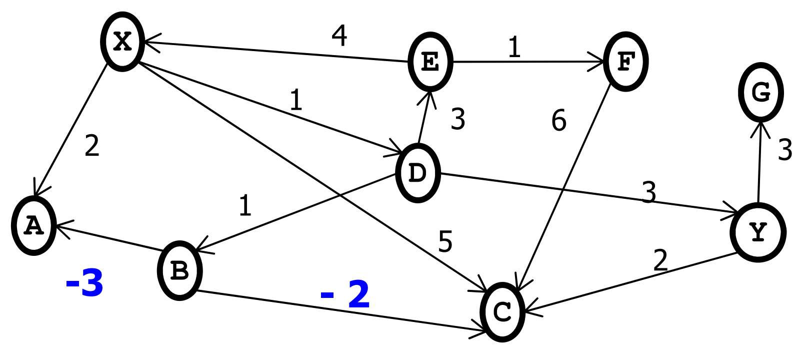
Recall: Transitive Closure

- Definition
  Let G=(V,E) be a digraph and $v_i, v_j \in V$. The transitive closure of G is a graph G'=(V, E') where $(v_i, v_j) \in E'$ iff G contains a path from v_i to v_j.





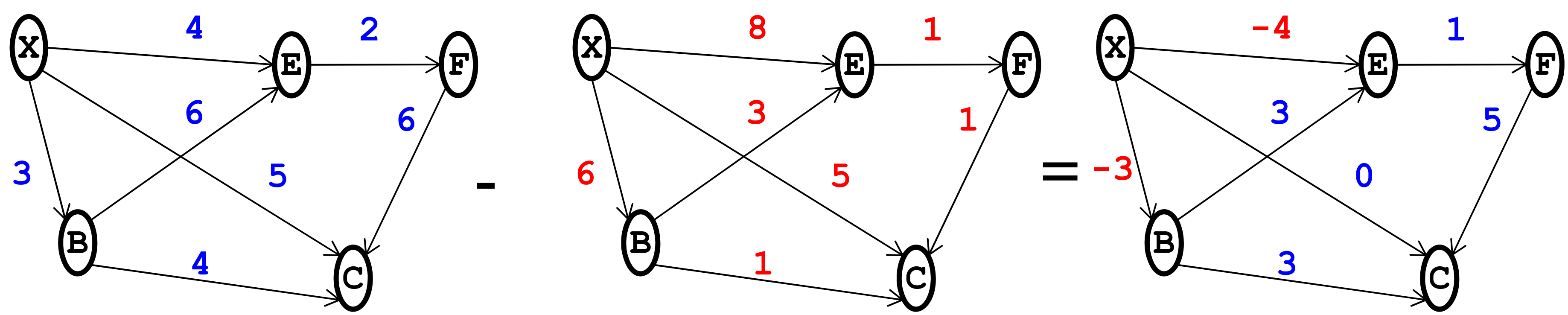
All-Pairs Shortest Paths



- To compute shortest paths for all pairs of nodes, we could n times call a single-source-shortest-path algorithm
  - Positive edge weights: Dijkstra → O(n*(m+n)*log(n))
  - Negative edge weights: Bellman-Ford → $O(m*n^2)$
    - Is $O(n^4)$ for dense graphs
    - Will turn out: Floyd-Warshall solves the general problem in $O(n^3)$

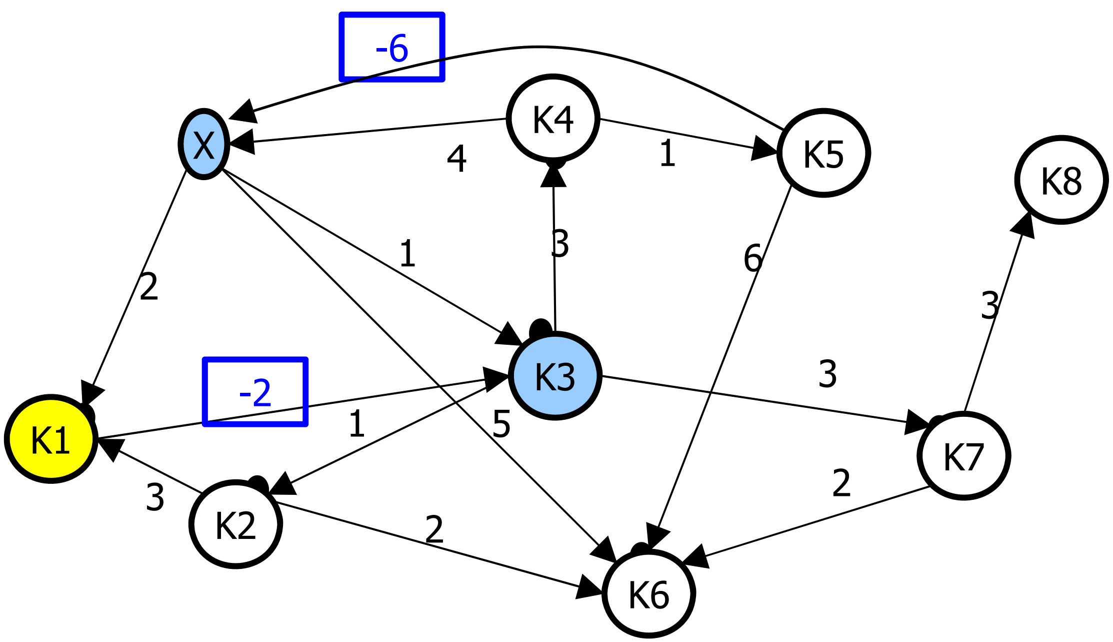
Why Negative Edge Weights?

- One application: Transportation company
  - Every route incurs cost (for fuel, salary, etc.)
  - Every route creates income (for carrying the freight)
- If cost>income, edge weights become negative
  - But still important to find the best route
  - Example: Best tour from X to C



No Dijkstra

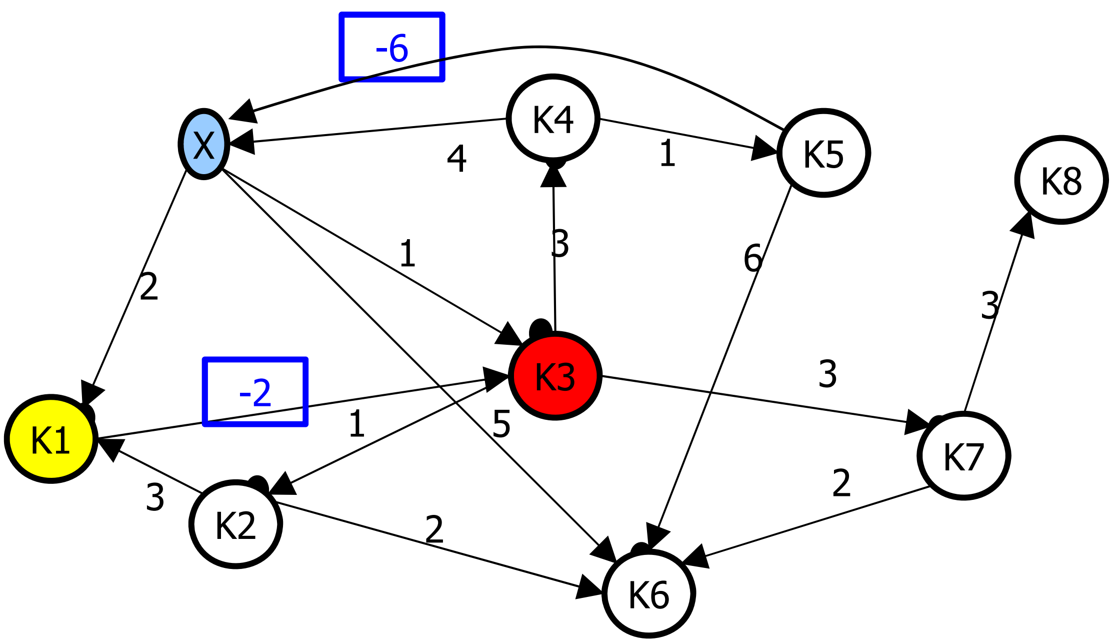
- Dijkstra’s algorithm does not work
  - Recall that Dijkstra enumerates nodes by their shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it “shorter” (by negative edge weights)



X	0
K1	2
K2	2
K3	1
K4	4
K5	
K6	5
K7	4
K8	

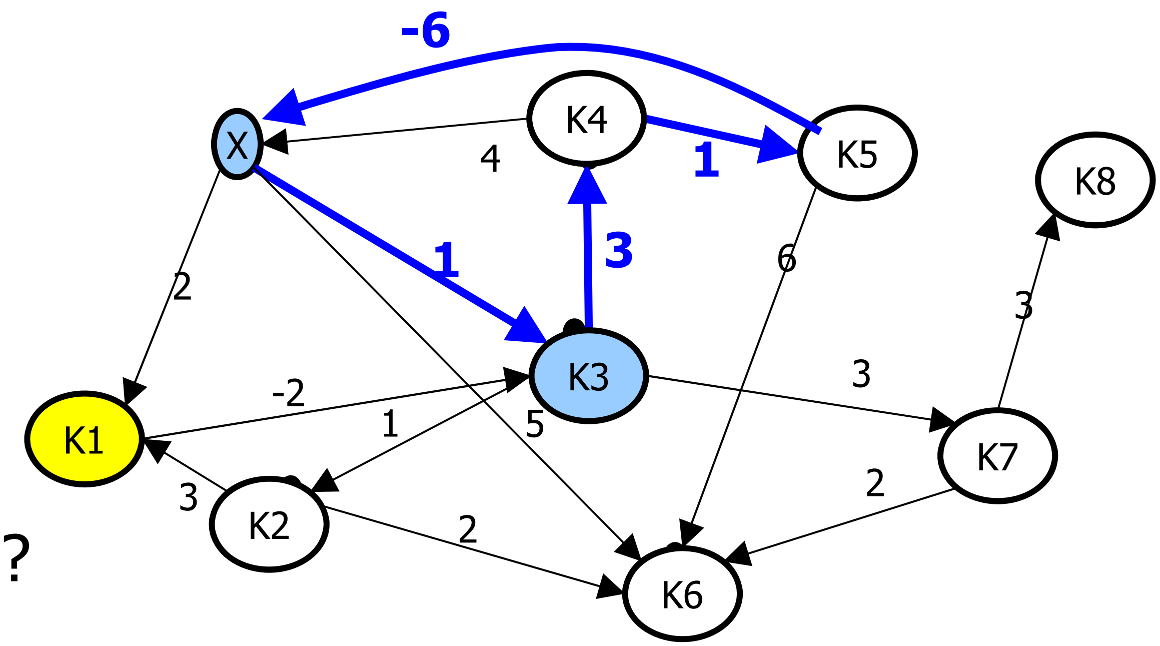
No Dijkstra

- Dijkstra's algorithm does not work
  - Recall that Dijkstra enumerates nodes by their shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it "shorter" (by negative edge weights)



X	0
K1	0
K2	2
K3	0
K4	4
K5	
K6	5
K7	4
K8	

Moreover: Negative Cycles



- Shortest path between X and K5?
  - X-K3-K4-K5: 5
  - X-K3-K4-K5-X-K3-K4-K5: 4
  - X-K3-K4-K5-X-K3-K4-K5-X-K3-K4-K5: 3
  - …
- SP-Problem undefined if G contains a negative cycle

Content of this Lecture

- Single-Source-Shortest-Paths: Dijkstra’s Algorithm
- Single-Source-Single-Target
- All-Pairs Shortest Paths
  - Transitive closure & unweighted: Warshall’s algorithm
  - Negative weights: Floyd’s algorithm

All-Pairs: First Approach

- We start with a simpler problem: Computing the transitive closure of a digraph G without edge weights
  - Solution for negative edge weights will be similar
- First idea
  - Reachability is transitive: $x \to y$ and $y \to z \Rightarrow x \to z$
  - We use this idea to iteratively build longer and longer paths
  - First extend edges with edges – path of length 2
  - Extend those paths with edges – paths of length 3
  - …
  - No necessary path can be longer than $|V|$
- In each step, we store "reachable by a path of length $\leq k$" in a matrix

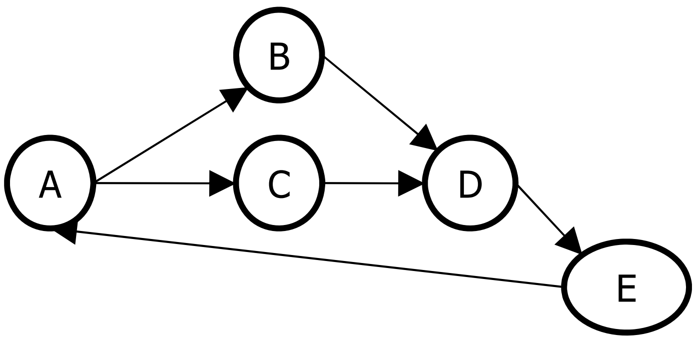
Naïve Algorithm

z appears nowhere; it is there to ensure that we stop when the longest possible shortest paths has been found

```
G = (V, E);
M := adjacency_matrix( G);
M'' := M;
n := |V|;
for z := 1..n-1 do
  M' := M'';
  for i = 1..n do
    for j = 1..n do
      if M'[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M''[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
 end for;
end for;
```

- M is the adjacency matrix of G, M" eventually the TC of G
- M': Represents paths ≤z
- Loops i and j look at all pairs reachable by a path of length at most z+1
- Loop k extends path of length at most z by all outgoing edges
- Analysis: $O(n^4)$

Example – After z=1, 2, 3, 4



	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A		1	1	1	1
B	1			1	1
C	1			1	1
D	1	1	1		1
E	1	1	1	1	

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

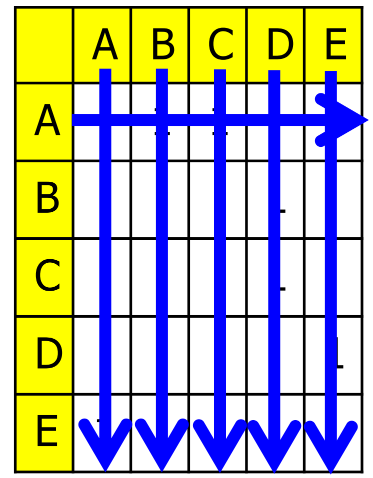
	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

Path length: ≤ 2 ≤ 3 ≤ 4 ≤ 5

Observation

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		



- In the first step, we actually compute M*M, and then replace each value ≥1 with 1
  - We only state that there is a path; not how many and not how long
- Computing TC can be described as matrix operations

Paths in the Naïve Algorithm

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A		1	1	1	1
B	1			1	1
C	1			1	1
D	1	1	1		1
E	1	1	1	1	

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

- The naive algorithm always extends paths by one edge
  - Computes $M*M$, M^2*M, M^3*M, … $M^{n-1}*M$

Idea for Improvement

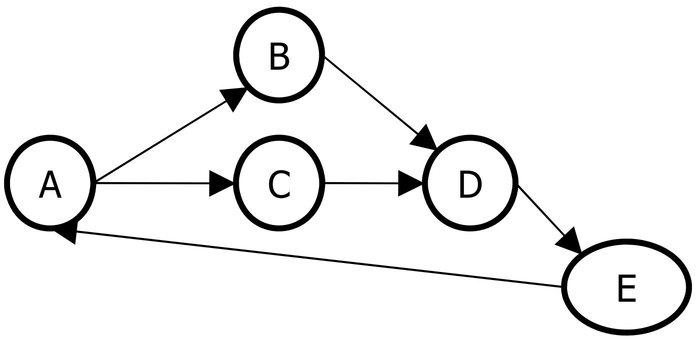
- Why not extend paths by all paths found so-far?
  - We compute
    $M_2=M*M$: Path of length at most 2
    $M_3=M_2*M_2$: Path of length at most 4
    $M_4=M_3*M_3$: Path of length at most 8
    …
    $M_{log(n)+1}=M_{log(n)}*M_{log(n)}$: Path of length at most n
  - [We will implement it differently]
- Trick: We can stop much earlier
  - The longest shortest path can be at most n
  - Thus, it suffices to compute $M_{ceil(log(n))+1}$

Algorithm Improved

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for z := 1..ceil(log(n)) do
  for i = 1..n do
    for j = 1..n do
      if M[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
  end for;
end for;
```

- We use only one matrix M
- In the extension, we see if a path of length $\leq 2^{z-1}$ (stored in M) can be extended by a path of length $\leq 2^{z-1}$ (stored in M) to a path of length 2^z
- Analysis: $O(n^3 * \log(n))$
- But … we still can be faster

Example – After z=1, 2, 3



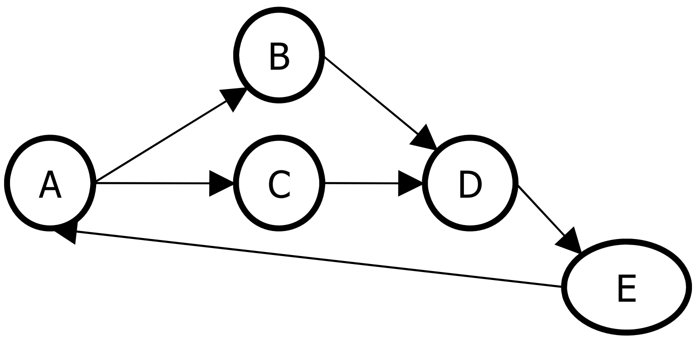
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

Path length: ≤2 ≤4 Done

Further Improvement



	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

- Note: The path A→D is found twice: A→B→D / A→C→D
- Can we stop "searching" A→D once we found A→B→D?
- Can we enumerate paths such that redundant paths are discovered less often (i.e., less paths are tested)?

Warshall's Algorithm

- Warshall, S. (1962). A theorem on Boolean matrices. *Journal of the ACM 9(1): 11-12.*
- Key idea: Enumerate paths by the IDs of the nodes they may use as internal nodes
  - Suppose a path $i \rightarrow k$ and $(i,k) \notin E$
  - Then there must be at least one node j with $i \rightarrow j$ and $j \rightarrow k$
  - Let j be the "smallest" such node (the one with the smallest ID)
  - If we fix the highest allowable ID t, then $i \rightarrow k$ is found iff $j \leq t$
  - Suppose we found all paths consisting only of nodes smaller than t (excluding the edge nodes i,k)
  - We increase t by one, i.e., we allow the usage of node t+1
  - Every new path must have the form $x \rightarrow (t+1) \rightarrow y$

Algorithm

- t gives the highest allowed node ID inside a path
- Thus, node t must be on any new path
- We find all pairs i,k with i→t and t→k
- For every such pair, we set the path i→k to 1

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for t := 1..n do
  for i = 1..n do
    if M[i,t]=1 then
      for k=1 to n do
        if M[t,k]=1 then
          M[i,k] := 1;
        end if;
      end for;
    end if;
  end for;
end for;
```

Proof of Correctness

- Induction: Case t=1 is clear
- Going from t-1 to t
  - Assumption: We know all reachable pairs using as bridges <u>only</u> nodes with ID<t
  - We enter the i-loop
  - L5-6 builds new paths over t
  - L7-11 adds all paths which additionally contain the node with ID t
  - Induction assumption true for t
- These are all paths once t=n

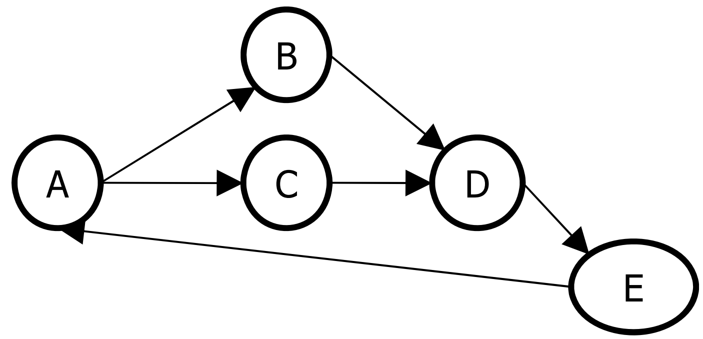
```
1.  G = (V, E);
2.  M := adjacency_matrix( G);
3.  n := |V|;
4.  for t := 1..n do
5.    for i = 1..n do
6.      if M[i,t]=1 then
7.        for k=1 to n do
8.          if M[t,k]=1 then
9.            M[i,k] := 1;
10.         end if;
11.       end for;
12.     end if;
13.   end for;
14. end for;
```

Example – Warshall's Algorithm

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

maxlen=2

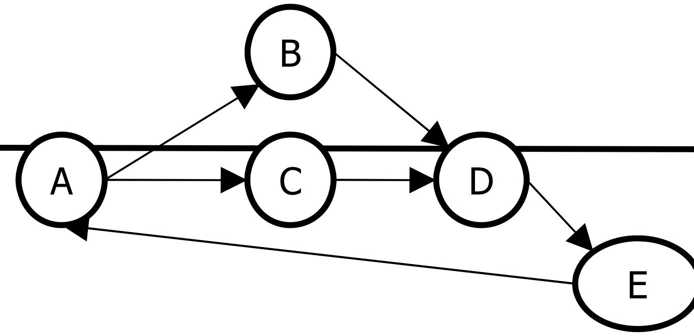
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1	1	1		



A allowed

Connect
E-A with
A-B, A-C

Example – After t=A,B,C,D,E



maxlen=2

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1	1	1		

B allowed

Connect
A-B/E-B
with B-D

=4

	A	B	C	D	E
A		1	1	1	
B				1	
C				1	
D					1
E	1	1	1	1	

C allowed

Connect
A-C/E-C
with C-D

No news

=8

	A	B	C	D	E
A		1	1	1	
B				1	
C				1	
D					1
E	1	1	1	1	

D allowed

Connect
A-D, B-D,
C-D,E-D
with D-E

	A	B	C	D	E
A		1	1	1	1
B				1	1
C				1	1
D					1
E	1	1	1	1	1

E allowed

Connect
everything
with
everything

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

Little change – Consequence: Save a Loop

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for z := 1..ceil(log(n)) do
  for i = 1..n do
    for j = 1..n do
      if M[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
  end for;
end for;
```

Swap i and j loop

Rephrase j into t

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for t := 1..n do
  for i = 1..n do
    if M[i,t]=1 then
      for k=1 to n do
        if M[t,k]=1 then
          M[i,k] := 1;
        end if;
      end for;
    end if;
  end for;
end for;
```

$O(n^3 \log(n))$

$O(n^3)$

Content of this Lecture

- Single-Source-Shortest-Paths: Dijkstra’s Algorithm
- Single-Source-Single-Target
- All-Pairs Shortest Paths
  - Transitive closure & unweighted: Warshall’s algorithm
  - Negative weights: Floyd’s algorithm

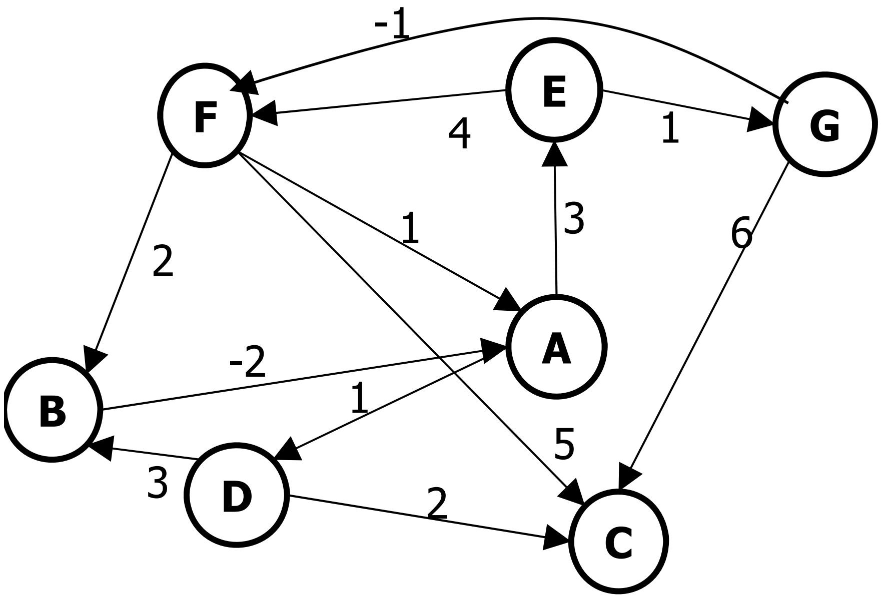
Back to our Original Problem …

… of computing the all-pairs shortest paths for graphs with negative edges:

- We use the same idea: Enumerate paths using only nodes smaller than t
- Invariant: Before step t, M[i,j] contains the length of the shortest path that uses no node with ID higher than t
- When increasing t, we find new paths $i \rightarrow t \rightarrow k$ and look at their lengths
- Thus: $M[i,k] := \min(M[i,k] \cup \{ M[i,t]+M[t,k] \mid i \rightarrow t \wedge t \rightarrow k \})$

Floyd, R. W. (1963). Algorithm 97: Shortest Path. *Communications of the ACM 5(6): 345.*

Example



	A	B	C	D	E	F	G
A				1	3		
B	-2						
C							
D		3	2				
E						4	1
F	1	2	5				
G			6			-1	

	A	B	C	D	E	F	G
A				1	3		
B	-2			-1	1		
C							
D		3	2				
E						4	1
F	1	2	5	2	4		
G			6			-1	

	A	B	C	D	E	F	G
A				1	3		
B	-2			-1	1		
C							
D	1	3	2	2	4		
E						4	1
F	0	2	5	1	3		
G			6			-1	

Summary

- Warshall‘s algorithm computes the transitive closure of any unweighted digraph G in O(|V|^3)
- Floyd‘s algorithm computes the distances between any pair of nodes in a digraph without negative cycles in O(|V|^3)
- Storing both information requires O(|V|^2)
- Problem is easier for …
  - undirected graphs: Connected components (next lecture)
  - graphs with only positive edge weights: All-pairs Dijkstra
  - trees