Algorithms and Data Structures

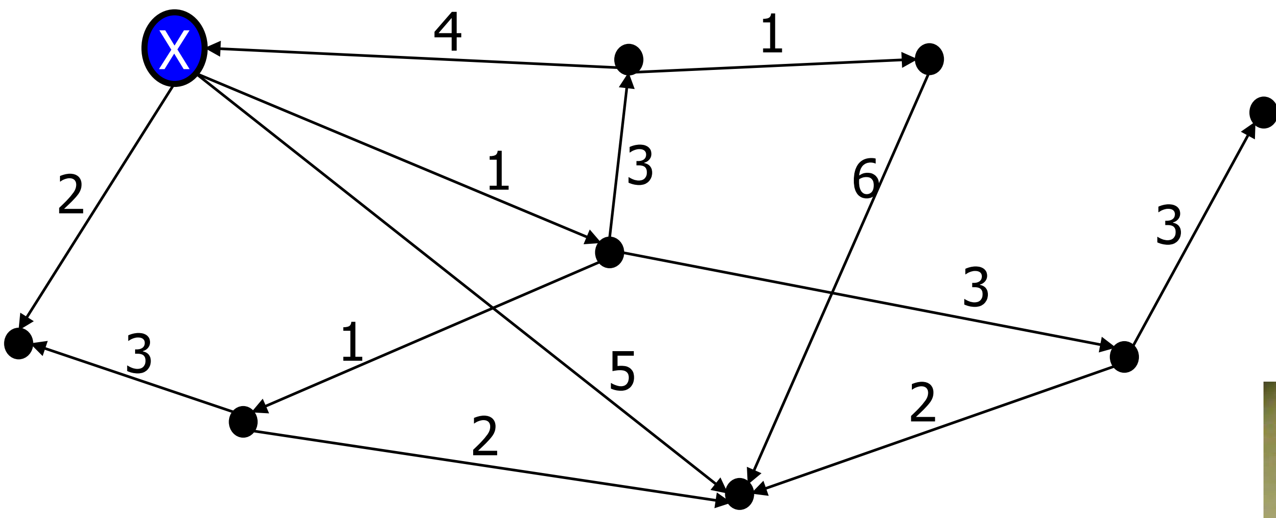
Graphs: Single-Source Shortest Paths

Ulf Leser

Content of this Lecture

- Shortest Paths
  - Single-Source-Shortest-Paths: Dijkstra's Algorithm
  - Shortest Path between two given nodes
  - Other

Shortest Paths in a Graph



- Task: Find the distance between X and all other nodes
  - Classical problem: Single-Source-Shortest-Paths
  - Famous solution: Dijkstra's algorithm
    - E. Dijsktra: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1 (1959), S. 269–271

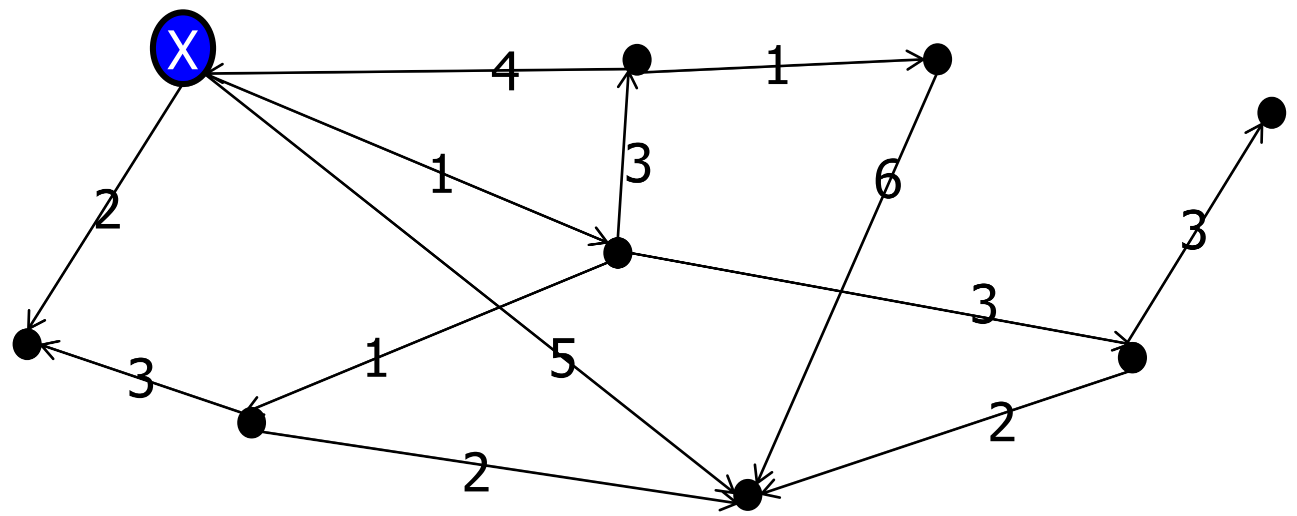
Computer Science is no more about computers than astronomy is about telescopes.

Attributed to Edsger Dijkstra, 1970.

Distance in Graphs

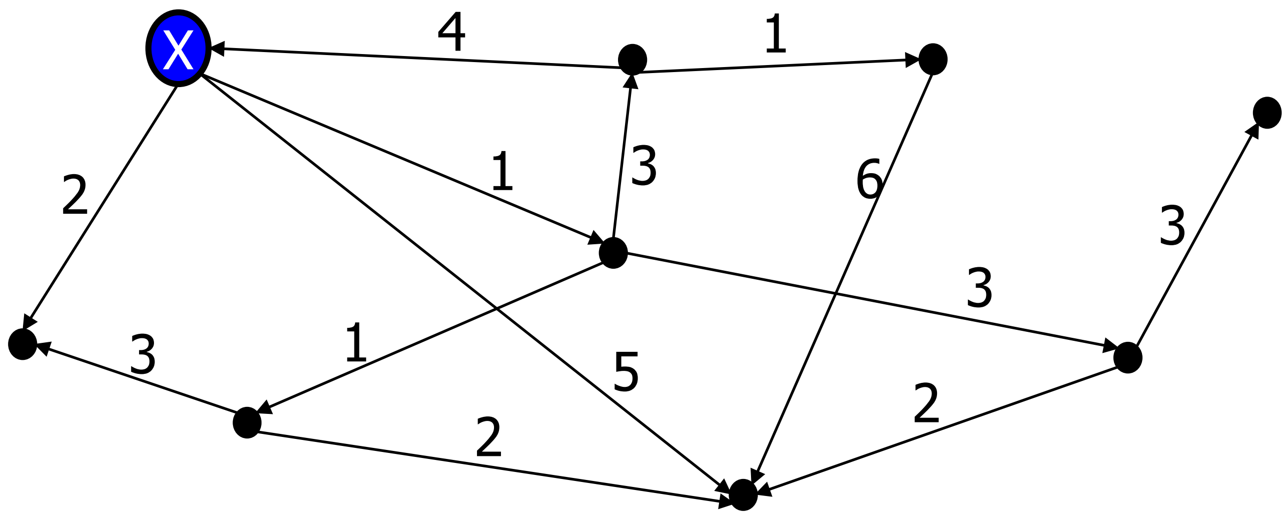
- Definition
  Let $G=(V, E)$ be a graph. The distance $d(u,v)$ between any two nodes $u,v \in V$ for $u \neq v$ is defined as
  - *G unweighted: The length of the shortest path from u to v, or ∞ if no path from u to v exists*
  - *G weighted: The minimal aggregated edge weight of all non-cyclic paths from u to v, or ∞ if no path from u to v exists*
  - *If $u=v$, $d(u,v)=0$*
- Remark
  - Distance in unweighted graphs is the same as distance in weighted graphs with unit cost
  - Beware of negative cycles in directed graphs

Single-Source Shortest Paths in a Graph



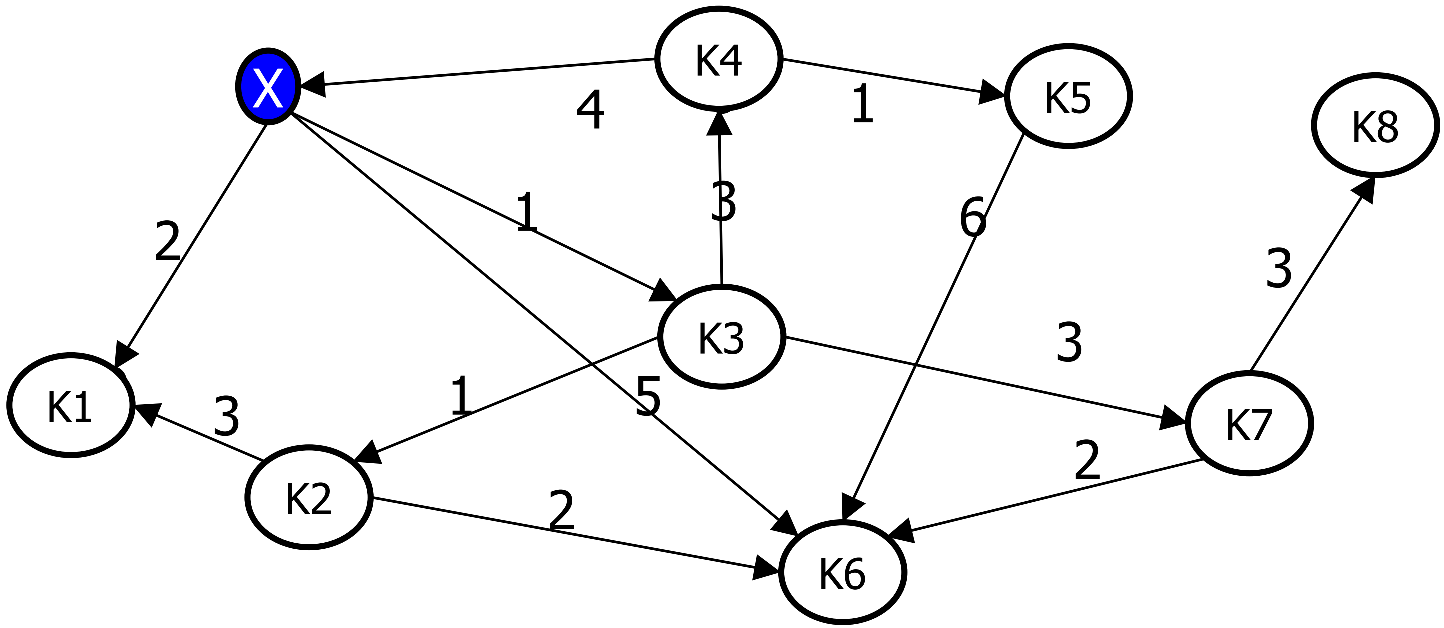
- Task: Find the distance between X and all other nodes
- Only positive edge weights allowed
  - Bellman-Ford algorithm solves the general case
- Floyd-Warshall finds distances between any pair of nodes

Assumptions



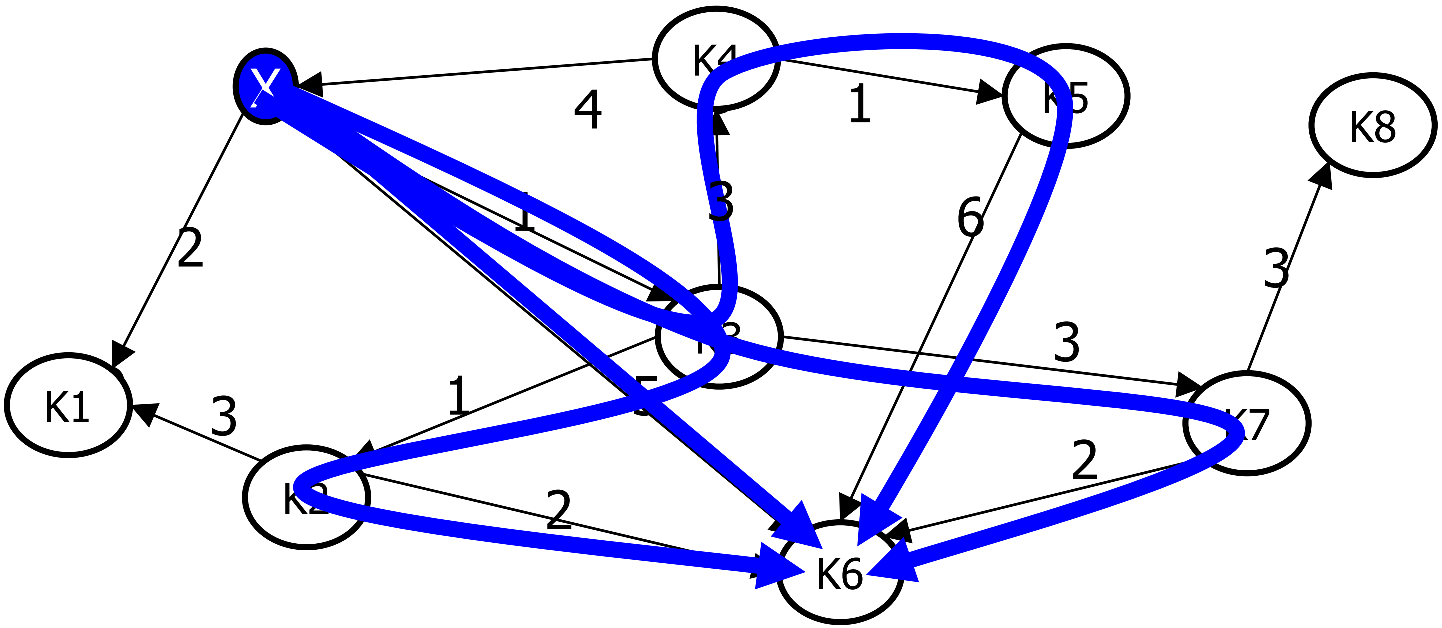
- We assume that every node is reachable from X
- There might be many shortest paths to node Y, but distance is unique
  - We only want the distances and need no "witness paths"
- Only positive edge weights
  - Whenever we extend a path with an edge, its length increases
  - Thus, no shortest path may contain a cycle

Exhaustive Solution



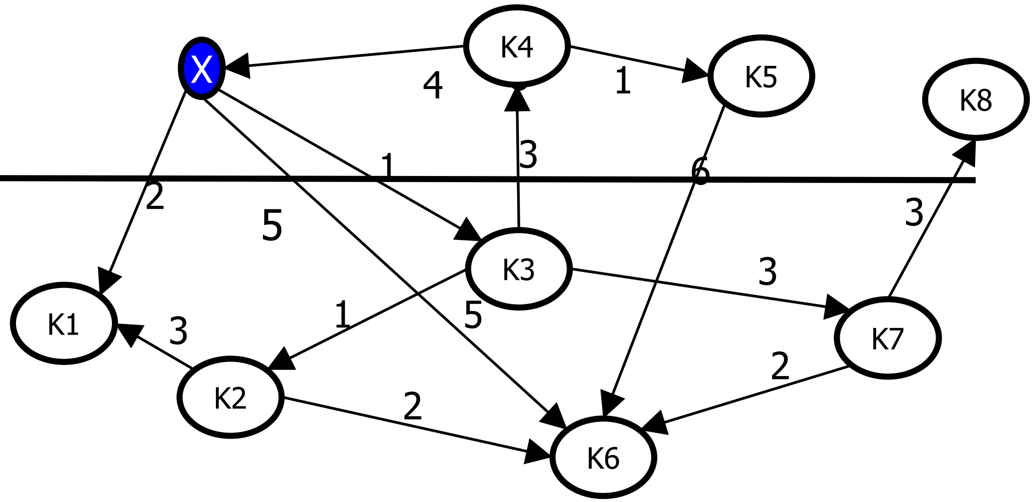
- First approach: Enumerate all paths ("BT": Backtrack)
  - Still need to break cycles (e.g. X – K3 – K4 – X – K3 - …)
  - Using DFS: X – K3 – K4 – X [BT-K4] – K5 – K6 [BT-K5] [BT-K4] [BT-K3] – K7 – K8 [BT-K7] – K6 [BT-K7] [BT-K3] – K2 – K6 [BT-K2] – K1 [BT-K2] [BT-K3] [BT-X] K6 - …

Redundant work



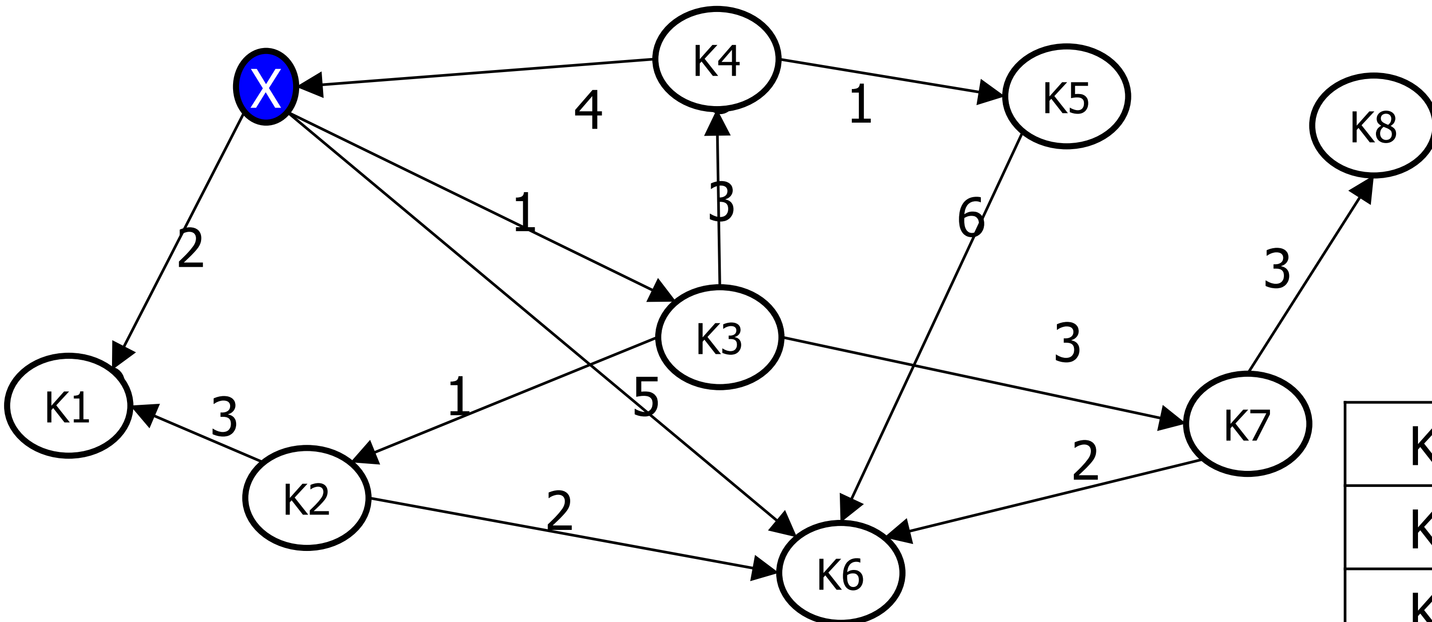
- First approach: Enumerate all paths
  - Need to break cycles (e.g. X – K3 – K4 – X – K3 - …)
  - Using DFS: X – K3 – K4 – X [BT-K4] – K5 – K6 [BT-K5] [BT-K4] [BT-K3] – K7 – K8 [BT-K7] – K6 [BT-K7] [BT-K3] – K2 – K6 [BT-K2] – K1 [BT-K2] [BT-K3] [BT-X] K6 - …

Dijkstra’s Idea



- Enumerate paths from X by their length
  - Neither DFS nor BFS
- Assume we reach a node Y by a path p of length l and we have already explored all paths from X with length l’ < l and that Y was not reached yet
- Then p must be a shortest path between X and Y
  - Because any p’ between X and Y would have a prefix of length at least l and (a) a continuation with length>0 (only positive weights) or (b) would not need a continuation (then p is as short as p’)

Example for Idea



K3	1
K2	2
K1	2
K6	4
K4	4
K7	4
K5	5
K8	7

- 1: X – K3
- 2: X – K3 – K2
  2: X – K1
- 4: X – K3 – K2 – K6
  4: X – K3 – K4
  4: X – K3 – K7
- 5: X – K3 – K4 – K5
- 7: X – K3 – K7 – K8
- Stop (all nodes found)

Algorithmic Idea

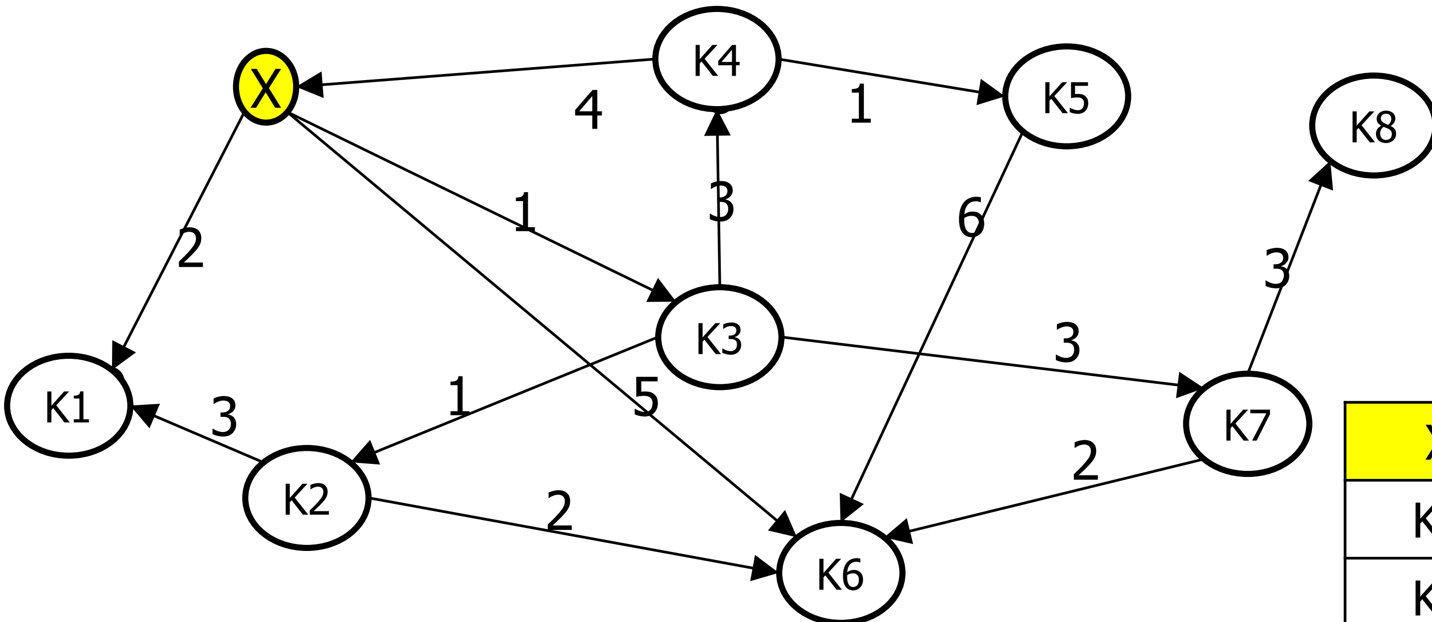
- Enumerate paths by iteratively extending already found shortest paths by all possible extensions
  - All edges outgoing from the end node of a short path
- These extensions
  - … either lead to a node which we didn't reach before – then we found a path, but cannot yet be sure it is the shortest
  - … or lead to a node which we already reached but we are not yet sure of we found the shortest path to it – update current best distance
  - … or lead to a node which we already reached and for which we also surely found a shortest path already – these can be ignored
- Extensions are stored in a priority queue with prio=length
- We enumerate nodes by their distance

Algorithm

```
G = (V, E);
x : start_node;     # x∈V
A : array_of_distances;
∀i: A[i]:= ∞;
L := V;
A[x] := 0;
while L≠∅
  k := L.get_closest_node(x);
  L := L \ k;
  forall (k,f,w)∈E do
    if f∈L then
      new_dist := A[k]+w;
      if new_dist < A[f] then
        A[f] := new_dist;
      end if;
    end if;
  end for;
end while;
```

- Assumptions
  - Nodes have IDs between 1 … |V|
  - Edges are `(from, to, weight)`
- We enumerate nodes by length of their shortest paths
  - In the first loop, we pick x and update distances (A) to all adjacent nodes
  - When we pick a node k, we already have computed its distance to x in A
  - We adapt the current best distances to all neighbors of k we haven't picked yet
- Once we picked all nodes, we are done

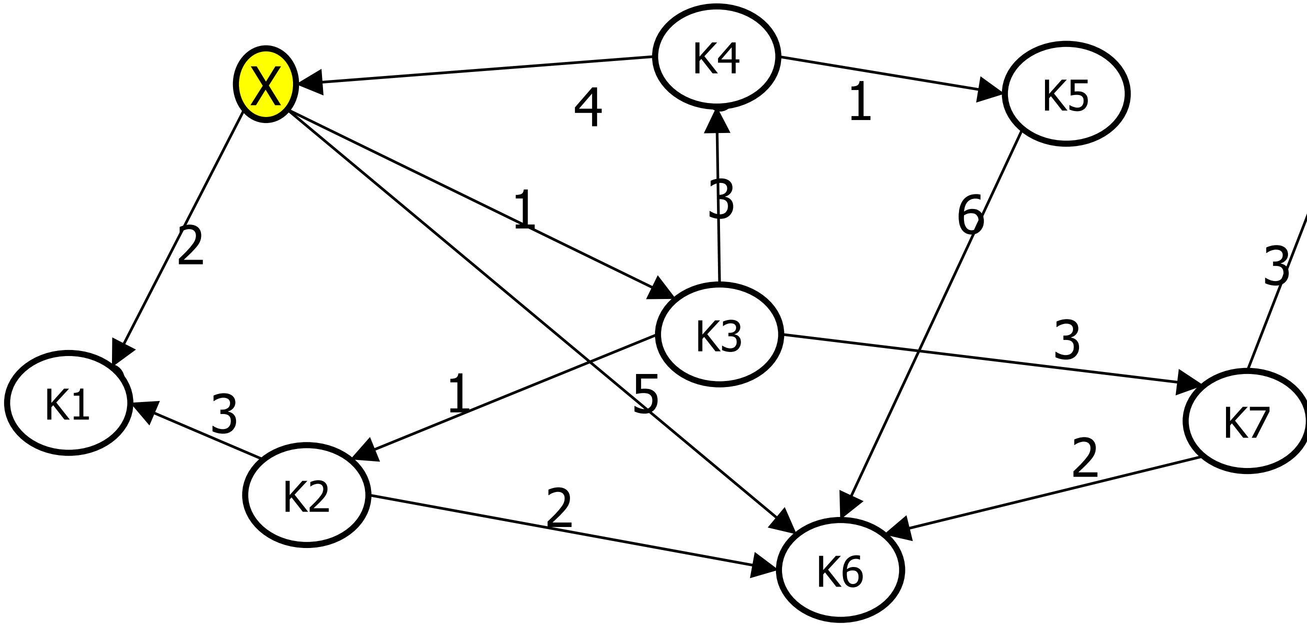
Example for Algorithm



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

- Pick x

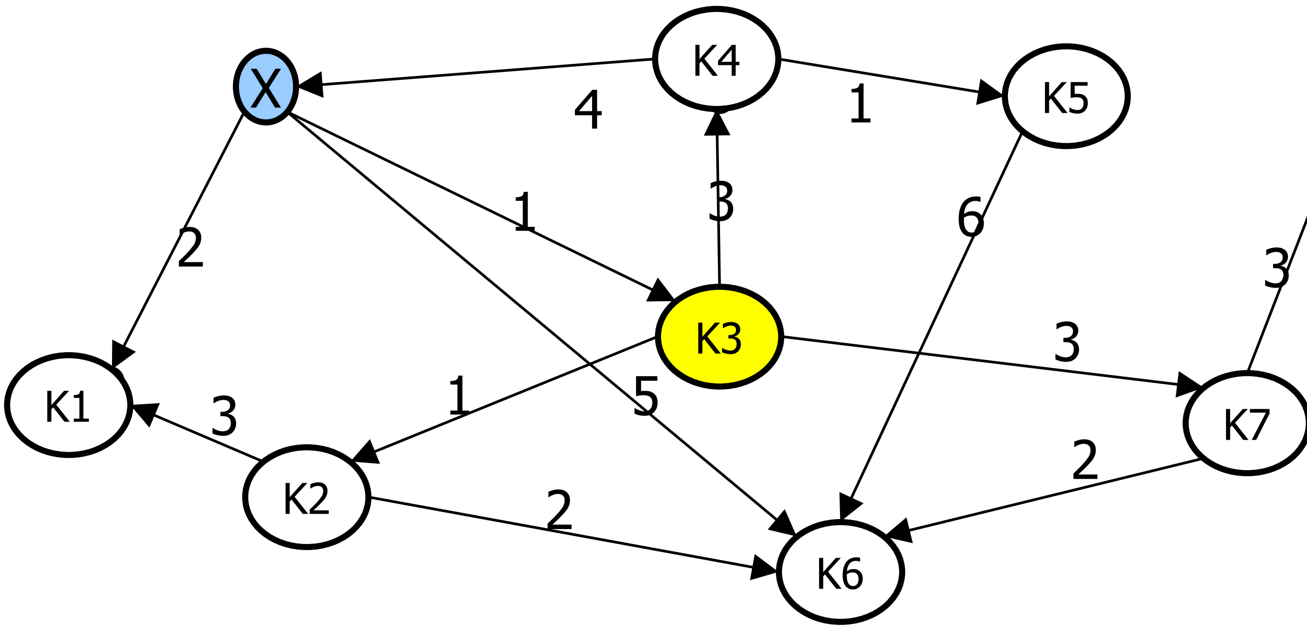
Example for Algorithm



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

- Pick x
- Adapt distances to all neighbors

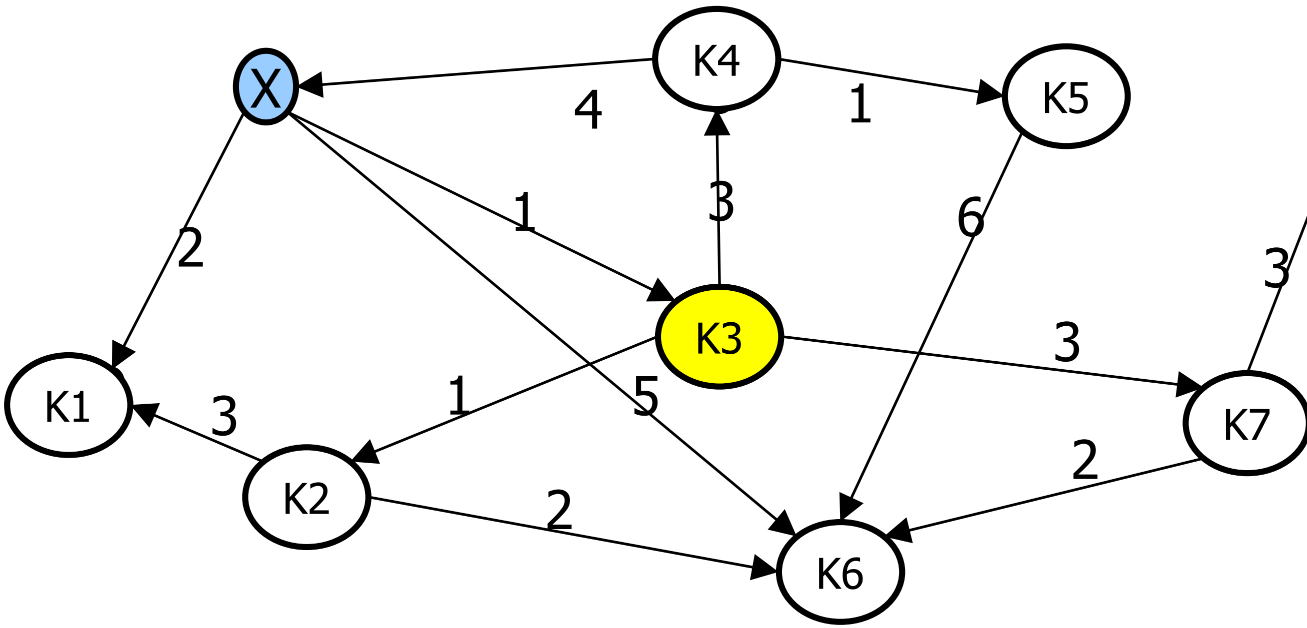
Example for Algorithm



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

- X is done – remove from L
- Pick K3 (closest to x)

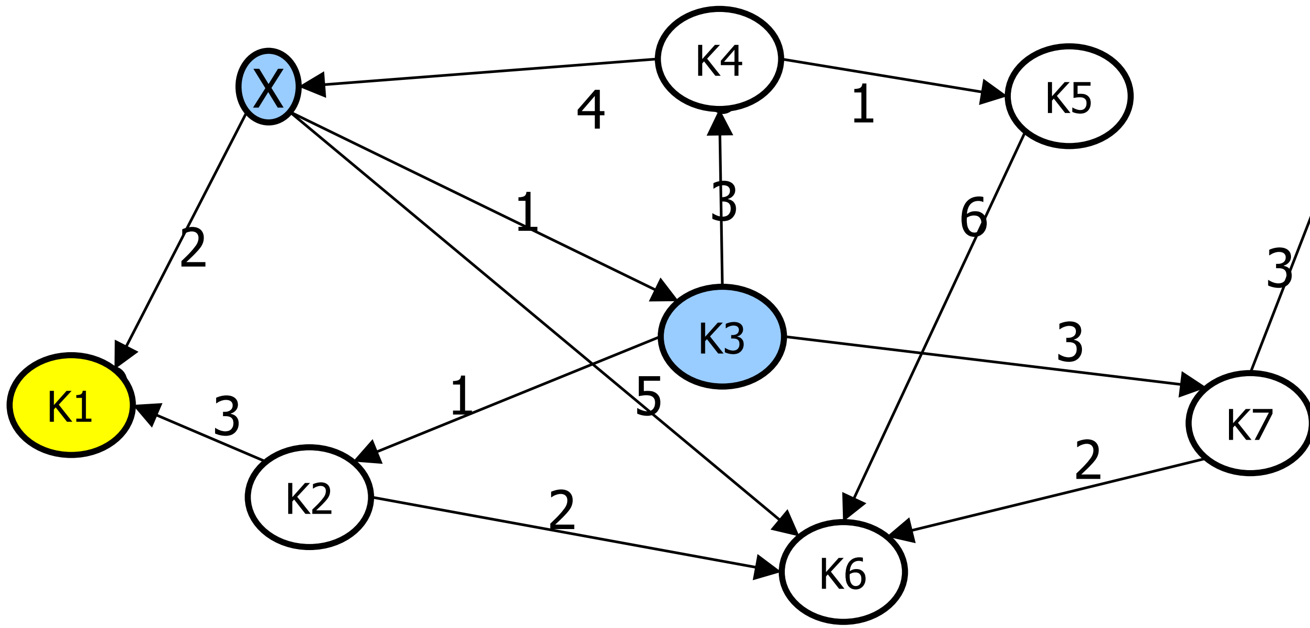
Example for Algorithm



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)

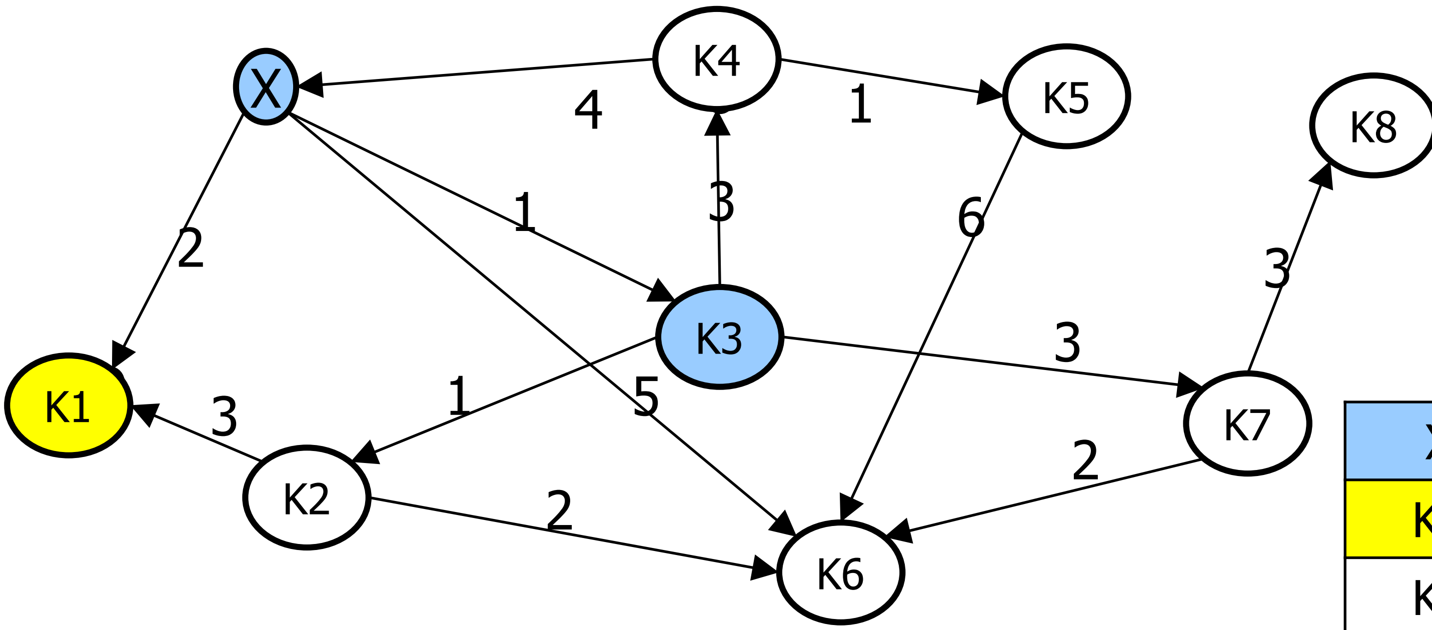
Example for Algorithm



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

- K3 is done (we cannot find a shorter path)
- Pick K1 (or K2)

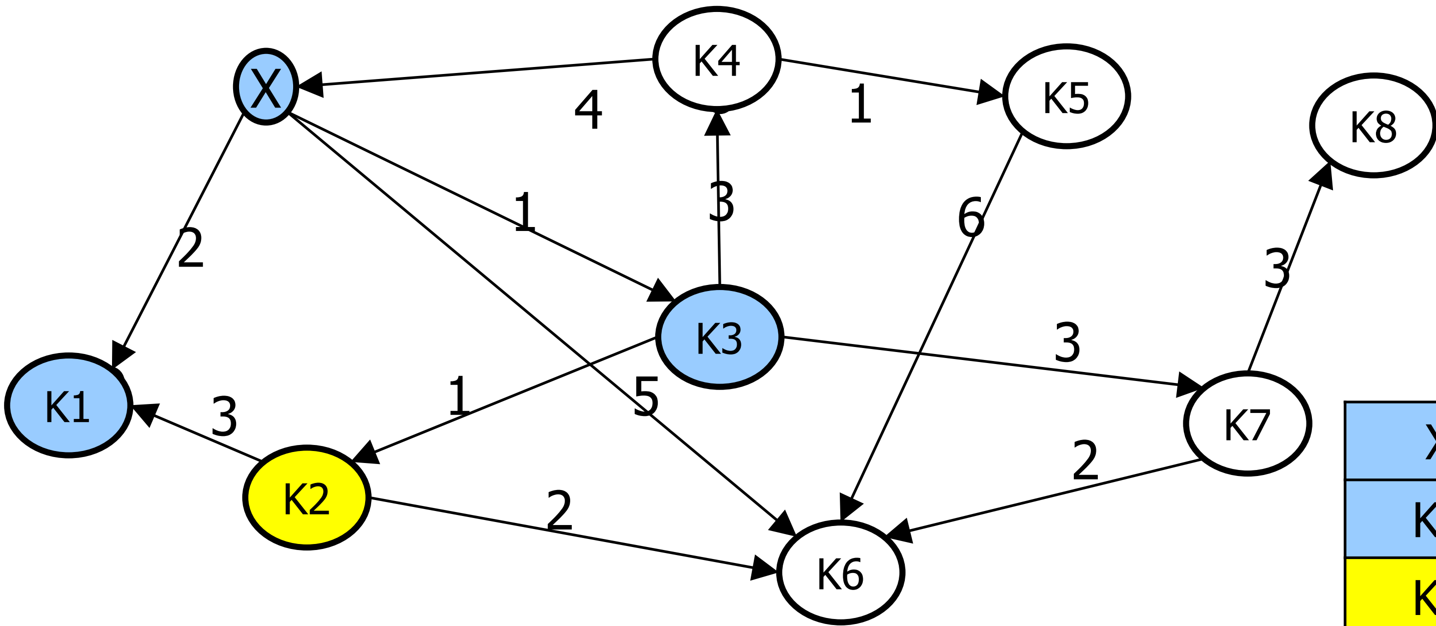
Example for Algorithm



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

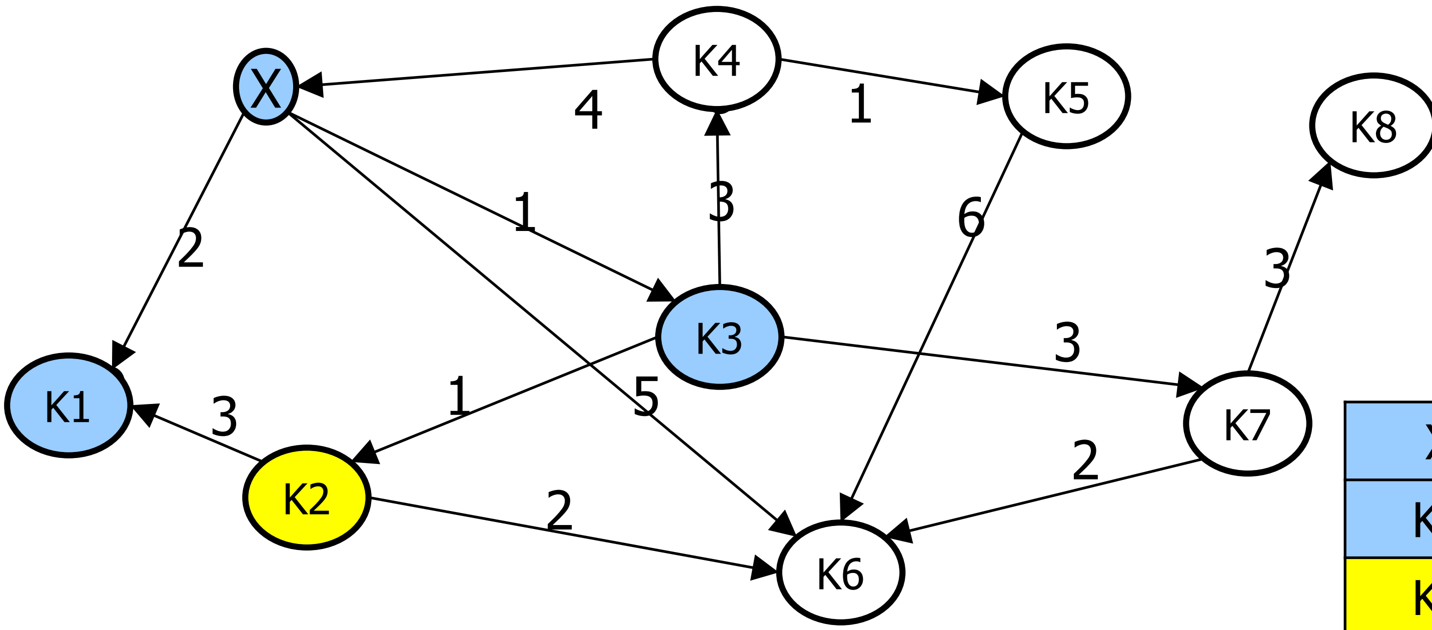
Example for Algorithm



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

- K1 is done
- Pick K2

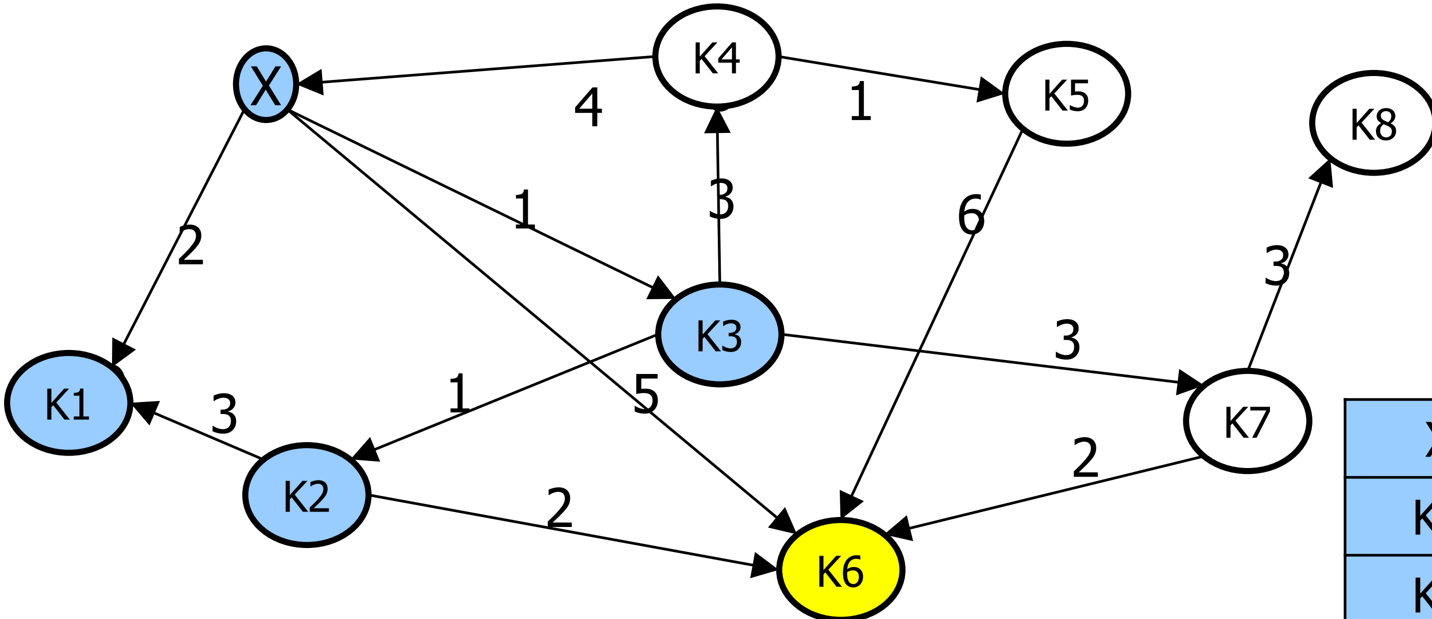
Example for Algorithm



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

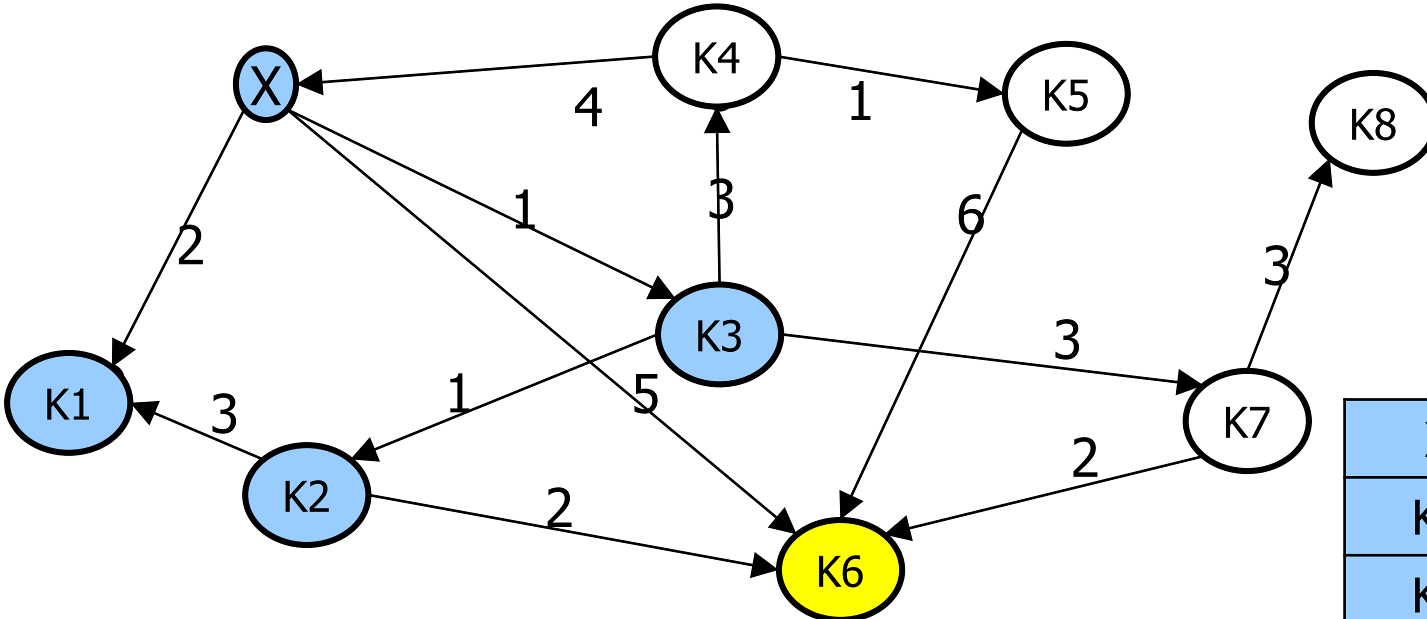
Example for Algorithm



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

- Pick K6 (or K4 or K7)

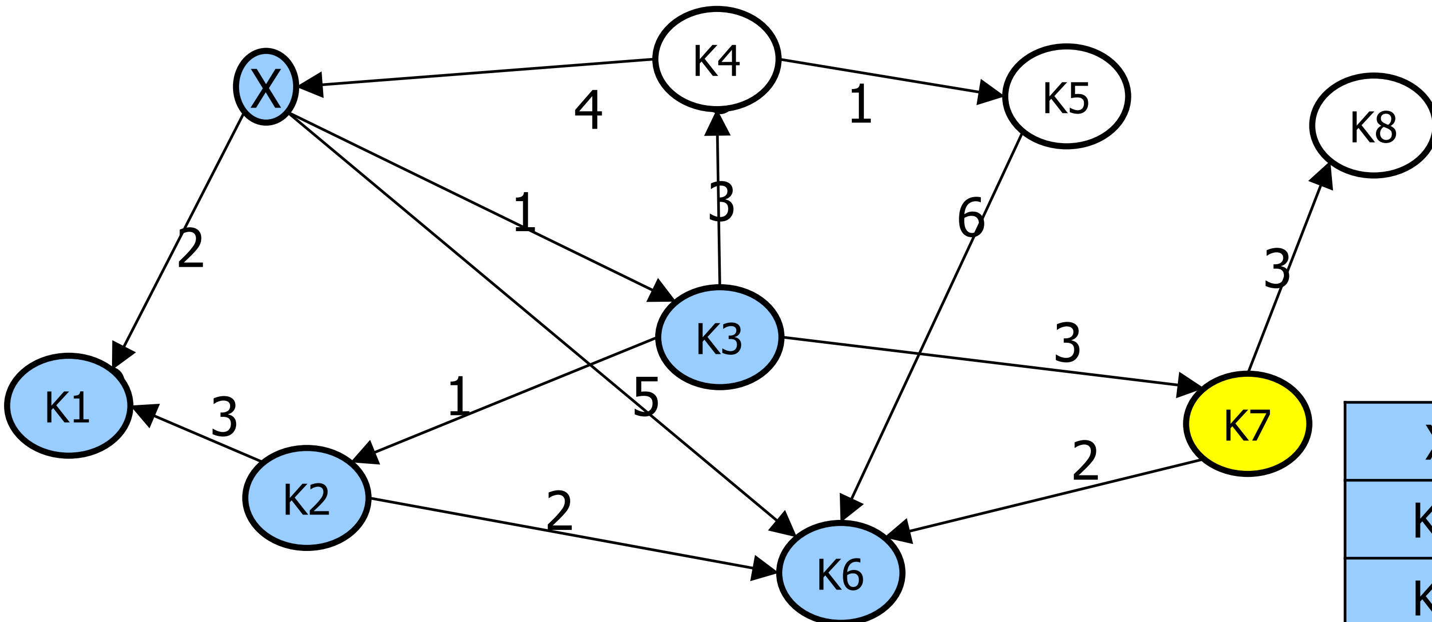
Example for Algorithm



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

- Pick K6
- Adapt distances to all neighbors
  - There are none

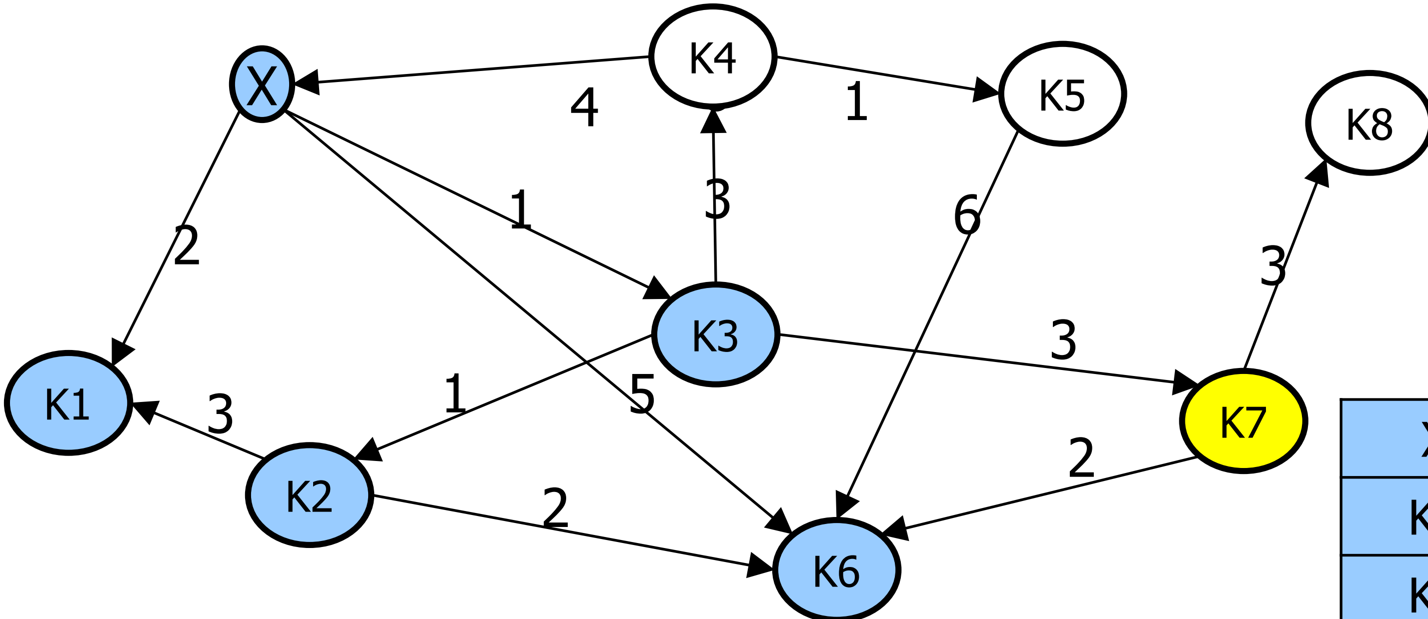
Example for Algorithm



- Pick K7

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

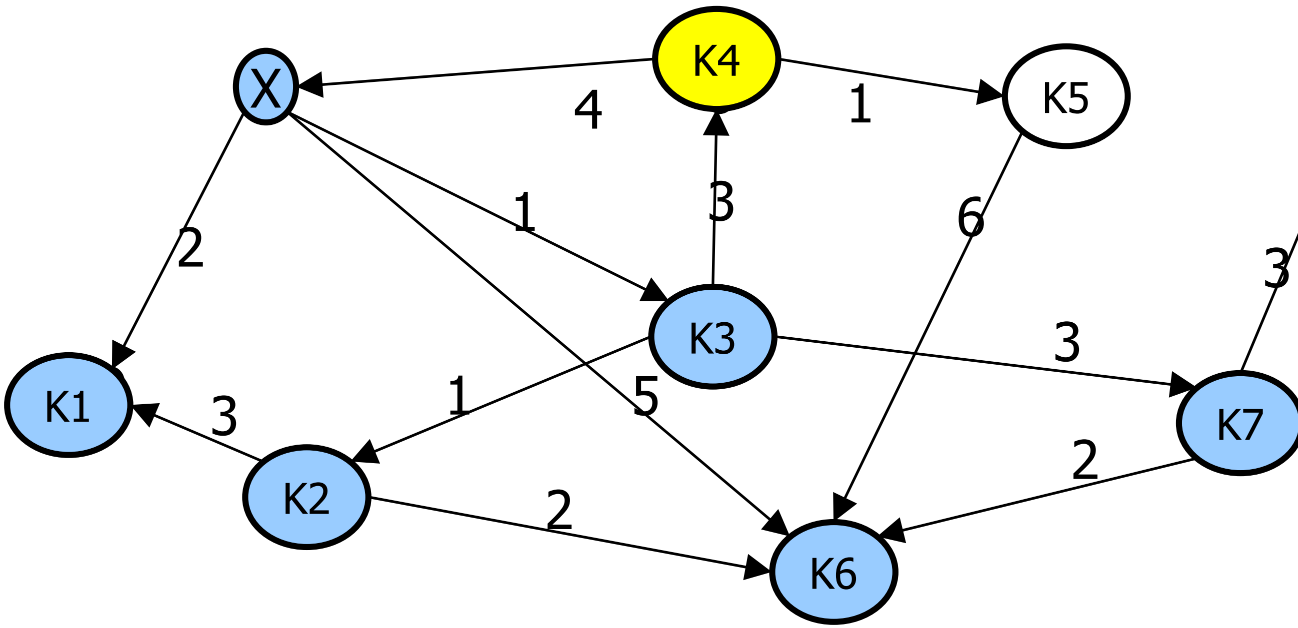
Example for Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	7

- Pick K7
- Adapt distances to all neighbors
  - K6 was visited already

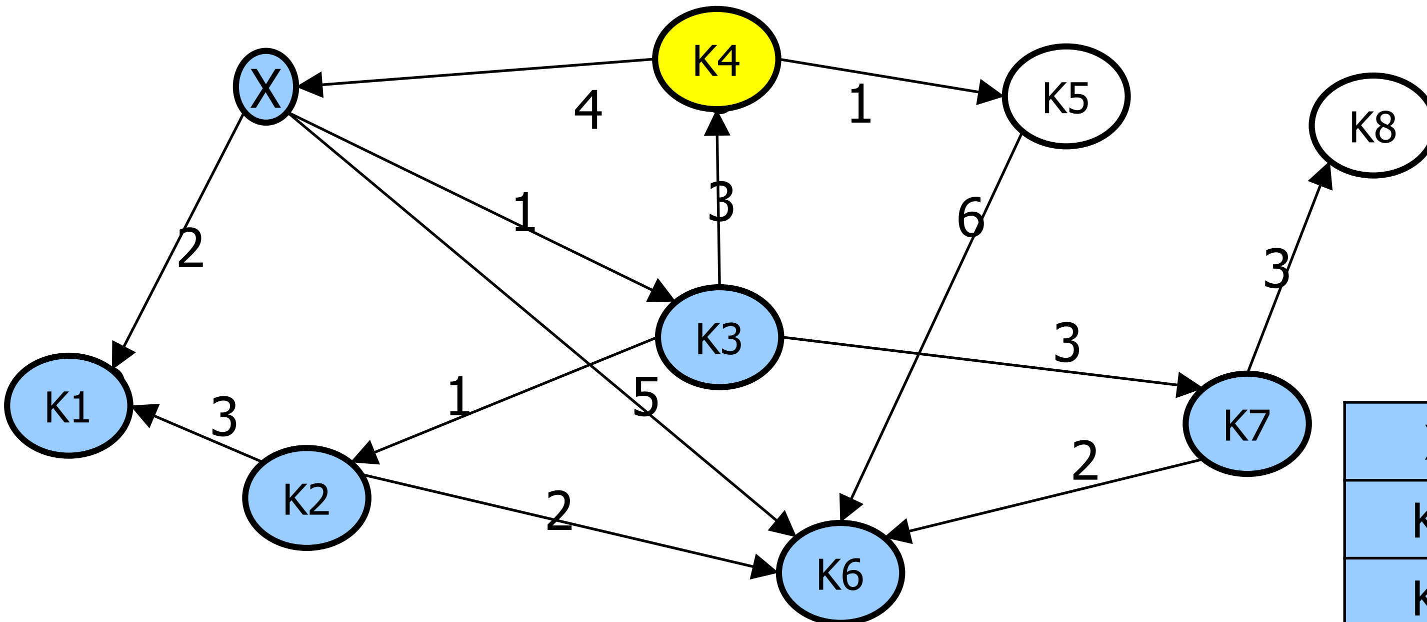
Example for Algorithm



- Pick K4

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	7

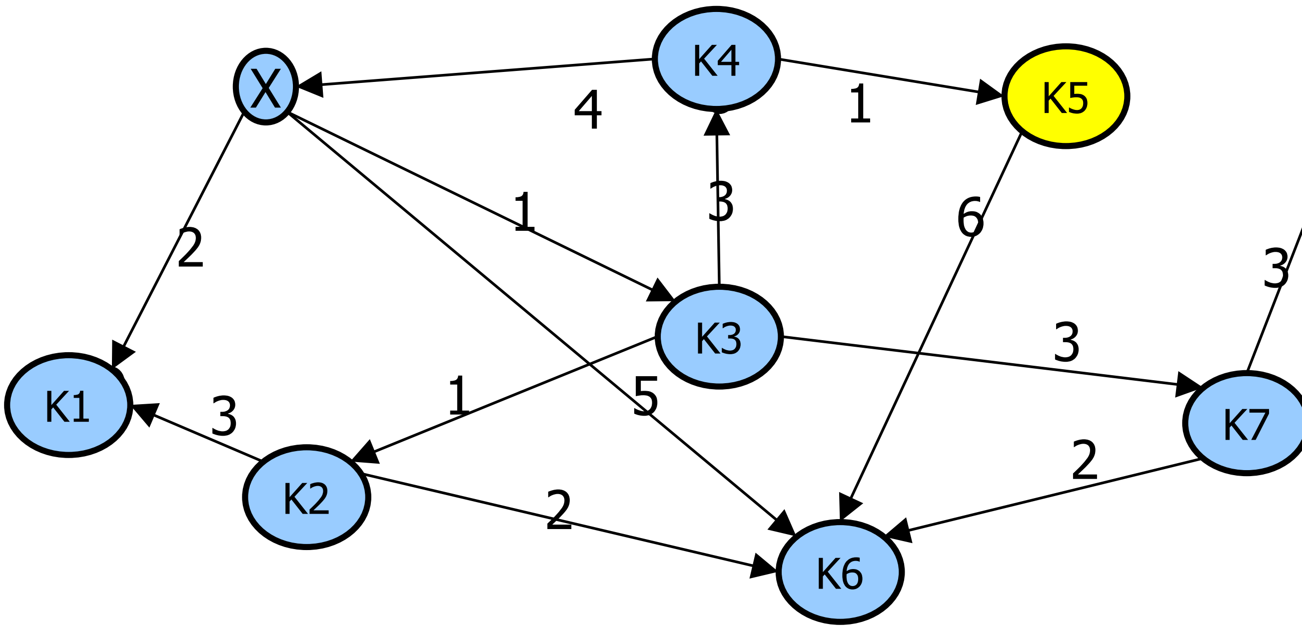
Example for Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	5
K6	4
K7	4
K8	7

- Pick K4
- Adapt distances to all neighbors
  - X was visited already

Example for Algorithm



- Pick K5 … Pick K8

X	0
K1	2
K2	2
K3	1
K4	4
K5	5
K6	4
K7	4
K8	7

A Closer Look

```
G = (V, E);
x : start_node;     # x∈V
A : array_of_distances;
∀i: A[i]:= ∞;
L := V;
A[x] := 0;
while L≠∅
  k := L.get_closest_node(x);
  L := L \ k;
  forall (k,f,w)∈E do
    if f∈L then
      new_dist := A[k]+w;
      if new_dist < A[f] then
        A[f] := new_dist;
      end if;
    end if;
  end for;
end while;
```

- Central: `get_closest_node(x)`
  - Needs to find the node k in L for which A[k] is the smallest
  - A[k] may change all the time
- Searching A? Resorting A?
- Trick: Organize L as min-heap "enhanced" priority queue
  - We need to be able to update the priority of nodes
  - Done in O(log(n)) by removing then re-inserting the node

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. update( L);
8. while L≠∅
9.   k := L.get_closest_node();
10.  L := L \ k;
11.  forall (k,f,w)∈E do
12.    if f∈L then
13.      new_dist := A[k]+w;
14.      if new_dist < A[f] then
15.        A[f] := new_dist;
16.        update( L);
17.      end if;
18.    end if;
19.  end for;
20.end while;
```

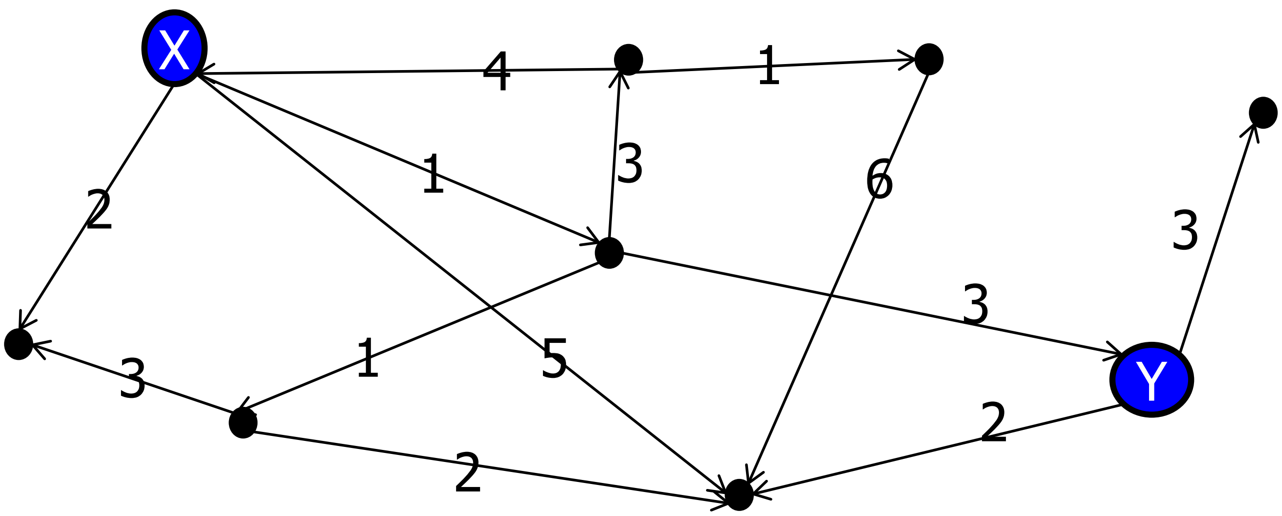
- Assume a heap-based PQ L
  - L holds at most all nodes (n)
  - L4: O(n)
  - L5: O(n) (build PQ)
  - L9: O(1) (getMin)
  - L10: O(log(n)) (deleteMin)
  - L11: O(m) (with adjacency list)
  - L12: O(1)
    - Requires additional array LA of size |V| storing membership of nodes in L
  - L16: O(log(n)) (updatePQ)
    - Store in LA pointers to nodes in L; then remove/insert node

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.  update( L);
8.  while L≠∅
9.    k := L.get_closest_node();
10.   L := L \ k;
11.   forall (k,f,w)∈E do
12.     if f∈L then
13.       new_dist := A[k]+w;
14.       if new_dist < A[f] then
15.         A[f] := new_dist;
16.         update( L);
17.       end if;
18.     end if;
19.   end for;
20. end while;
```

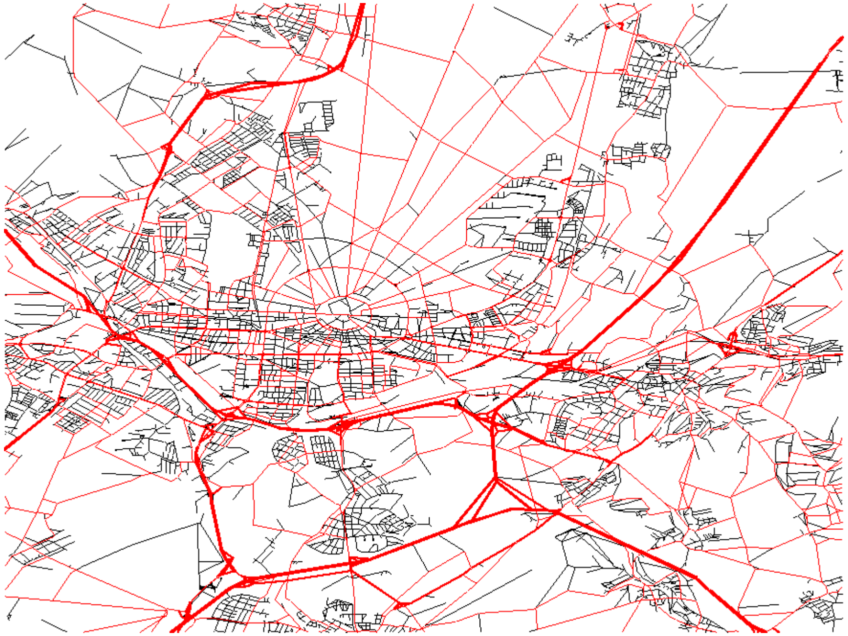
- Central costs
  - L10: O(log(n)) (deleteMin)
  - L16: O(log(n)) (del+ins)
- Loops
  - Lines 8-19: O(n)
  - Line 11-18: All edges exactly once
  - Together: O(m+n)
- Altogether: O((n+m)*log(n))
  - With Fibonacci heaps: Amortized costs are O(n*log(n)+m))
  - Also possible in $O(n^2)$; this is better in dense graphs ($m \sim n^2$)

Single-Source, Single-Target



- Task: Find the distance between X and only Y
- Solution: Dijkstra as well
  - We can stop as soon as Y appears at the min position of the PQ
  - We can visit edges in order of increasing weight (might help)
  - Worst-case complexity unchanged
- Things are different in planar graphs (navigators!)

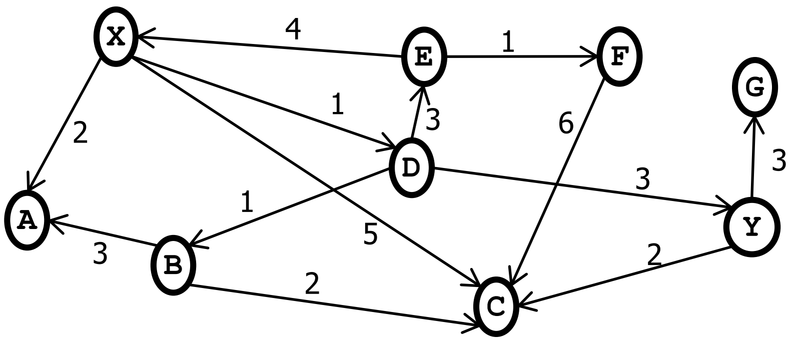
Outlook: Highway Hierarchies



- Shortest-Path Routing on maps
- Exploit Highway hierarchy
  - Autobahn, Bundesstrasse, Regionalstrasse, Strasse, Pfad …
- Iterative refinement in layered maps
- "towards O(1)" [SS07]
- Extensions
  - Second best non-overlapping path
  - Fleet management: Traveling salesman
  - Logistics: Pick-up-and-delivery with intermediate stocks
  - Budget optimization (gasoline, empty trips, slepp-restrictions, road tolls, border / customs regulations, …)

Faster SS-ST Algorithms

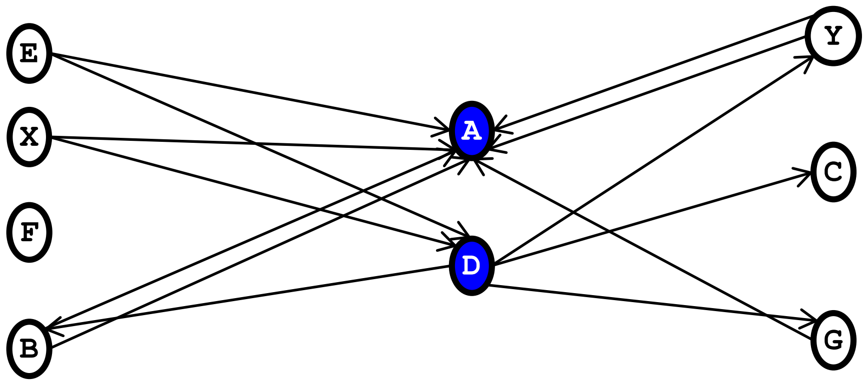
- Trick 1: Pre-compute all distances
  - Transitive closure with distances
  - Requires $O(|V|^2)$ space: Prohibitive for large graphs
  - How? See next lecture



→	A	B	C	D	E	F	G	X	Y
A	0	-	-	-	-	-	-	-	-
B	3	0	2	-	-	-	-	-	-
C	-	-	0	-	-	-	-	-	-
D	4	1	3	0	3	4	6	7	3
E	6	6	7	5	0	1	11	4	8
F	-	-	6	-	-	0	-	-	-
G	-	-	-	-	-	-	0	-	-
X	2	2	4	1	4	5	7	0	4
Y	-	-	2	-	-	-	3	-	0

Faster SS-ST Algorithms

- Trick 2: Two-hop cover with distances
  - Find a (hopefully small) set S of nodes such that
    - For every pair of nodes v_1,v_2, at least one shortest path from v_1 to v_2 goes through a node $s \in S$
    - Thus, the distance between v_1,v_2 is $\min\{ d(v_1,s)+d(s,v_2) \mid s \in S\}$
    - S is called a 2-hop cover
  - Problem: Finding a minimal S is NP-complete
    - And S need not be small



More Distances

- Graphs with negative edge weights
  - Shortest paths (in terms of weights) may be very long (edges)
  - Bellman-Ford algorithm is in $O(n^2*m)$
- All-pairs shortest paths
  - Only positive edge weights: Use Dijkstra n times
  - With negative edge weights: Floyd-Warshall in $O(n^3)$
    - See next lecture
- Reachability
  - Simple in undirected graphs: Compute all connected components
  - In digraphs: Use graph traversal or a special graph indexing method