Weighted Graphs

Computing 2 COMP1927 16x1

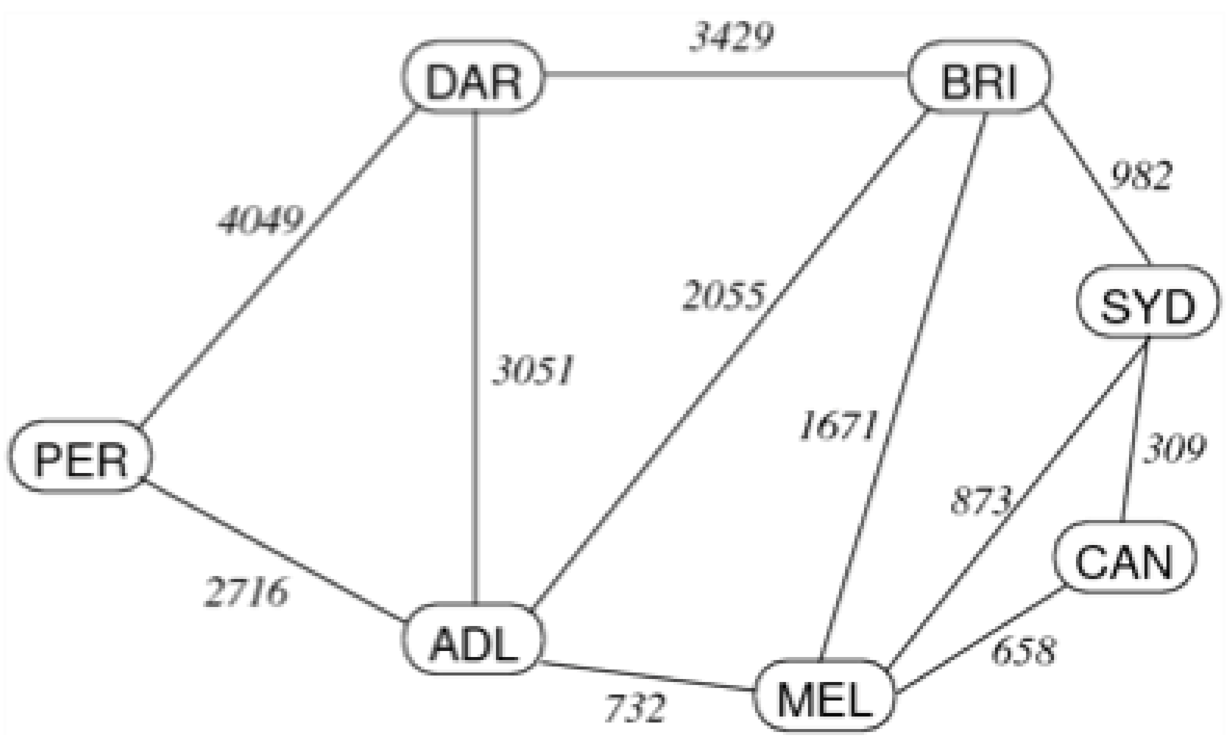
Sedgewick Part 5: Chapter 20.1 -20.4
21.1 - 21.3

WEIGHTED GRAPHS

- Some applications require us to consider a cost or weight
  - costs/weights are assigned to edges
- Often use a geometric interpretation of weights
  - low weight - short edge
  - high weight - long edge
- Weights are not always geometric
  - Some weights can be negative
    - this can make some problems more difficult!
    - Assume in our graphs we have non-negative weights

Example: Weighted Graphs

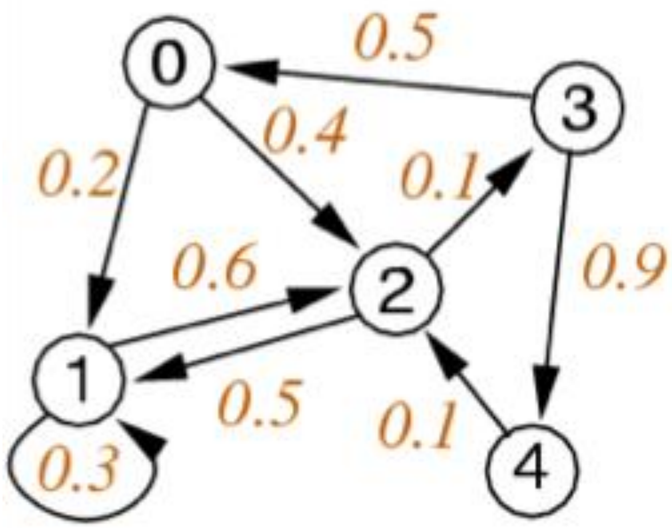
- Example: "map" of airline flight routes
  - vertices = airports
  - edge = flights
  - weights = distance/time/price



Weighted Graph Implementation

- Adjacency Matrix Representation
  - change 0 and 1 to float/double
  - Need a special float constant to indicate NO_EDGE
  - Can’t use 0. It may be a valid weight
- Adjacency Lists Representation
  - add float weight to each node
- This will work for directed or undirected graphs

ADJACENCY MATRIX WITH WEIGHTS

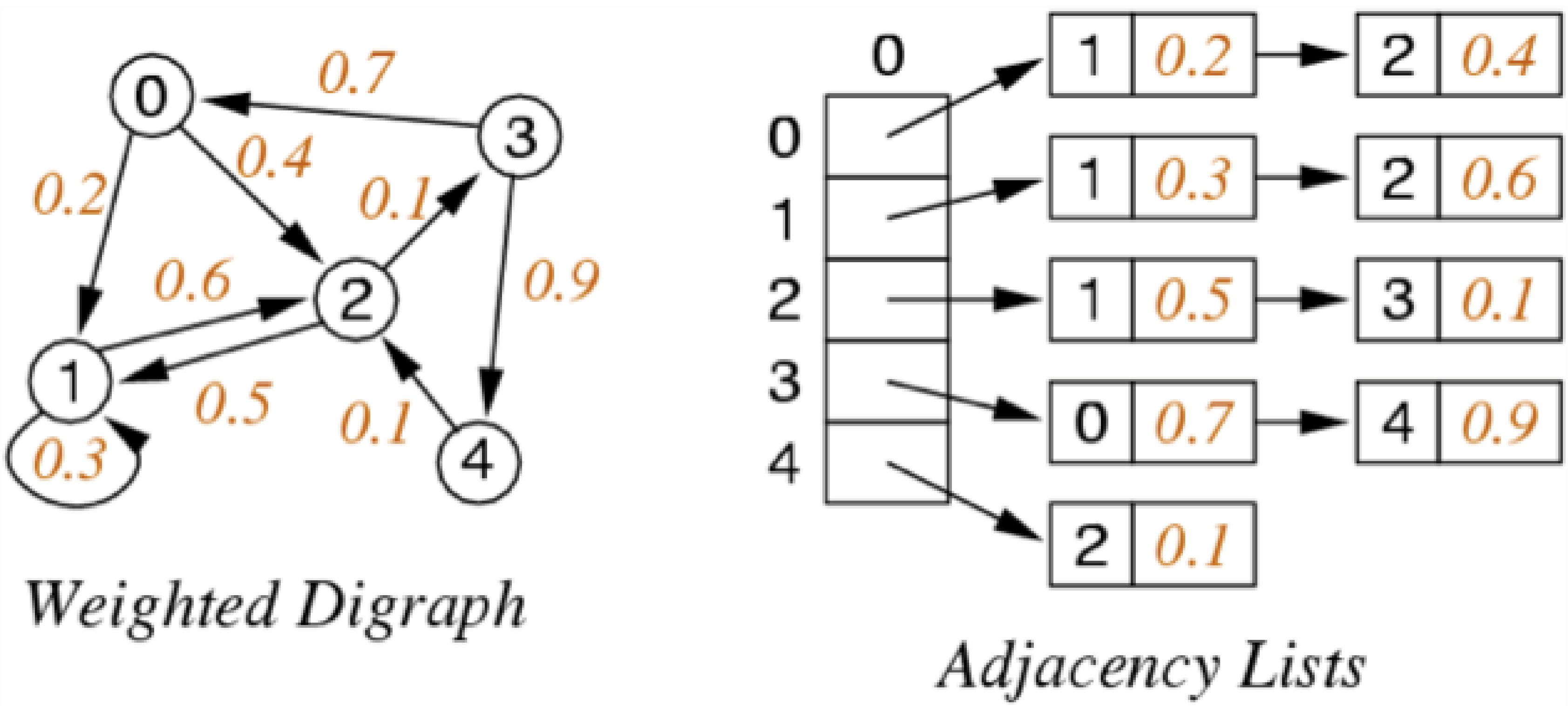


Weighted Digraph

	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

Adjacency Matrix

ADJACENCY LIST REPRESENTATION WITH WEIGHTS



Weighted Graph Problems

- Minimum spanning tree
  - find the minimal weight set of edges that connect all vertices in a weighted graph
    - might be more than one minimal solution
  - we will assume undirected graph
  - we will assume non-negative weights
- Shortest Path Problem
    - Find minimum cost path to from one vertex to another
    - Edges may be directed or undirected
    - We will assume non-negative weights

Minimal Spanning Tree Problem

- Origins
  - Otakar Boruvka, electrical engineer in 1926
  - most economical construction of electric power network

- Some modern applications of MST:
  - network layout: telephone, electric, computer, road, cable

- Has been studied intensely, still looking for faster algorithms

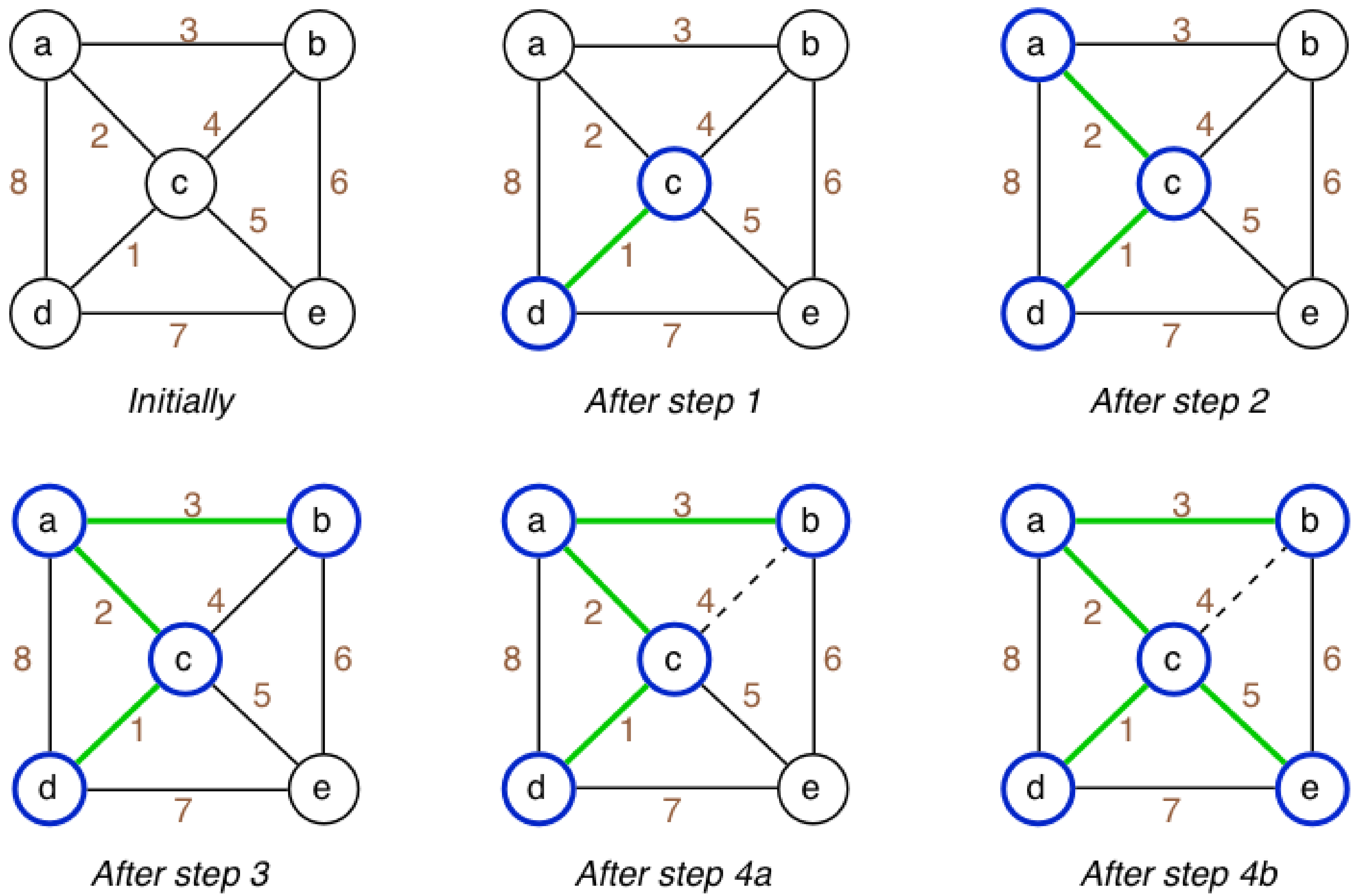
Minimum Spanning Trees (MST)

- Reminder: *Spanning tree ST* of graph *G(V,E)*
  - *ST* is a subgraph of *G*
    - (*G'(V,E')* where *E' is a subset of E*)
  - *ST* is *connected* and *acyclic*
- *Minimum spanning tree MST* of graph *G*
  - *MST* is a spanning tree of *G*
    - sum of edge weights is no larger than any other ST
- Problem: how to (efficiently) find MST for graph *G*?

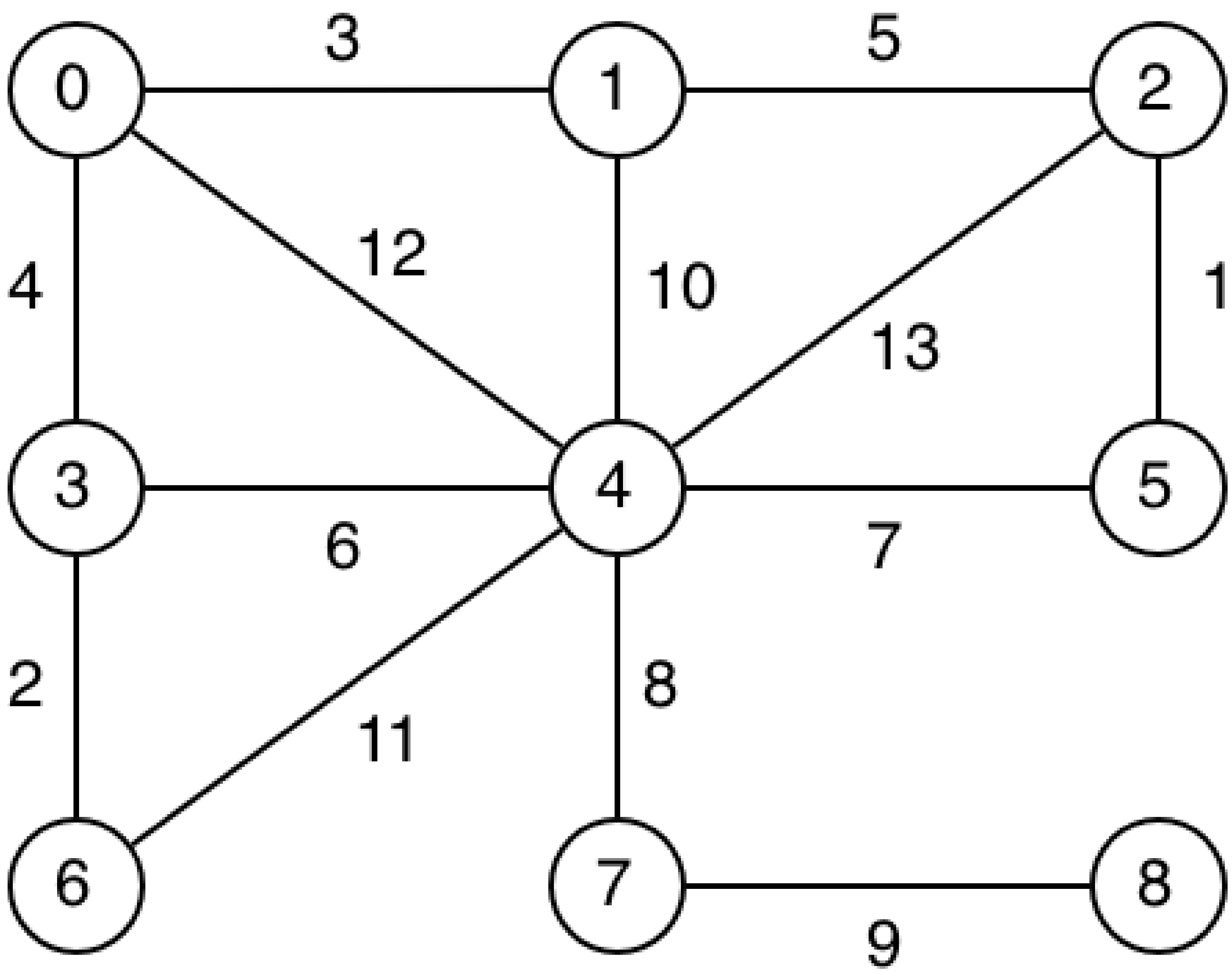
Kruskal's MST Algorithm

- One approach to computing MST for graph *G(V,E)*:
  - start with empty MST
  - consider edges in increasing weight order
  - add edge if it does not form a cycle in MST
  - repeat until *V-1* edges are added
- Critical operations:
  - iterating over edges in weight order
  - checking for cycles in a graph

EXECUTION TRACE OF KRUSKAL'S MST



Exercise: Trace Kruskal's Algorithm



KRUSKAL'S ALGORITHM: MINIMAL SPANNING TREE

- Implementation 1: Two main parts:
  - sorting edges according to their length ($E * log\ E$)
  - check if adding an edge would create a cycle
    - Could check for cycles using DFS ... but too expensive
    - use Union-Find data structure from Sedgewick ch.1
      - If we use this the cost of sorting dominates so over all
        - E log E
- Implementation 2: Using a pq instead of full sort
  - Create a priority queue using weights as priority
  - Allows us to remove edges from pq in weighted order
  - O($E + X *log\ V$), with X = number of edges shorter than the longest edge in the MST

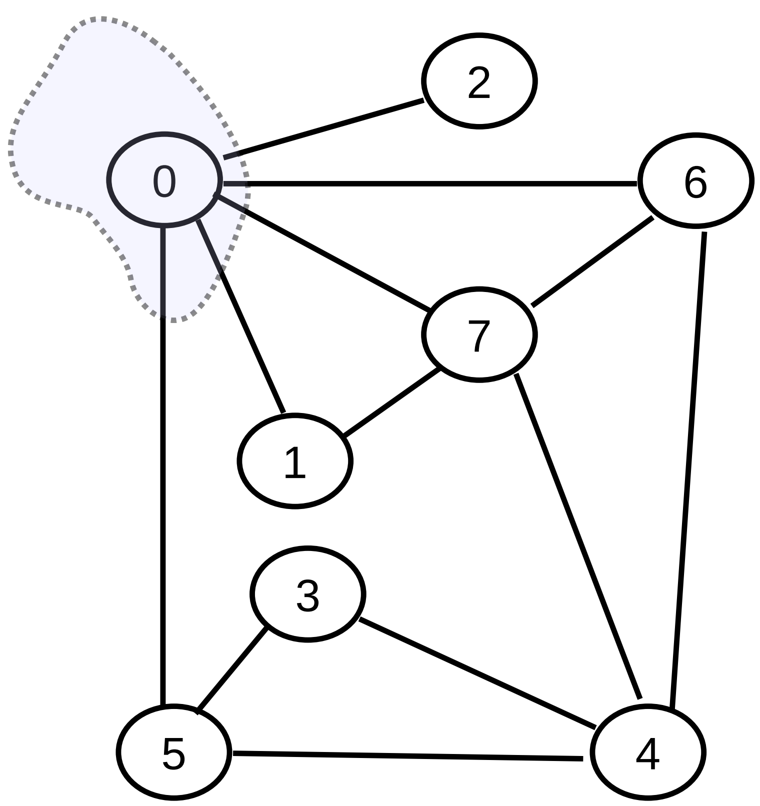
Prim's Algorithm: Minimal Spanning Tree

- Another approach to computing MST for graph *G(V,E)*:
  - start from any vertex *s* and empty MST
    - choose edge not already in MST to add to MST
      - must not contain a self-loop
      - must connect to a vertex already on MST
      - must have minimal weight of all such edges
    - check to see whether adding the new edge brought any of the non-tree vertices closer to the tree
    - repeat until MST covers all vertices

- Critical operations:
  - checking for vertex being connected in a graph
  - finding min weight edge in a set of edges
  - updating min weights in a set of edges

PRIM'S MST ALGORITHM

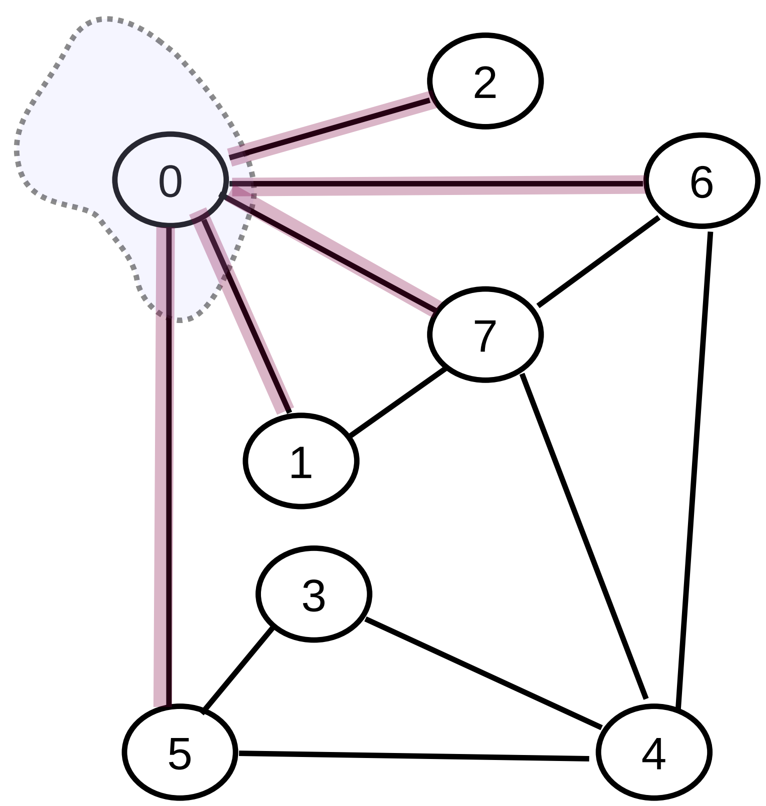
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

PRIM'S MST ALGORITHM

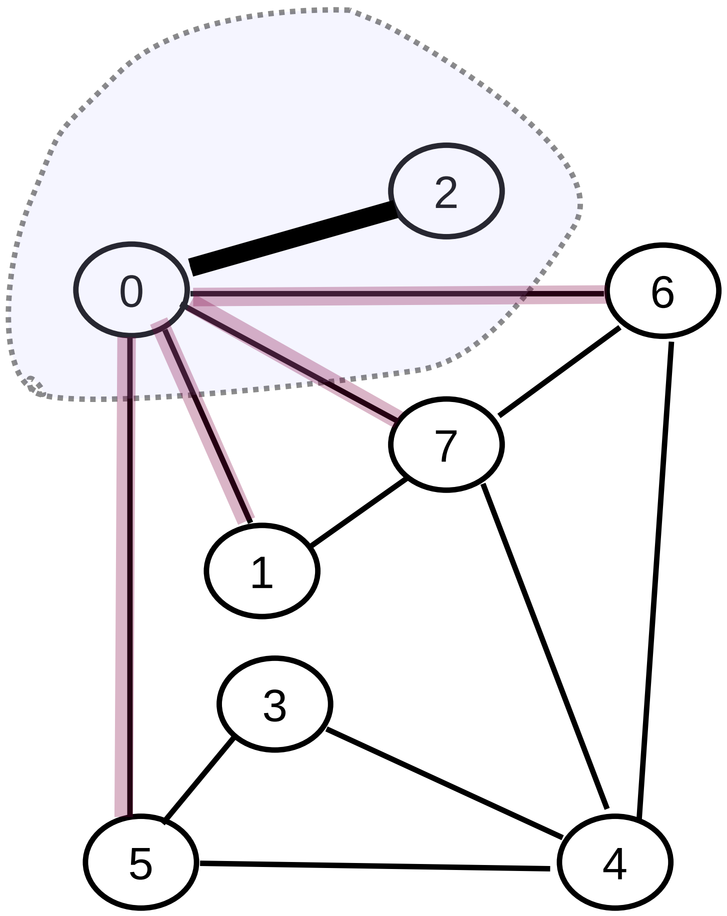
- Idea:
  - Starting from a sub-graph containing only one vertex, we successively add the shortest vertex connecting the sub-graph with the rest of the nodes to the tree
  - Edges in pink are in the fringe



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

PRIM'S MST ALGORITHM

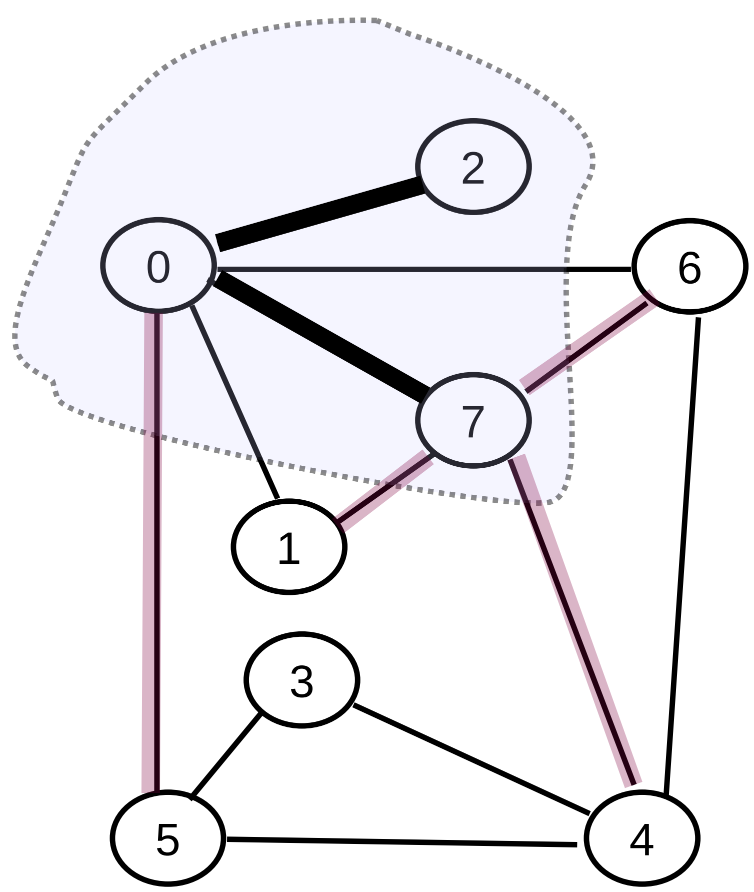
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

WEIGHTED GRAPH: MINIMAL SPANNING TREE II

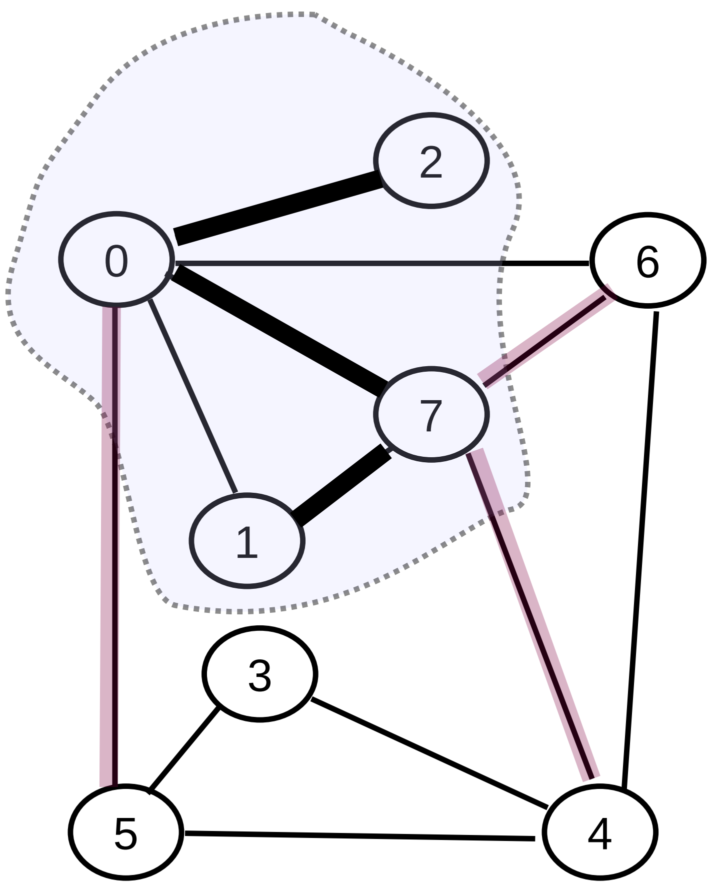
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

WEIGHTED GRAPH: MINIMAL SPANNING TREE II

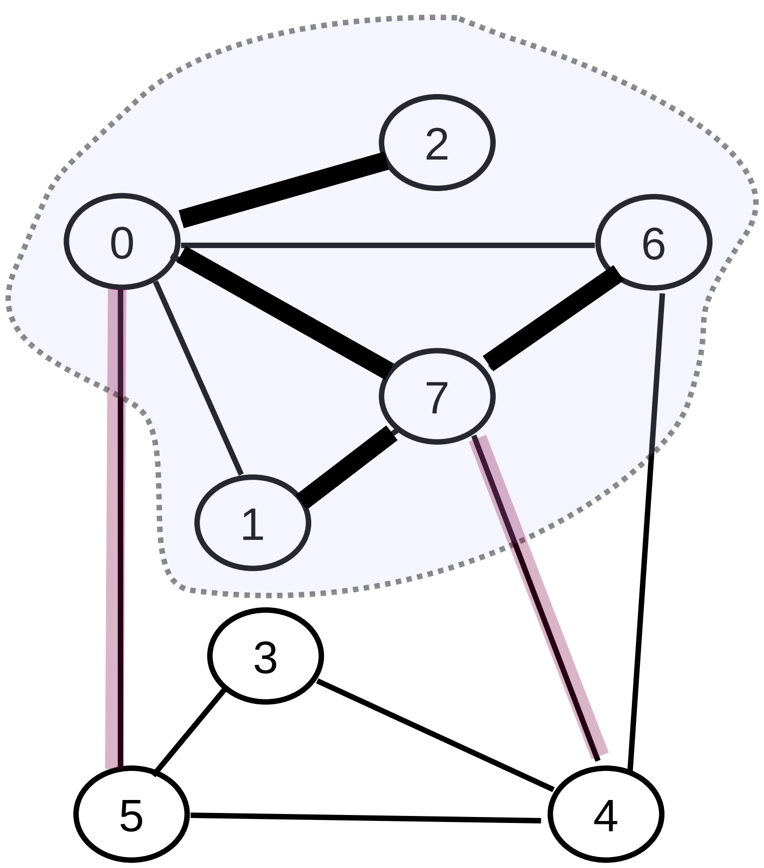
- **Idea**:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

WEIGHTED GRAPH: MINIMAL SPANNING TREE II

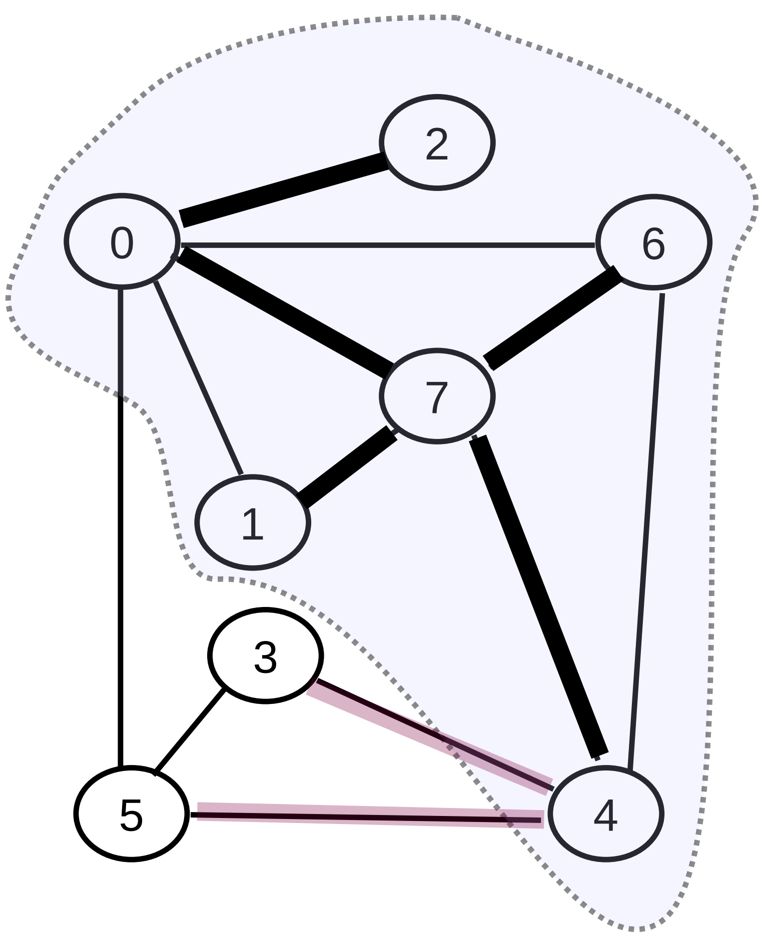
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

PRIM'S MST ALGORITHM

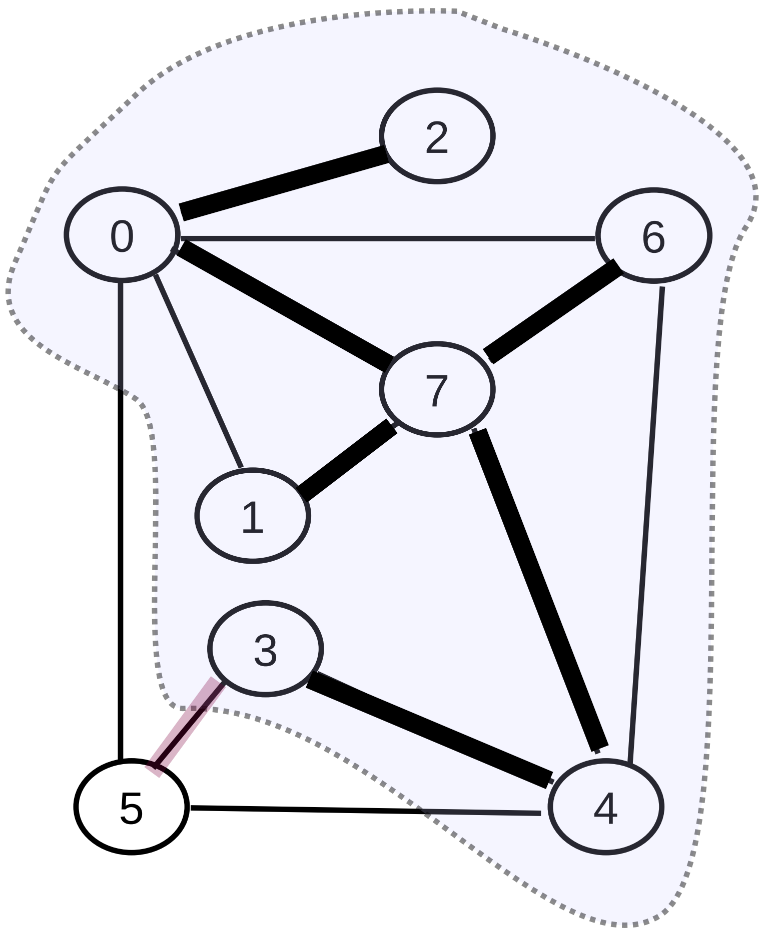
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

PRIM'S MST ALGORITHM

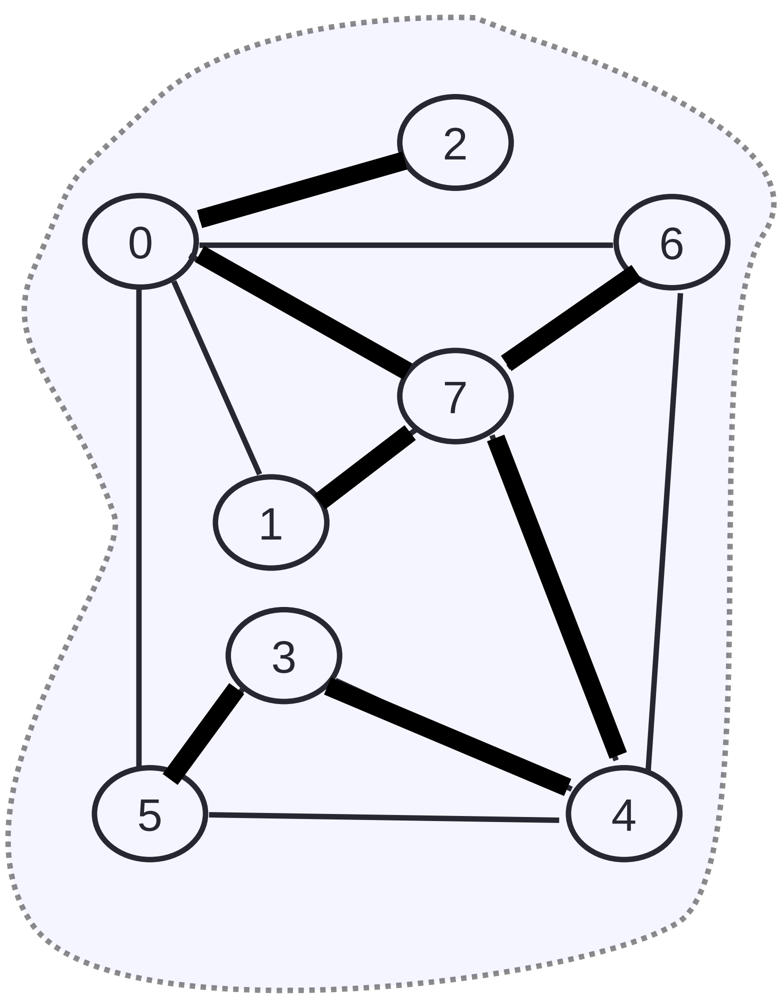
- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



- 0 – 1 (32)
- 0 – 2 (29)
- 0 – 5 (60)
- 0 – 6 (51)
- 0 – 7 (31)
- 1 – 7 (21)
- 3 – 4 (34)
- 3 – 5 (18)
- 4 – 5 (40)
- 4 - 6 (51)
- 4 – 7 (46)
- 6 – 7 (25)

PRIM'S MST ALGORITHM

- Idea:
  - Starting from a subgraph containing only one vertex, we successively add the shortest vertex connecting the subgraph with the rest of the nodes to the tree
  - Edges in pink are in the fringe
  - Edges in black bold are in the MST



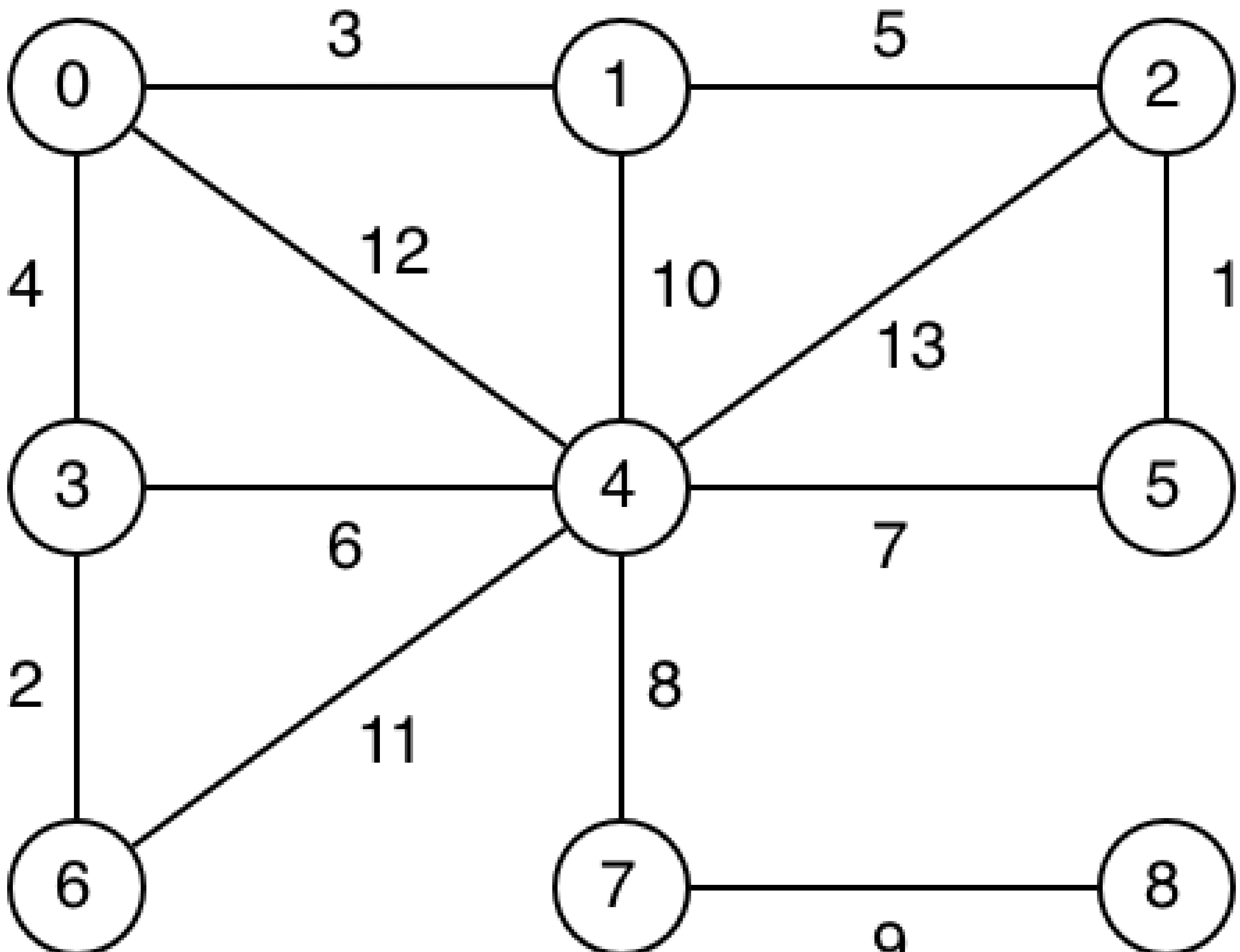
- 0 – 1 (32)
- **0 – 2 (29)**
- 0 – 5 (60)
- 0 – 6 (51)
- **0 – 7 (31)**
- **1 – 7 (21)**
- **3 – 4 (34)**
- **3 – 5 (18)**
- 4 – 5 (40)
- 4 - 6 (51)
- **4 – 7 (46)**
- **6 – 7 (25)**

PRIM'S ALGORITHM

- Prim's algorithm is just a graph search –
  - instead of depth first (using a stack) or breadth first (using a queue),
    - we choose a shortest first` strategy using a priority queue

- It can be implemented to run in
  - O(E * log V) steps
    - if the steps listed above are implemented efficiently (using adjacency lists and heap),
  - O(V^2) for adjacency matrix
- See lecture code for an implementation

EXERCISE: TRACE PRIM'S ALGORITHM

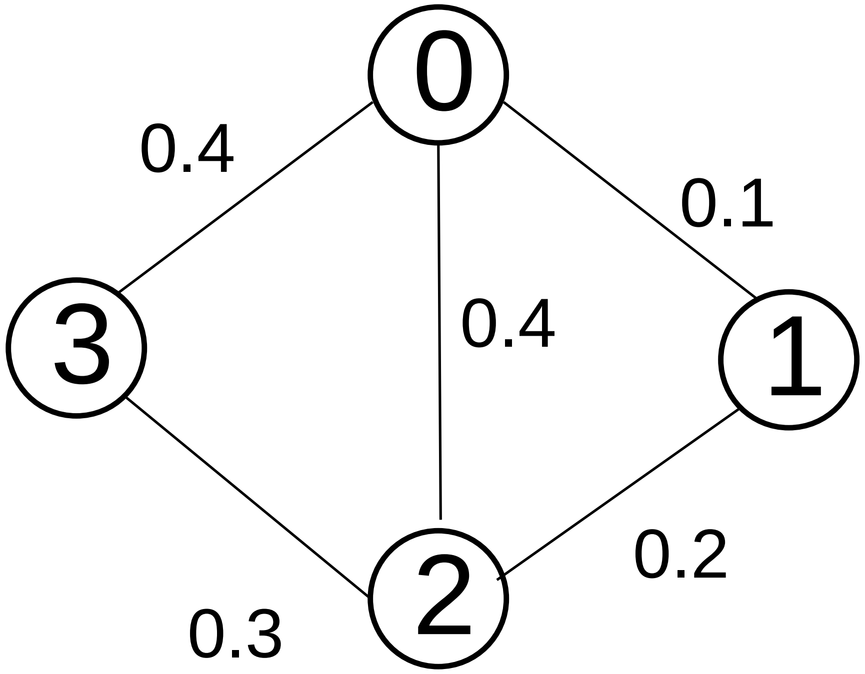


SHORTEST PATHS

- Weight of a path p in graph G
  - sum of weights on edges along path (weight(p))
- Shortest path between vertices s and t
  - a simple path p where s = first(p), t = last(p)
  - no other simple path q has weight(q) < weight(p)
- Problem: how to (efficiently) find shortestPath(G,s,t)?
  - Assumptions: weighted graph, no negative weights.

EXERCISE:

- What is the minimum spanning tree?
- What is the shortest path from 0 to 3?
- What is the least hops path (shortest unweighted path) from 0 to 2?

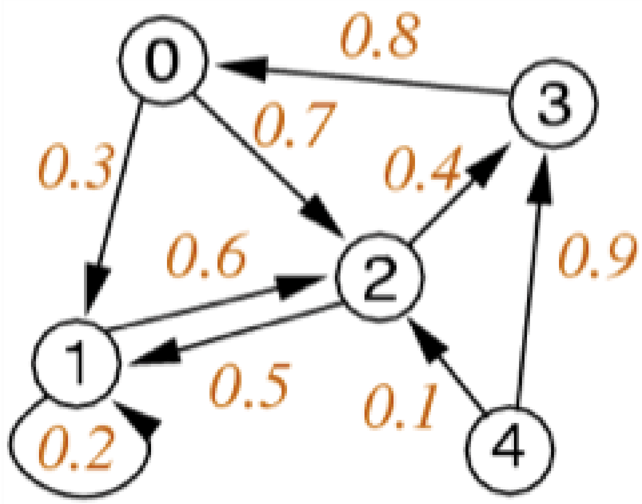


SHORTEST PATH ALGORITHMS

- Shortest-path is useful in a wide range of applications
  - robot navigation
  - finding routes in maps
  - routing in data/computer networks

- Flavours of shortest-path
  - source-target (shortest path from s to t)
  - single-source (shortest paths from s to all other V)
  - all-pairs (shortest paths for all (s,t) pairs)

DIJKSTRA'S ALGORITHM
SINGLE SOURCE SHORTEST PATHS



Weighted Digraph

V	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
st	–	0	0	2	–

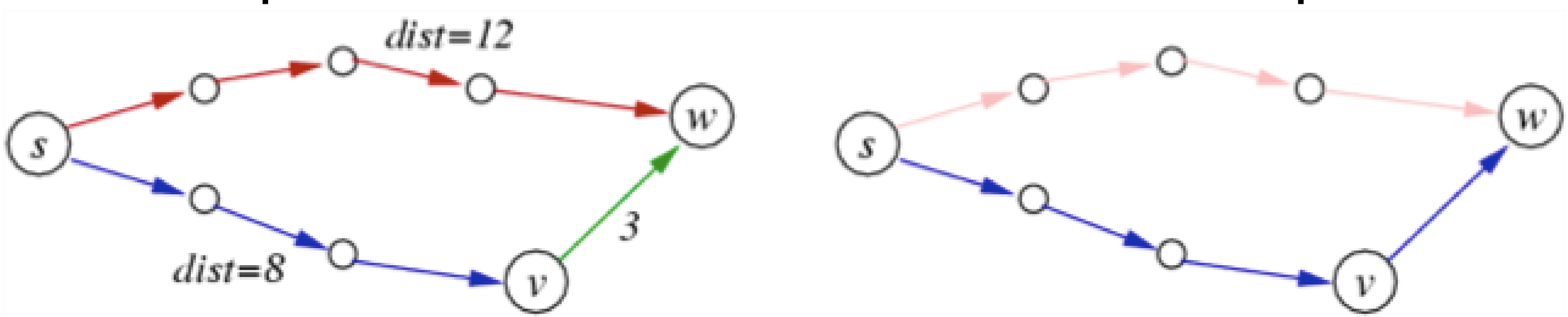
Shortest paths from s=0

DIJKSTRA'S ALGORITHM: SINGLE SOURCE SHORTEST PATH

- Given:
  - weighted digraph/graph *G*, source vertex *s*
- Result:
  - shortest paths from *s* to **all** other vertices
  - `dist[]` : *V*-indexed array of distances from *s*
  - `st[]` : *V*-indexed array of predecessors in shortest path
- Note: shortest paths can be viewed as tree rooted at *s*

EDGE RELAXATION

- Relaxation along edge e from v to w
  - dist[v] is length of some path from s to v
  - dist[w] is length of some path from s to w
  - if e gives shorter path s to w via v,
    then update dist[w] and st[w]
- Relaxation updates data on w if we find a shorter path to s.



dist[v]=8, dist[w]=12
st[v] = ?, st[w] = ?

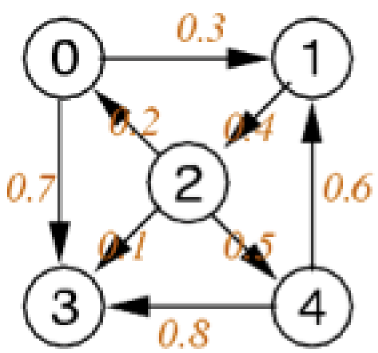
dist[v]=8, dist[w]=11
st[v] = ?, st[w] = v

```
if (dist[v] + e.weight < dist[w]) {
    dist[w] = dist[v] + e.weight;
    st[w] = v;
}
```

DIJKSTRA'S ALGORITHM

- Data:
  - G, s, dist[], st[], and a pq containing the set of vertices whose shortest path from s is not yet known
- Algorithm:
  - initialise dist[] to all ∞, except dist[s]=0
  - Initialise pq with all V, with dist[v] as priority
  - v = deleteMin from pq
    - Get e's that connect v to w in pq
    - relax along e if new dist is better
    - repeat until pq is empty

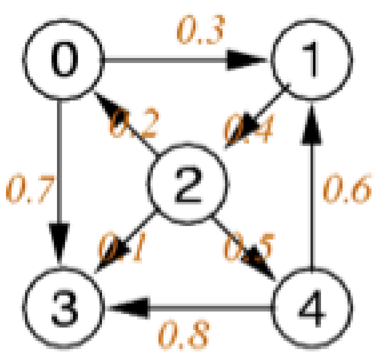
Execution Trace of Dijkstra's Algorithm



Initially

	[0]	[1]	[2]	[3]	[4]
dist	0	inf	inf	inf	inf
st	–	–	–	–	–

pq `{0,1,2,3,4}`

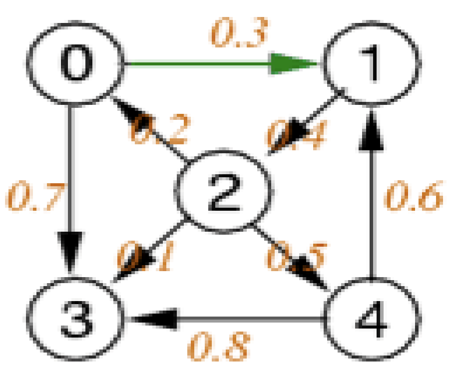


After 0 visited

	[0]	[1]	[2]	[3]	[4]
dist	0	0.3	inf	0.7	inf
st	–	0	–	0	–

pq `{1,2,3,4}`

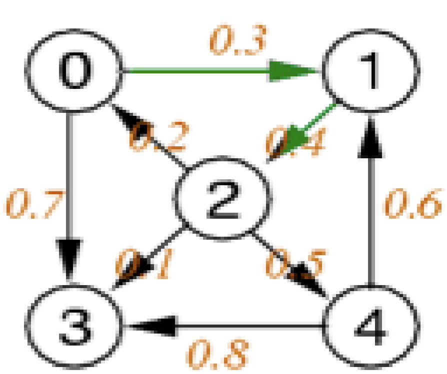
...Execution Trace of Dijkstra's Algorithn



After 1 visited

	[0]	[1]	[2]	[3]	[4]
dist	0	0.3	0.7	0.7	inf
st	–	0	1	0	–

pq {2,3,4}

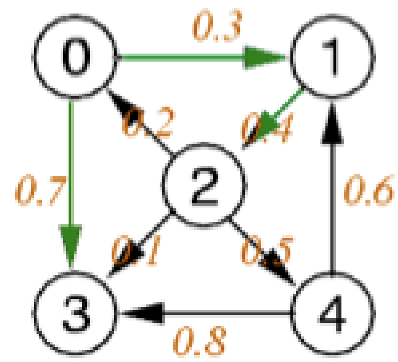


After 2 visited

	[0]	[1]	[2]	[3]	[4]
dist	0	0.3	0.7	0.7	1.2
st	–	0	1	0	2

pq {3,4}

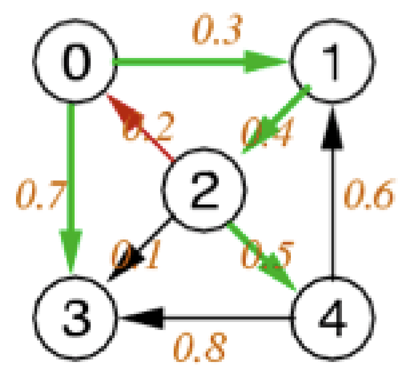
...EXECUTION TRACE OF DIJKSTRA'S ALGORITHM



After 3 visited

	[0]	[1]	[2]	[3]	[4]
dist	0	0.3	0.7	0.7	1.2
st	–	0	1	0	2

pq {4}



After 4 visited

	[0]	[1]	[2]	[3]	[4]
dist	0	0.3	0.7	0.7	1.2
st	–	0	1	0	2

pq {}

DIJKSTRA'S RESULTS

- After the algorithm has completed:
  - Shortest Path distances are in dist array
  - Actual path can be traced back from endpoint via the predecessors in the st array

EXERCISE

- Assume we have just completed running Dijkstra’s algorithm with starting vertex v. Write code to print out the path from vertex v to w or “No path” if the path does not exist. (It is ok to print it in reverse order.)

Trace execution of Dijkstra's Algorithm From starting vertex 2