CS171 Introduction to Computer Science II

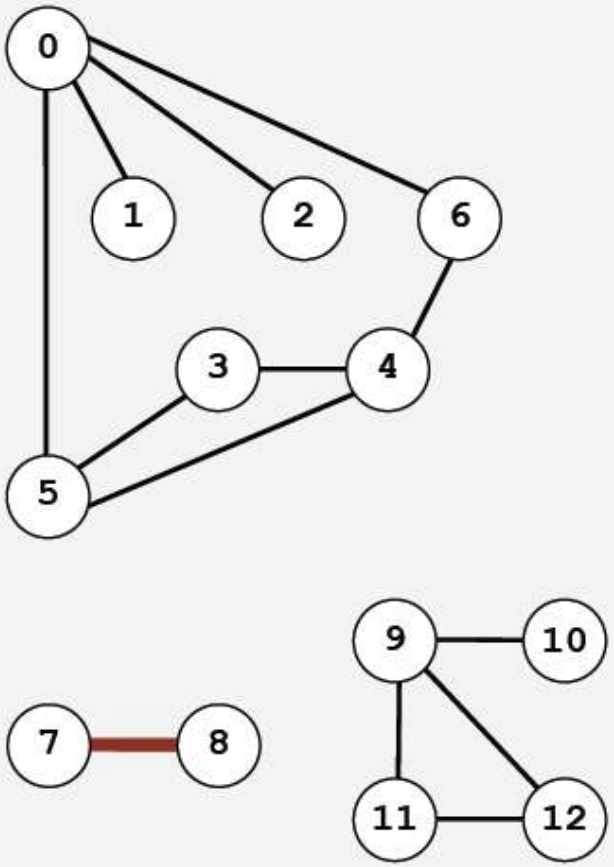
Graphs

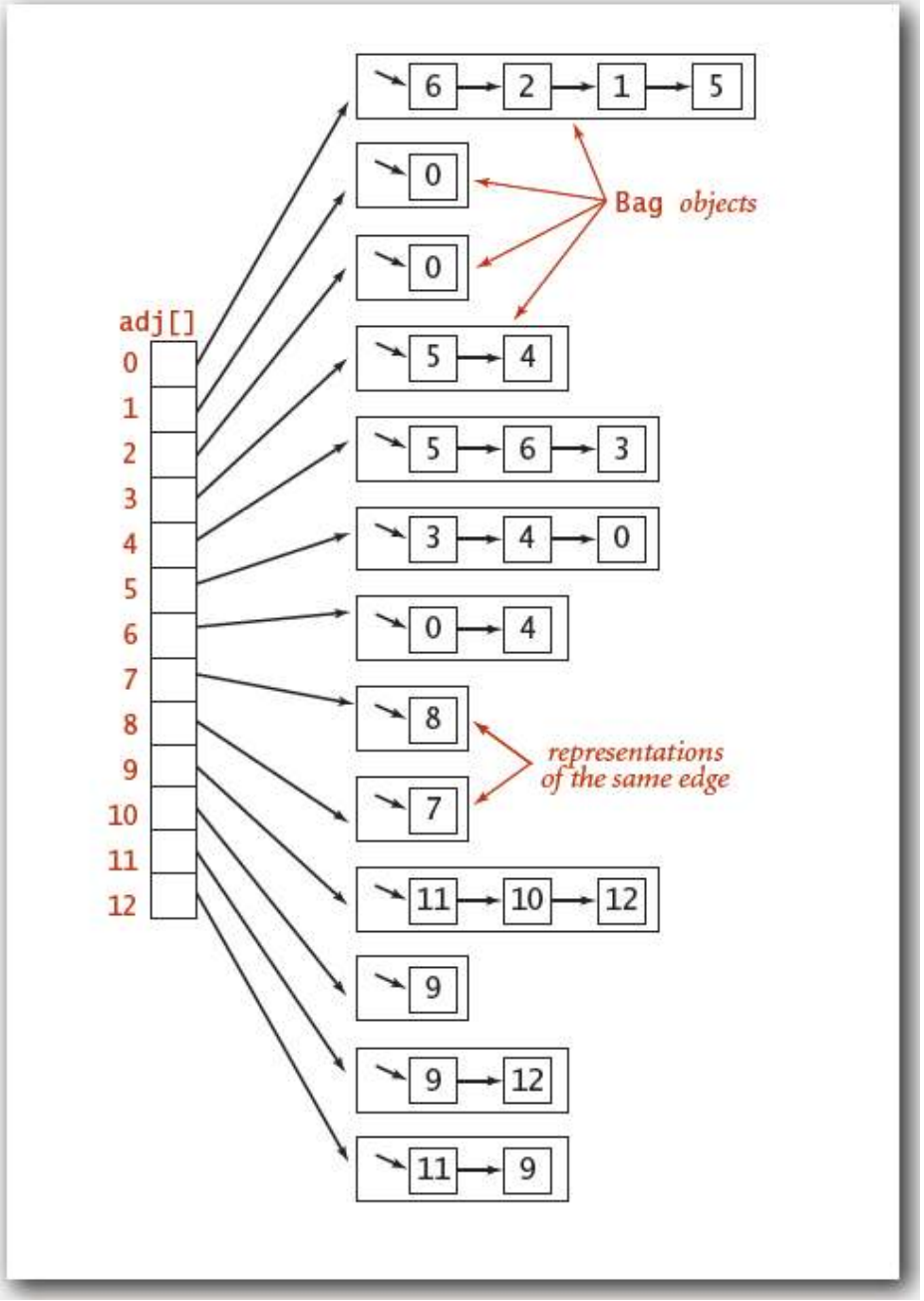
Graphs

- Simple graphs
- Algorithms
  - Depth-first search
  - Breadth-first search
  - shortest path
  - Connected components
- Directed graphs
- Weighted graphs
- Shortest path
- Minimum spanning tree

Adjacency-list graph representation

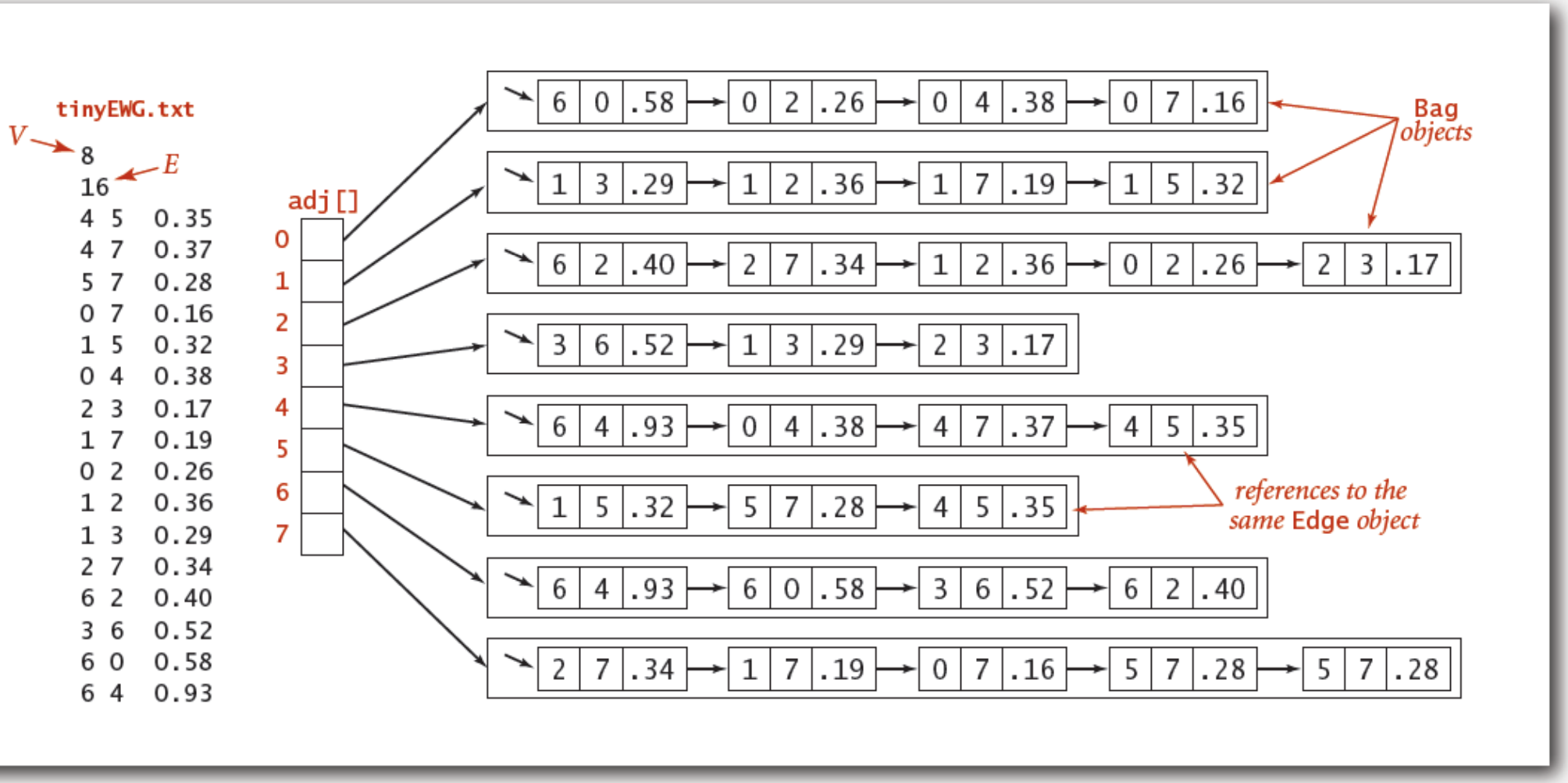
Maintain vertex-indexed array of lists.





Edge-weighted graph: adjacency-lists representation

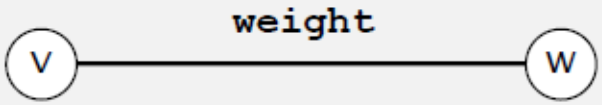
Maintain vertex-indexed array of `Edge` lists (use `Bag` abstraction).



Weighted edge API

Edge abstraction needed for weighted edges.

public class Edge implements Comparable<Edge>	
Edge(int v, int w, double weight)	*create a weighted edge v-w*
int either()	*either endpoint*
int other(int v)	*the endpoint that's not v*
int compareTo(Edge that)	*compare this edge to that edge*
double weight()	*the weight*
String toString()	*string representation*



Idiom for processing an edge `e`: `int v = e.either(), w = e.other(v);`

Edge-weighted graph API

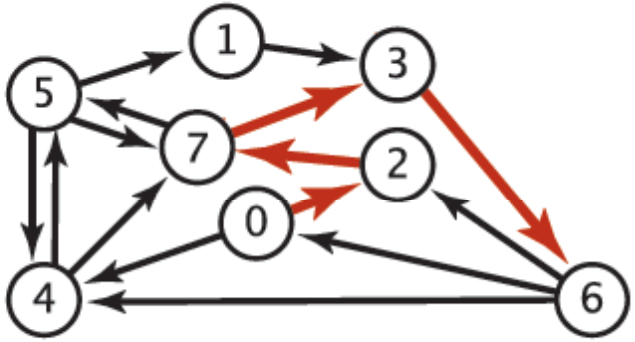
public class EdgeWeightedGraph		
	EdgeWeightedGraph(int V)	*create an empty graph with V vertices*
	EdgeWeightedGraph(In in)	*create a graph from input stream*
void	**addEdge(Edge e)**	*add weighted edge e to this graph*
Iterable<Edge>	**adj(int v)**	*edges incident to v*
Iterable<Edge>	edges()	*all edges in this graph*
int	V()	*number of vertices*
int	E()	*number of edges*
String	toString()	*string representation*

Shortest paths in a weighted digraph

Given an edge-weighted digraph, find the shortest (directed) path from s to t.

edge-weighted digraph

```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



shortest path from 0 to 6

```
0->2  0.26
2->7  0.34
7->3  0.39
3->6  0.52
```

Dijkstra's algorithm

- Maintain a queue of nodes to be examined (open set)
- Remove the node with shortest distance to the source to the closed set and add its neighbors to the open set

Shortest Paths – Dijkstra's Algorithm

- Initialization
  - Assign to every node a distance value: set it to zero for source node and to infinity for all other nodes.
  - Mark all nodes unvisited, insert source node into a queue (open set)

- Repeat until the queue is empty
  - Remove a node from the queue with **the smallest distance** from the **source node** as the "current node" and mark it as visited (closed set)
  - For current node, consider all its unvisited neighbors (not in the closed set) and calculate their ***tentative*** **distance**.
  - If this distance is **less than the previously** recorded distance, **overwrite** the distance and update the parent for the neighbor, and add the neighbor into the queue or update its distance if it is already in the queue (edge relaxation)

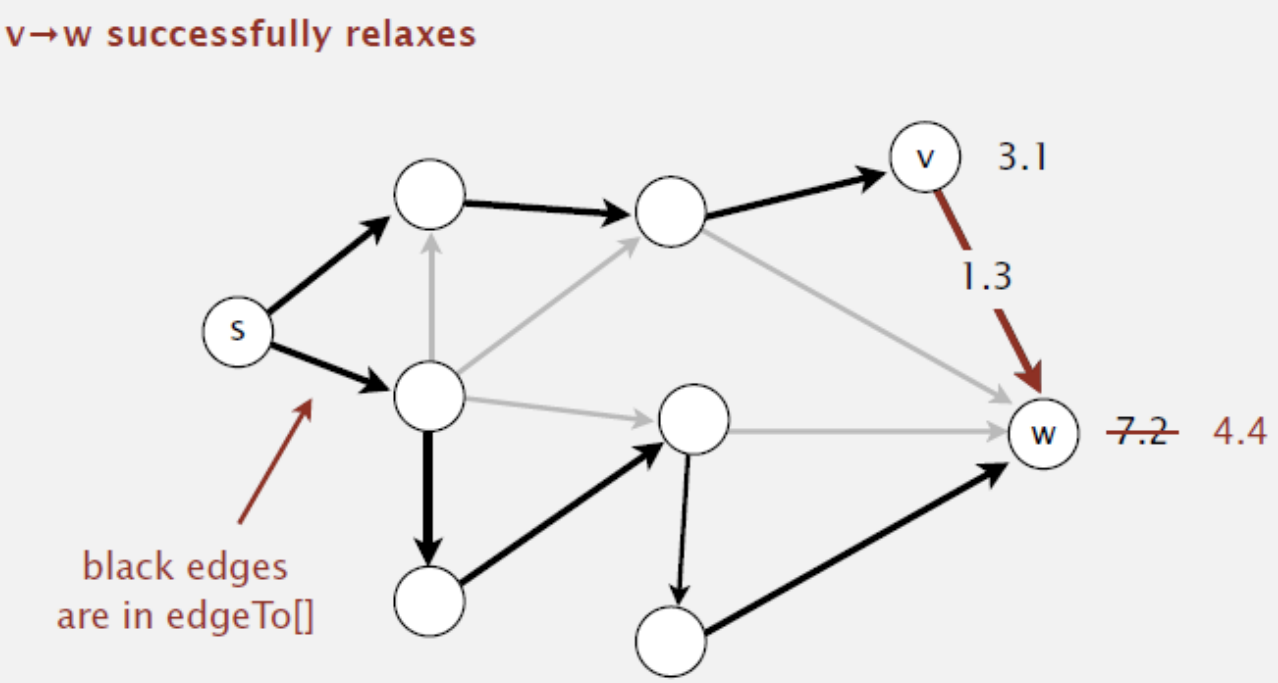
Data structures

- Distance to the source: a vertex-indexed array distTo[] such that distTo[v] is the length of the shortest known path from s to v
- Edges on the shortest paths tree: a parent-edge representation of a vertex-indexed array edgeTo[] where edgeTo[v] is the parent edge on the shortest path to v

Edge relaxation

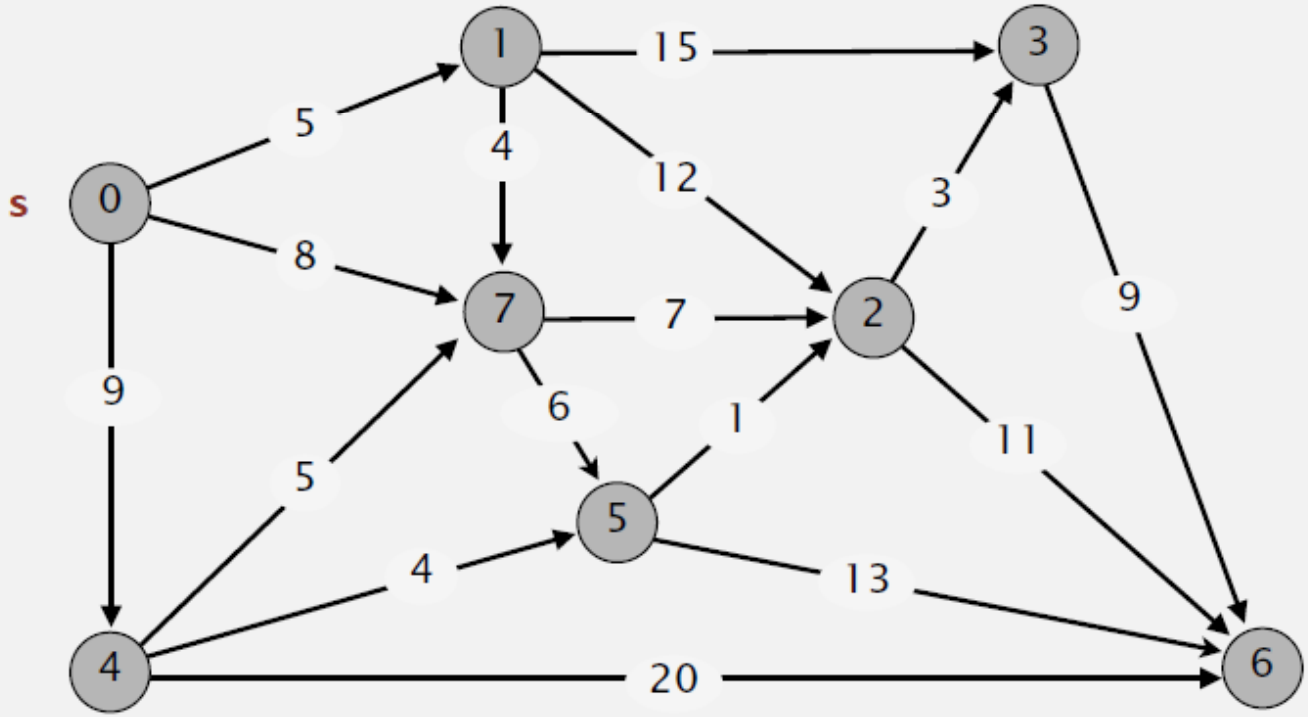
Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.



Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



an edge-weighted digraph

```
0→1   5.0
0→4   9.0
0→7   8.0
1→2  12.0
1→3  15.0
1→7   4.0
2→3   3.0
2→6  11.0
3→6   9.0
4→5   4.0
4→6  20.0
4→7   5.0
5→2   1.0
5→6  13.0
7→5   6.0
7→2   7.0
```

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
          int v = pq.delMin();
          for (DirectedEdge e : G.adj(v))
             relax(e);
      }
   }
}
```

relax vertices in order of distance from s

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
      if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
      else                pq.insert     (w, distTo[w]);
   }
}
```

update PQ

Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

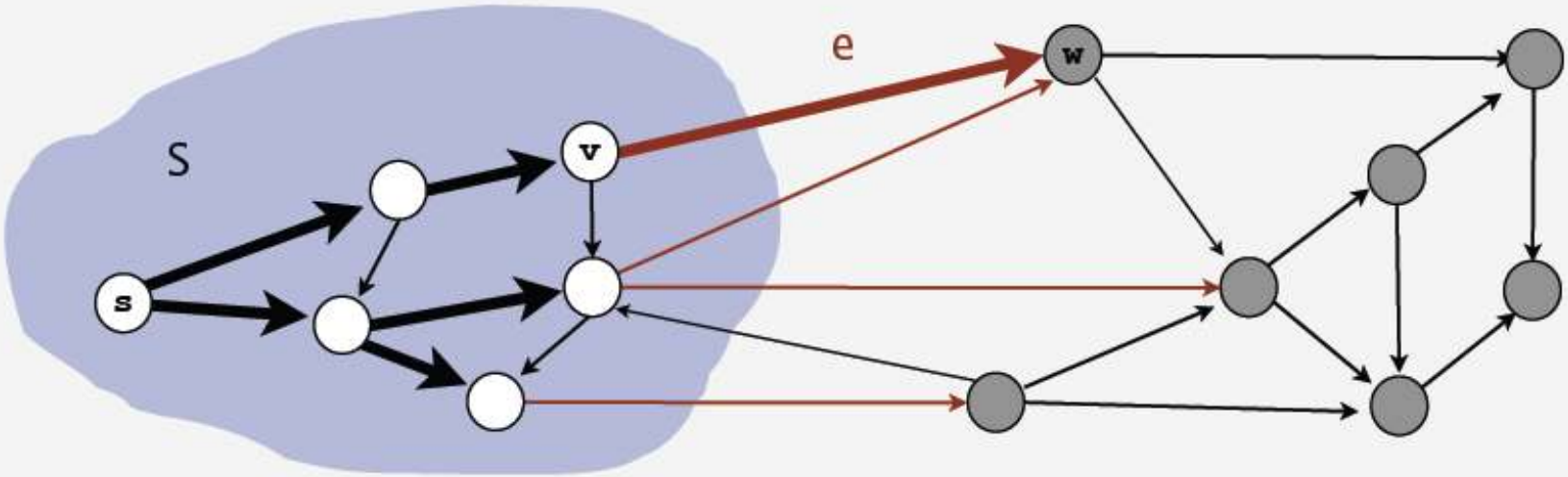
- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

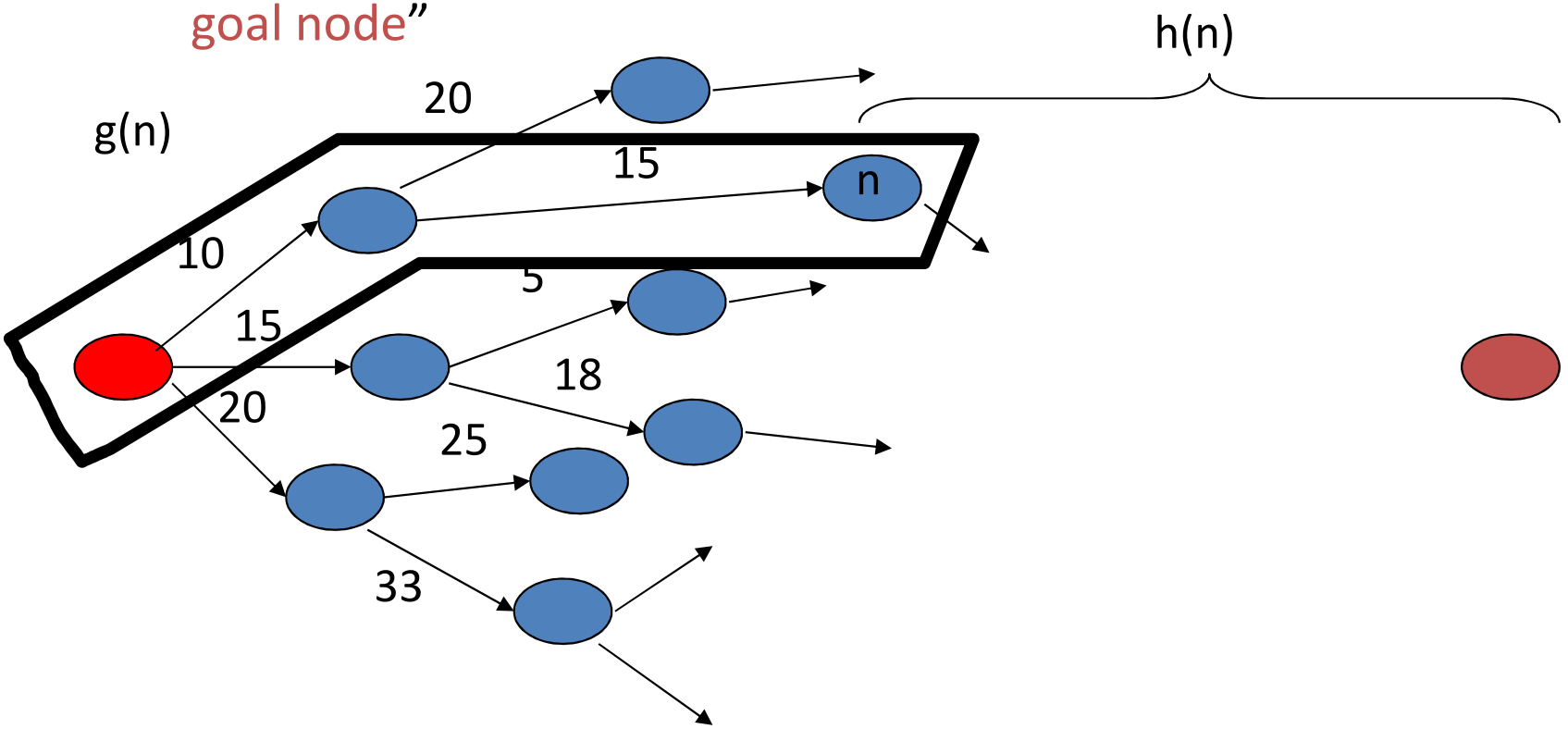
Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to S.



From Dijkstra to A*

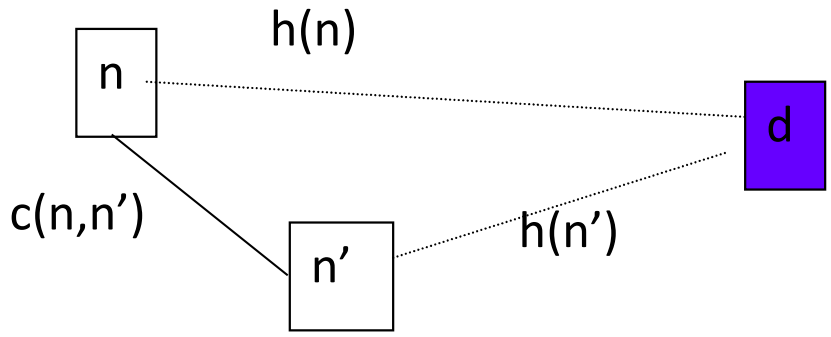
- Dijkstra: remove the node with shortest distance from the source
- A*: remove the node with shortest distance from the source and likely the shortest distance to the target
- f(n) = g(n) + h(n)
  - g(n) = "cost from the starting node to reach n"
  - h(n) = "estimate of the cost of the cheapest path from n to the goal node"



Properties of A*

- A* generates an optimal solution if h(n) is an admissible heuristic and the search space is a tree:
  - h(n) is **admissible** if it never overestimates the cost to reach the destination node
- A* generates an optimal solution if h(n) is a consistent heuristic and the search space is a graph:
  - h(n) is **consistent** if for every node n and for every successor node n' of n:

    $h(n) \leq c(n,n') + h(n')$



- If h(n) is consistent then h(n) is admissible
- Frequently when h(n) is admissible, it is also consistent

Admissible Heuristics

- A heuristic is admissible if it is optimistic, estimating the cost to be smaller than it actually is.

- MapQuest:

  h(n) = “Euclidean distance to destination”

  is admissible as normally cities are not connected by roads that make straight lines

Shortest Paths – A* Algorithm

- Initialization
  - Assign to every node a distance value: set it to zero for source node and to infinity for all other nodes.
  - Mark all nodes unvisited, compute the cost (distance + estimate cost) for source node, and insert it into a queue (open set)

- Repeat until the queue is empty
  - Remove a node from the queue with **the smallest cost** as the "current node“ and mark it as visited (closed set)
  - For current node, consider all its unvisited neighbors (not in the closed set) and calculate their *tentative* distance.
  - If this distance is less than the previously recorded distance, overwrite the distance and update the parent for the neighbor, and compute the cost (distance + estimated cost) for the neighbor, and add the neighbor into the queue or update its cost if it is already in the queue (edge relaxation)

Graphs

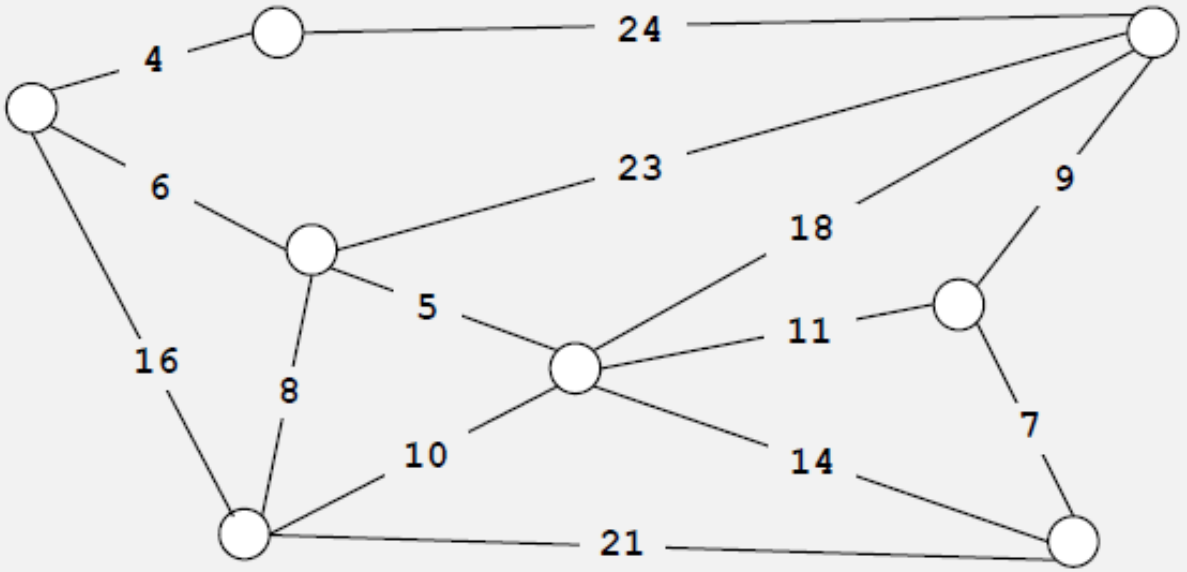
- Simple graphs
- Algorithms
  - Depth-first search
  - Breadth-first search
  - shortest path
  - Connected components
- Directed graphs
- Weighted graphs
- Shortest path
- Minimum spanning tree

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



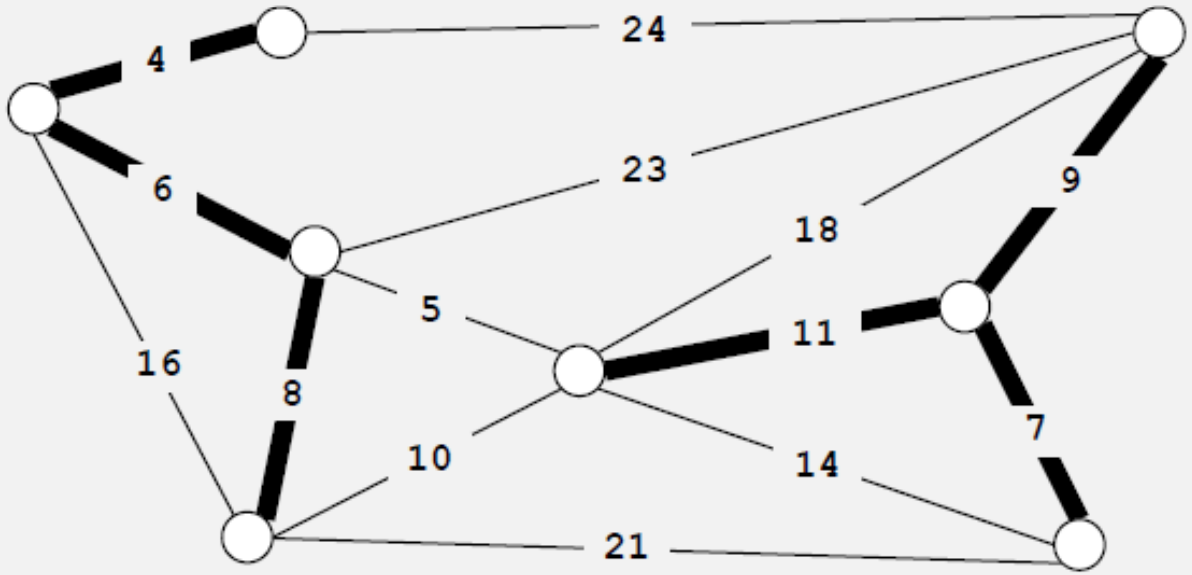
graph G

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

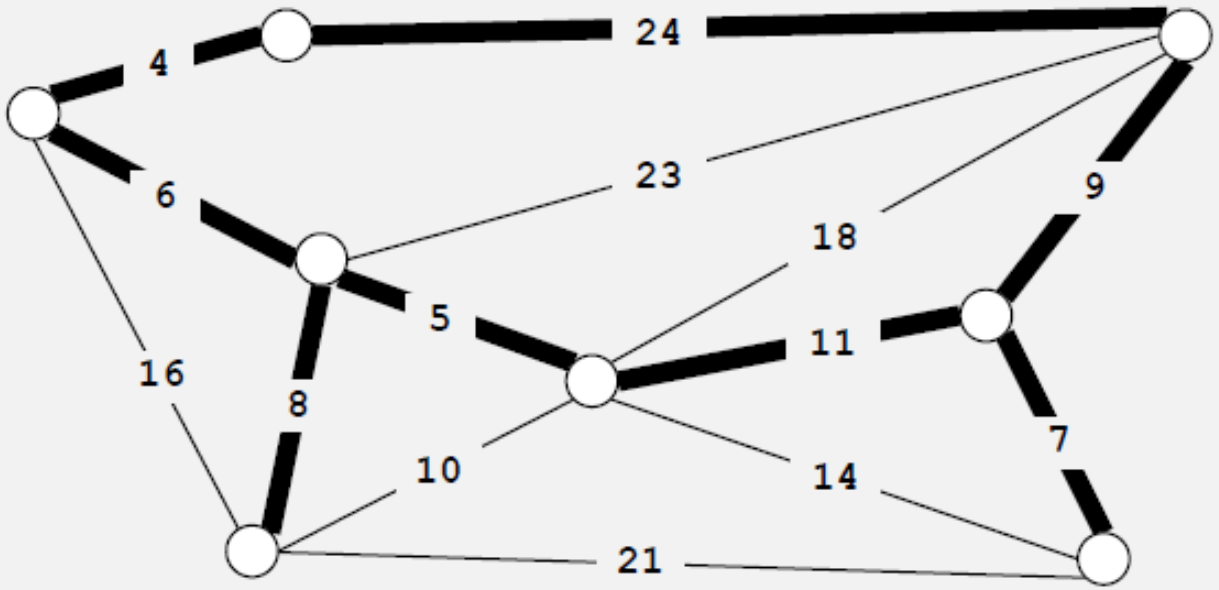


Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

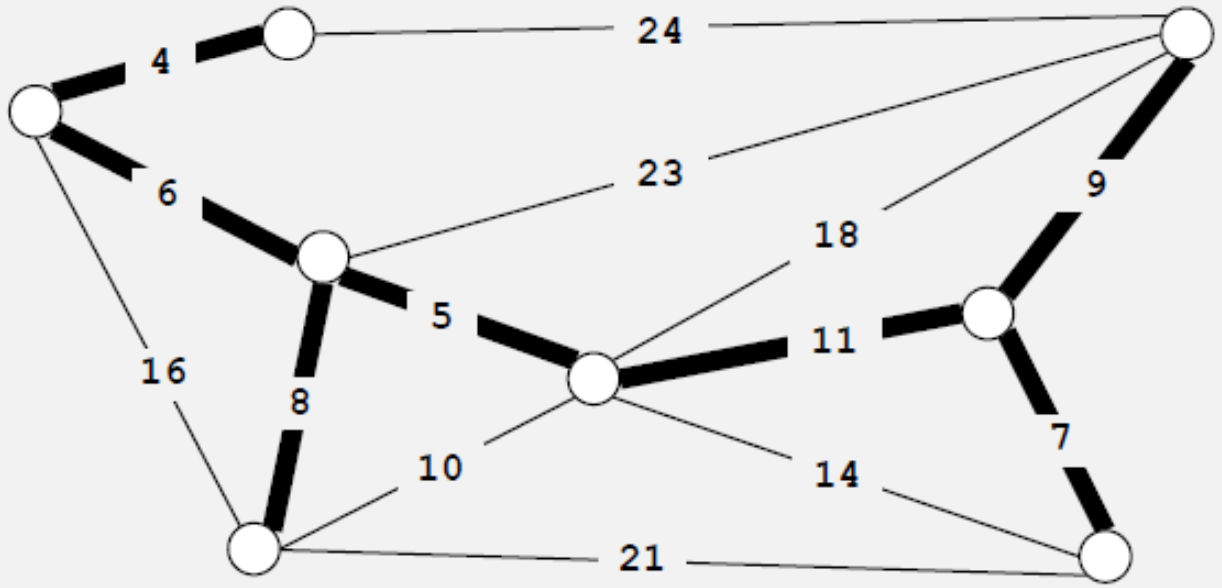


Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

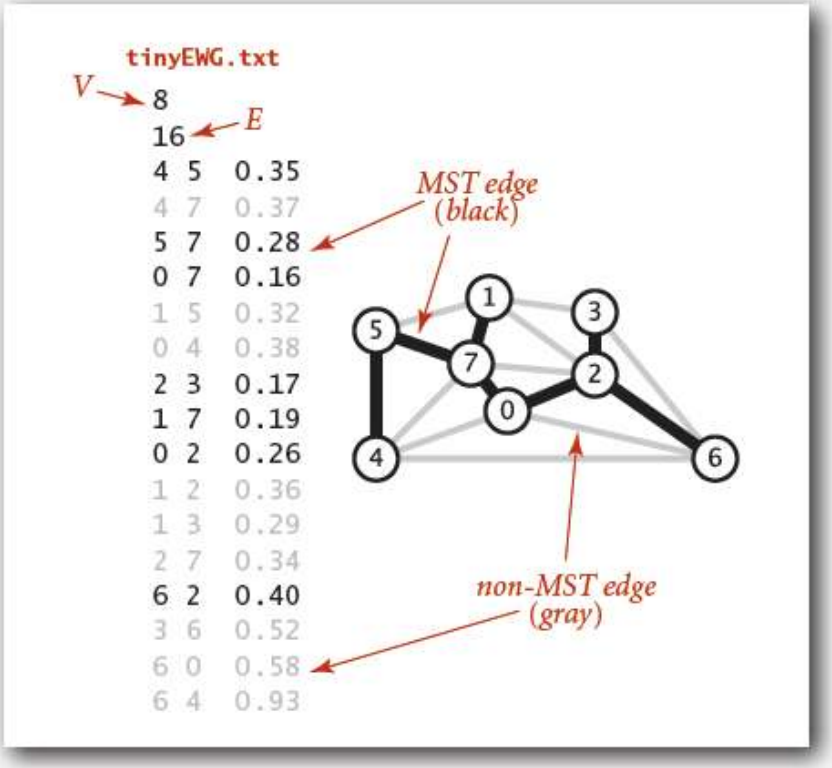
Applications

- Phone/cable network design – minimum cost
- Approximation algorithms for NP-hard problems

Minimum spanning tree API

Q. How to represent the MST?

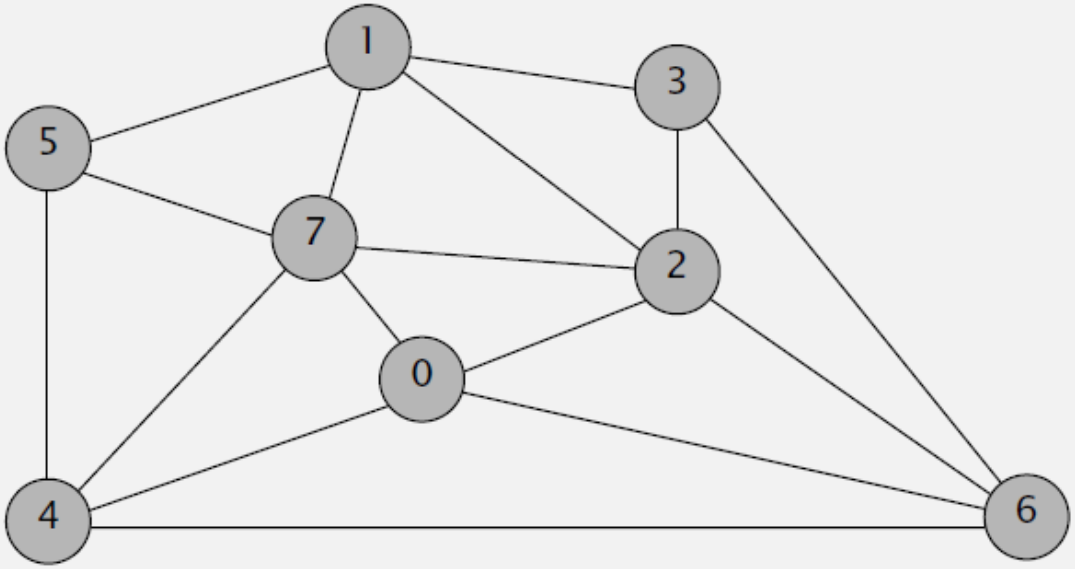
public class MST	
MST(EdgeWeightedGraph G)	*constructor*
Iterable<Edge> edges()	*edges in MST*
double weight()	*weight of MST*




```
% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Prim's algorithm

- Start with vertex 0 and greedily grow tree T.
- At each step, add to T the min weight edge with exactly one endpoint in T.



an edge-weighted graph

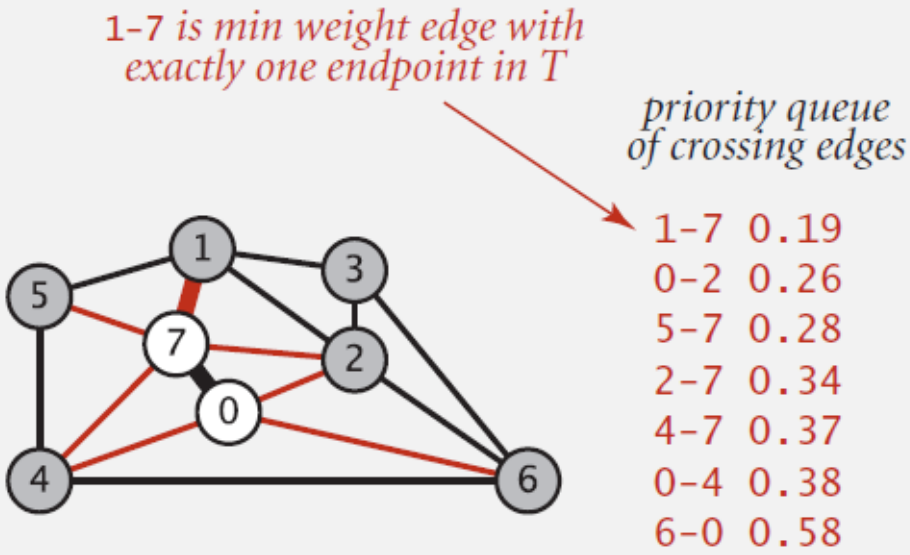
```
0-7  0.16
2-3  0.17
1-7  0.19
0-2  0.26
5-7  0.28
1-3  0.29
1-5  0.32
2-7  0.34
4-5  0.35
1-2  0.36
4-7  0.37
0-4  0.38
6-2  0.40
3-6  0.52
6-0  0.58
6-4  0.93
```

Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Delete min to determine next edge $e = v$–w to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let v be vertex not in T:
  - add to PQ any edge incident to v (assuming other endpoint not in T)
  - add v to T



Prim's algorithm demo: lazy implementation

Use `MinPQ`: key = edge, prioritized by weight.

(lazy version leaves some obsolete edges on the PQ)

Prim's algorithm: lazy implementation

```
public class LazyPrimMST
{
   private boolean[] marked;    // MST vertices
   private Queue<Edge> mst;     // MST edges
   private MinPQ<Edge> pq;      // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);

        while (!pq.isEmpty())
        {
           Edge e = pq.delMin();
           int v = e.either(), w = e.other(v);
           if (marked[v] && marked[w]) continue;
           mst.enqueue(e);
           if (!marked[v]) visit(G, v);
           if (!marked[w]) visit(G, w);
        }
   }
}
```

assume G is connected

repeatedly delete the min weight edge e = v–w from PQ

ignore if both endpoints in T

add edge e to tree

add v or w to tree

Prim's algorithm: lazy implementation

```
private void visit(WeightedGraph G, int v)
{
   marked[v] = true;
   for (Edge e : G.adj(v))
      if (!marked[e.other(v)])
         pq.insert(e);
}

public Iterable<Edge> mst()
{  return mst;  }
```

add v to T

for each edge e = v–w, add to PQ if w not already in T

Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to S.