http://algs4.cs.princeton.edu

4.3 MINIMUM SPANNING TREES

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- ***Kruskal's algorithm***
- *Prim's algorithm*
- *context*

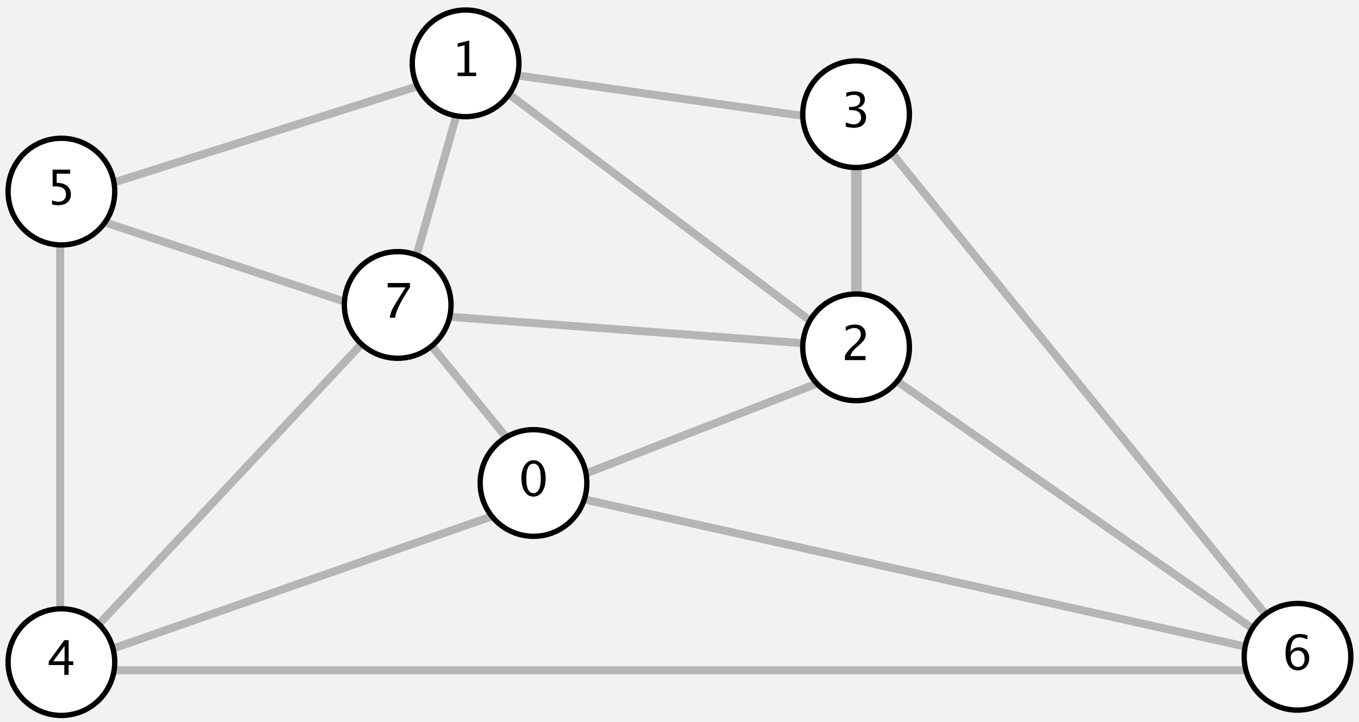
http://algs4.cs.princeton.edu

KRUSKAL'S ALGORITHM DEMO

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



an edge-weighted graph

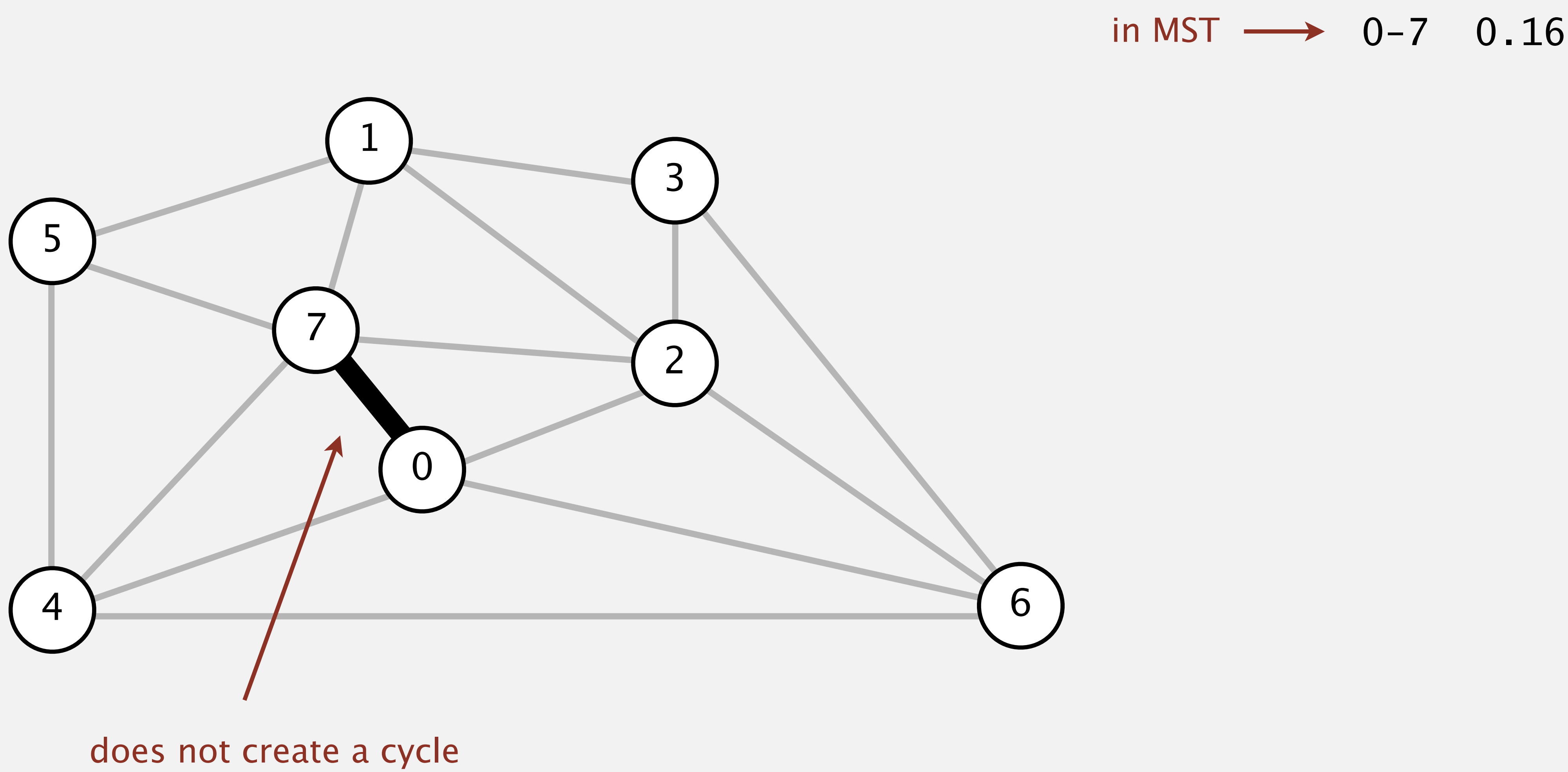
graph edges sorted by weight

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

Kruskal's algorithm demo

Consider edges in ascending order of weight.

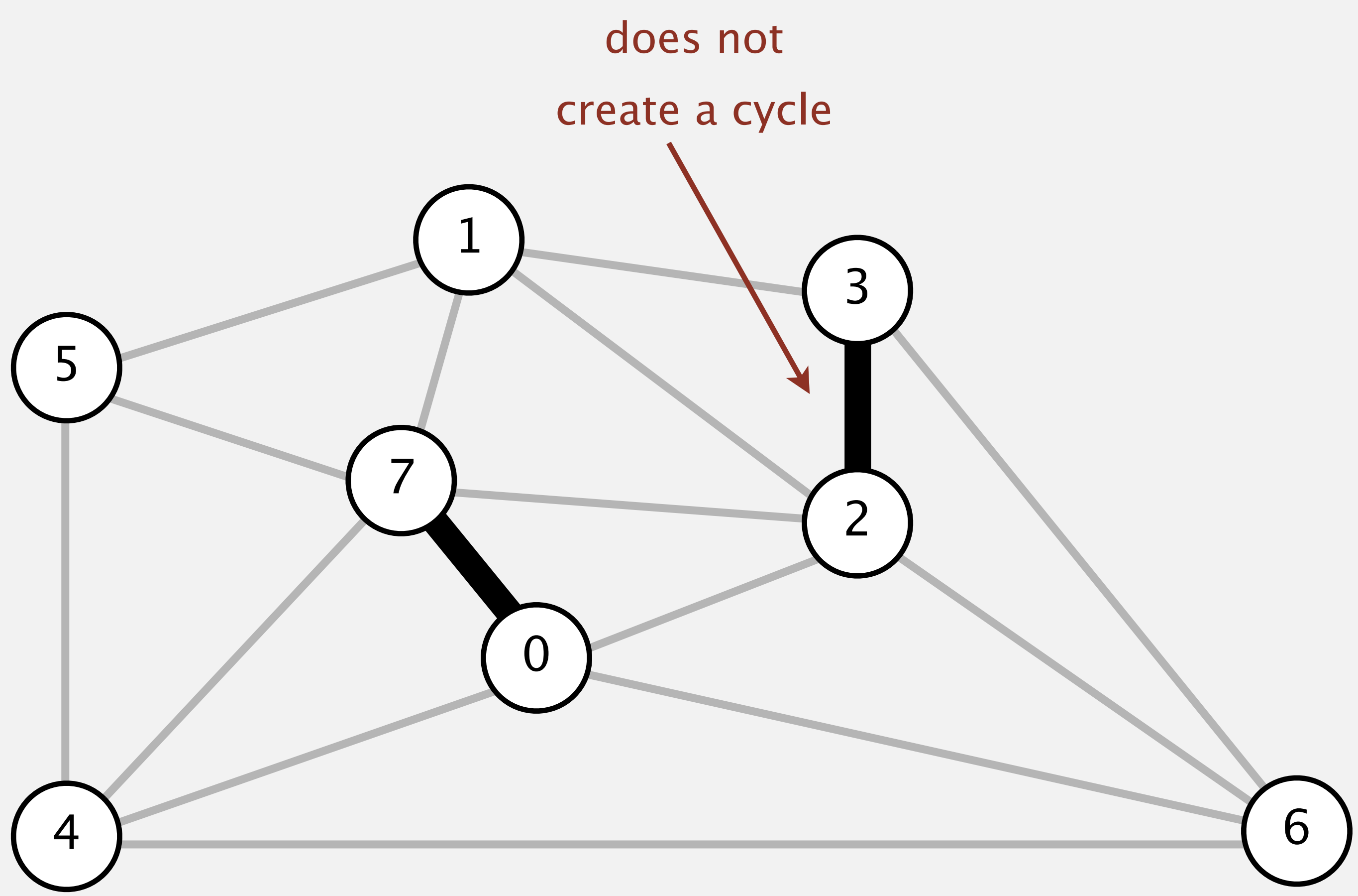
- Add next edge to tree *T* unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



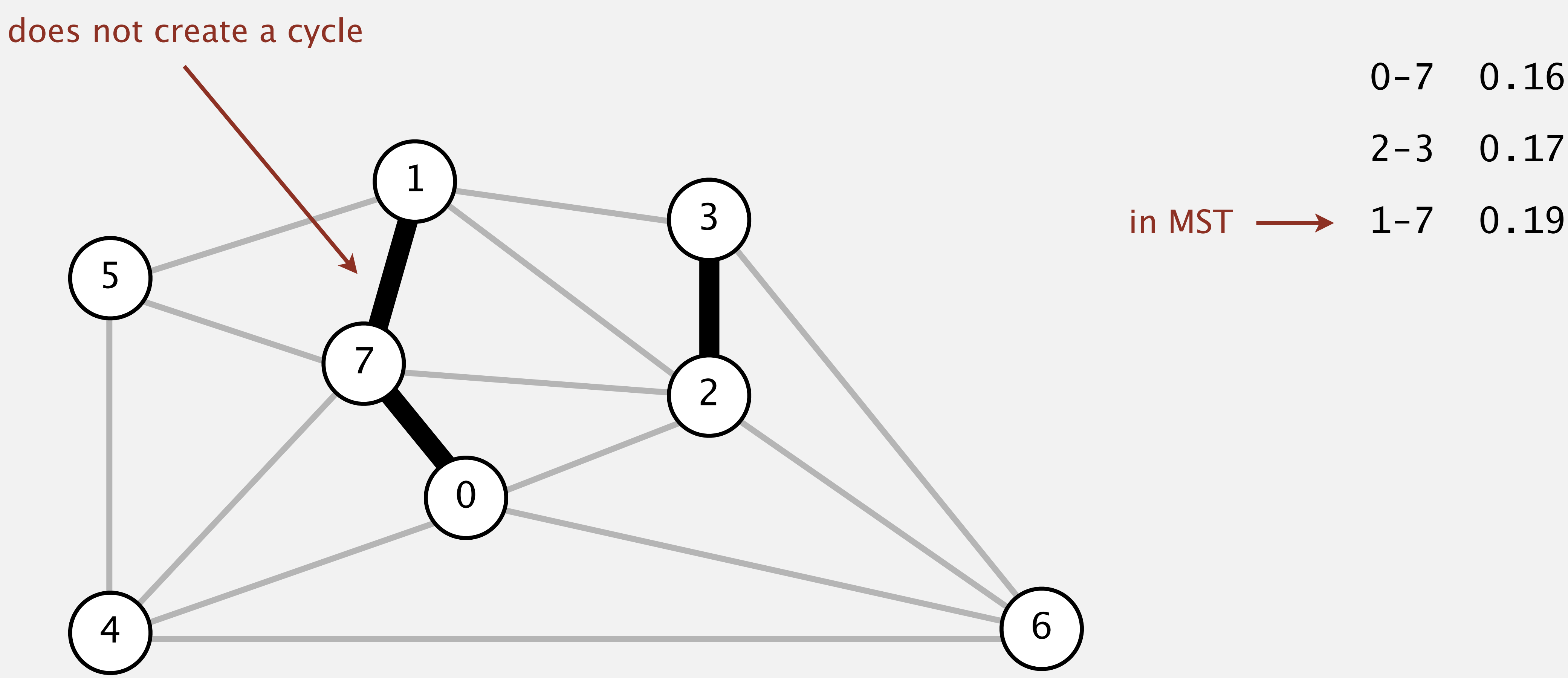
in MST →

0-7	0.16
2-3	0.17

Kruskal's algorithm demo

Consider edges in ascending order of weight.

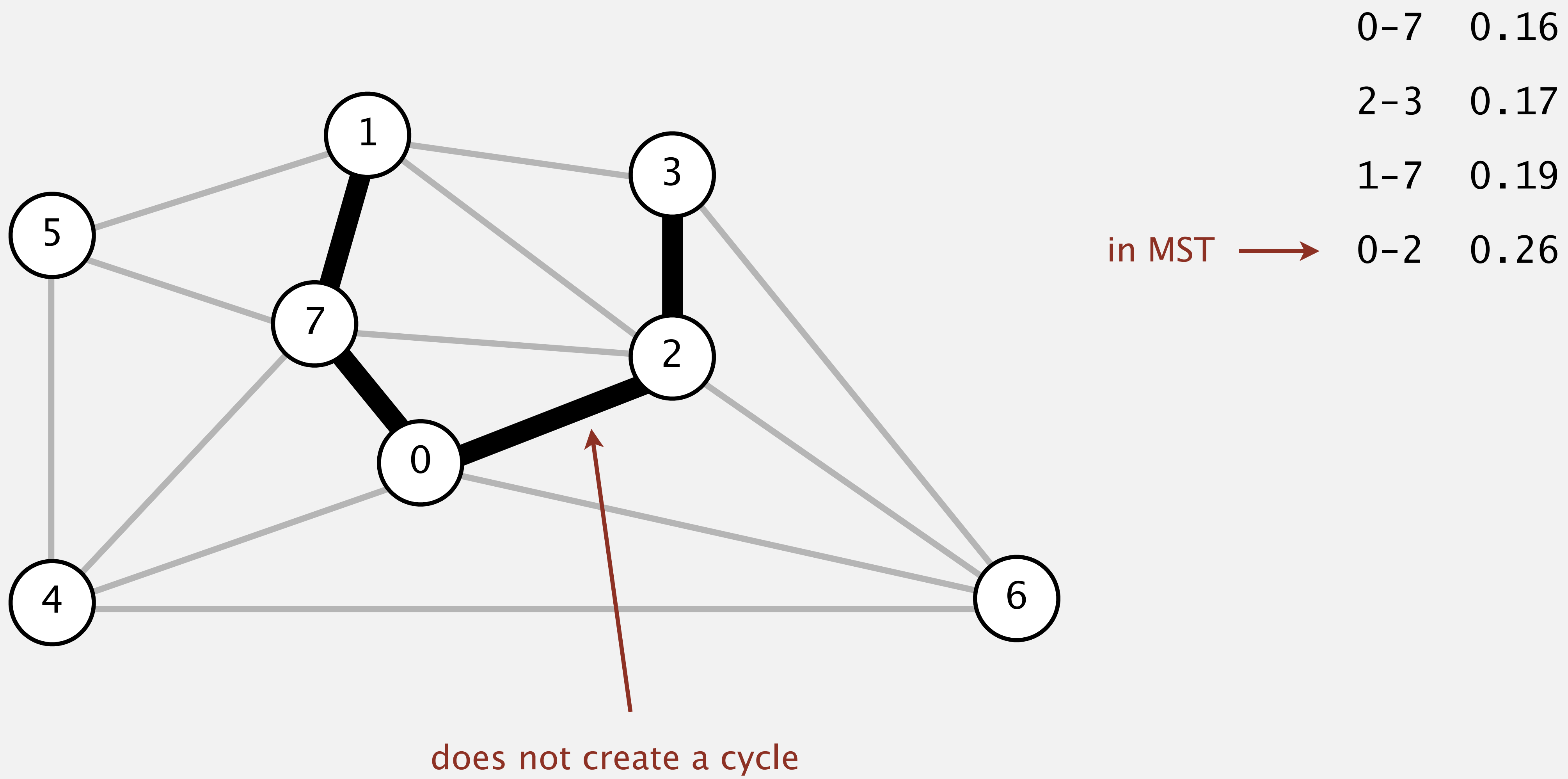
- Add next edge to tree *T* unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

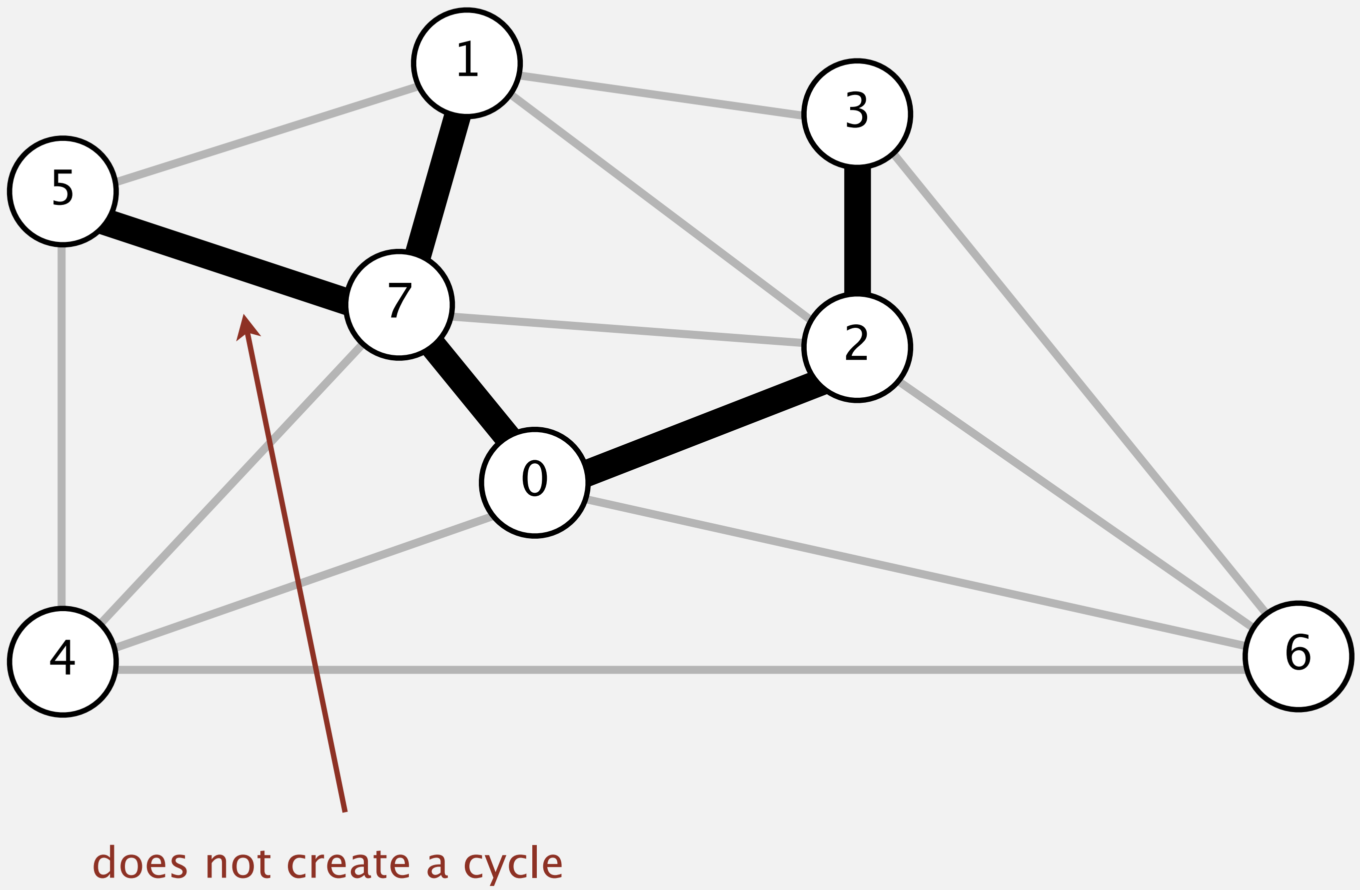
- Add next edge to tree *T* unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



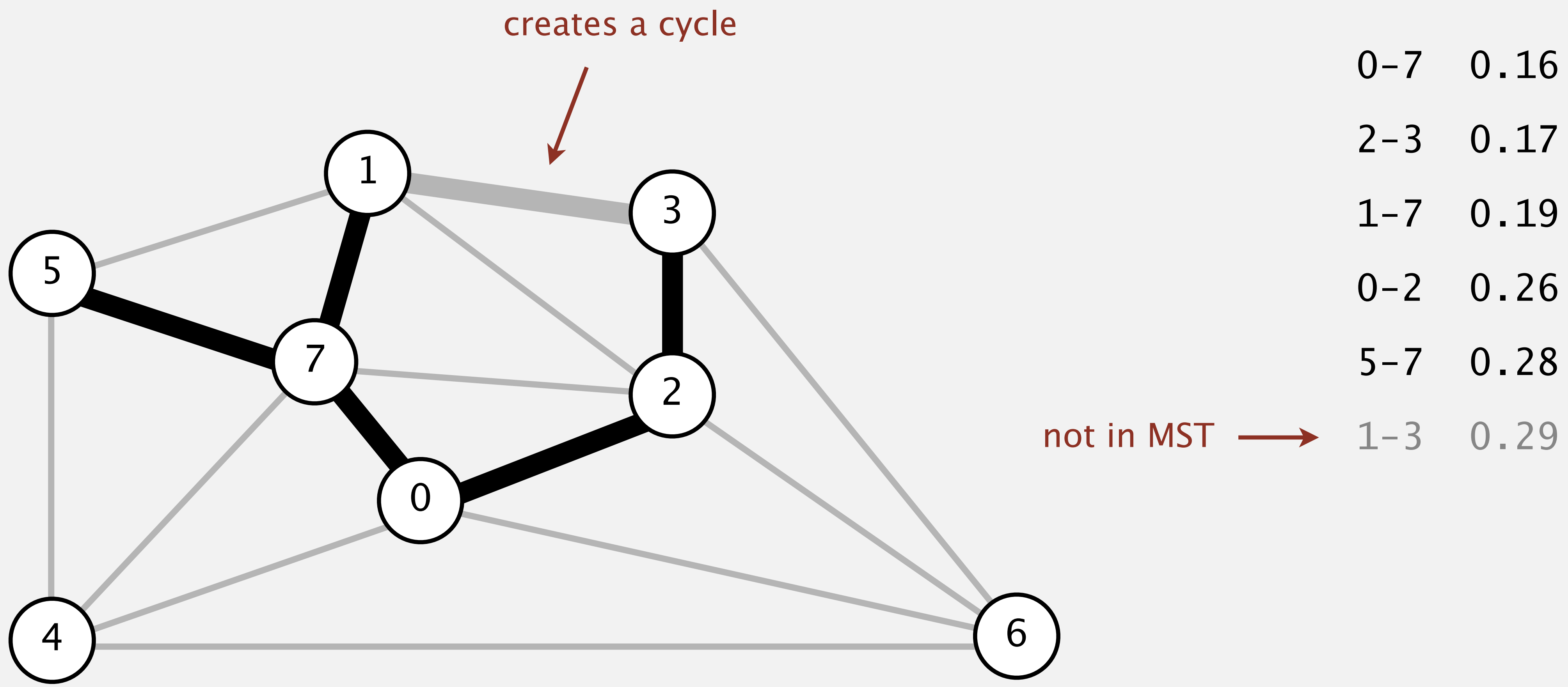
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28

in MST → 5-7 0.28

Kruskal's algorithm demo

Consider edges in ascending order of weight.

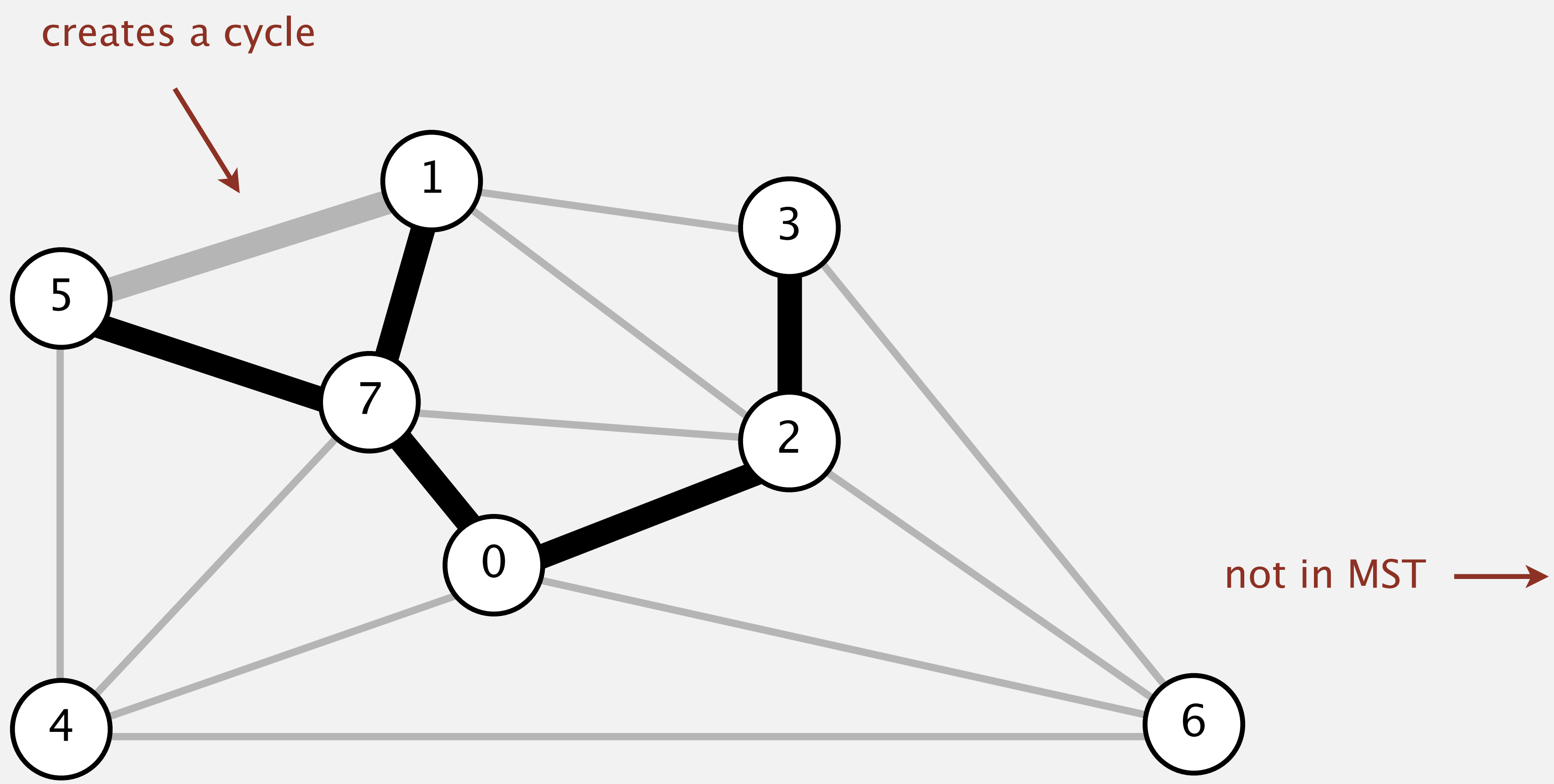
- Add next edge to tree *T* unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.

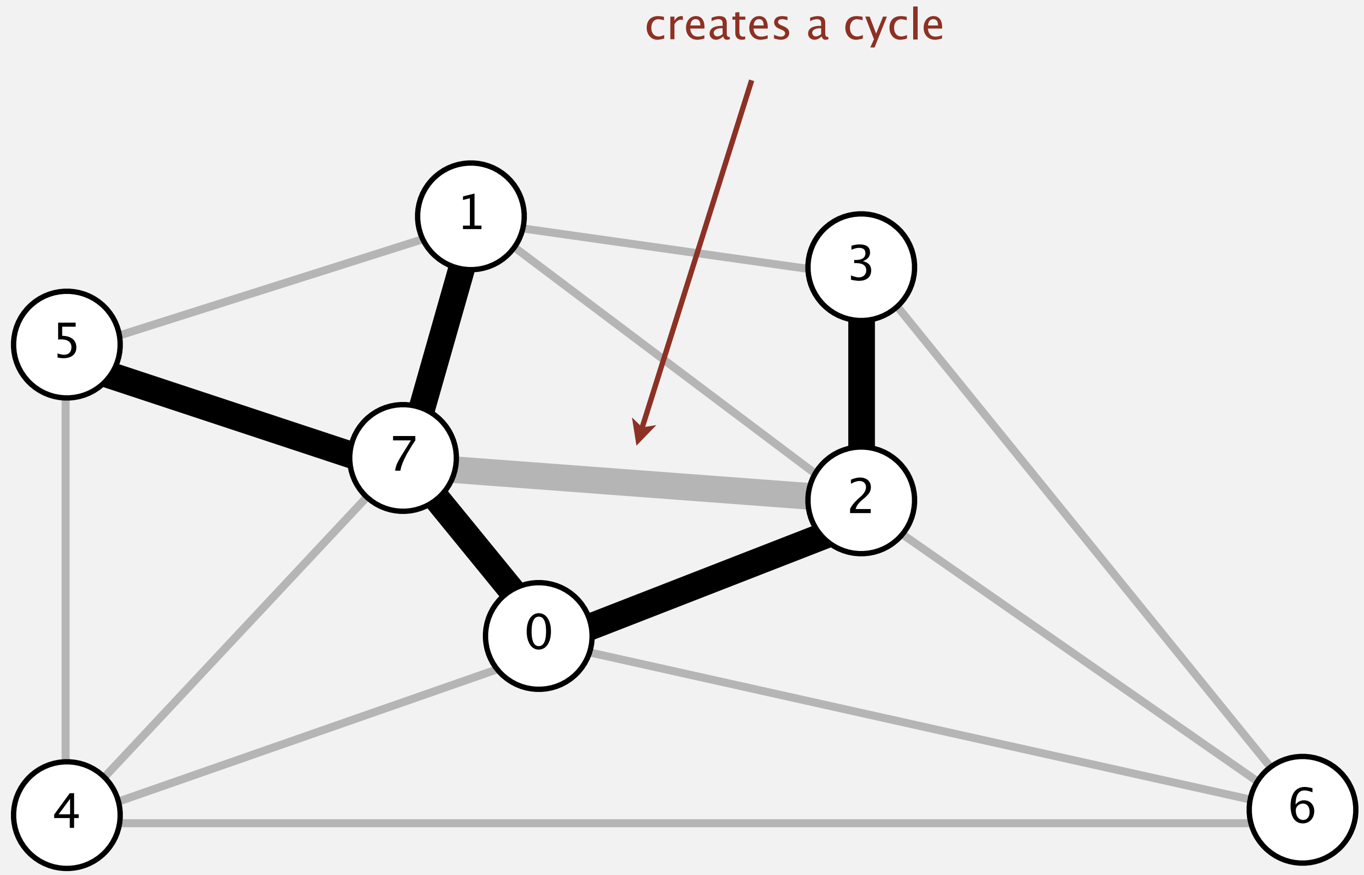


0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



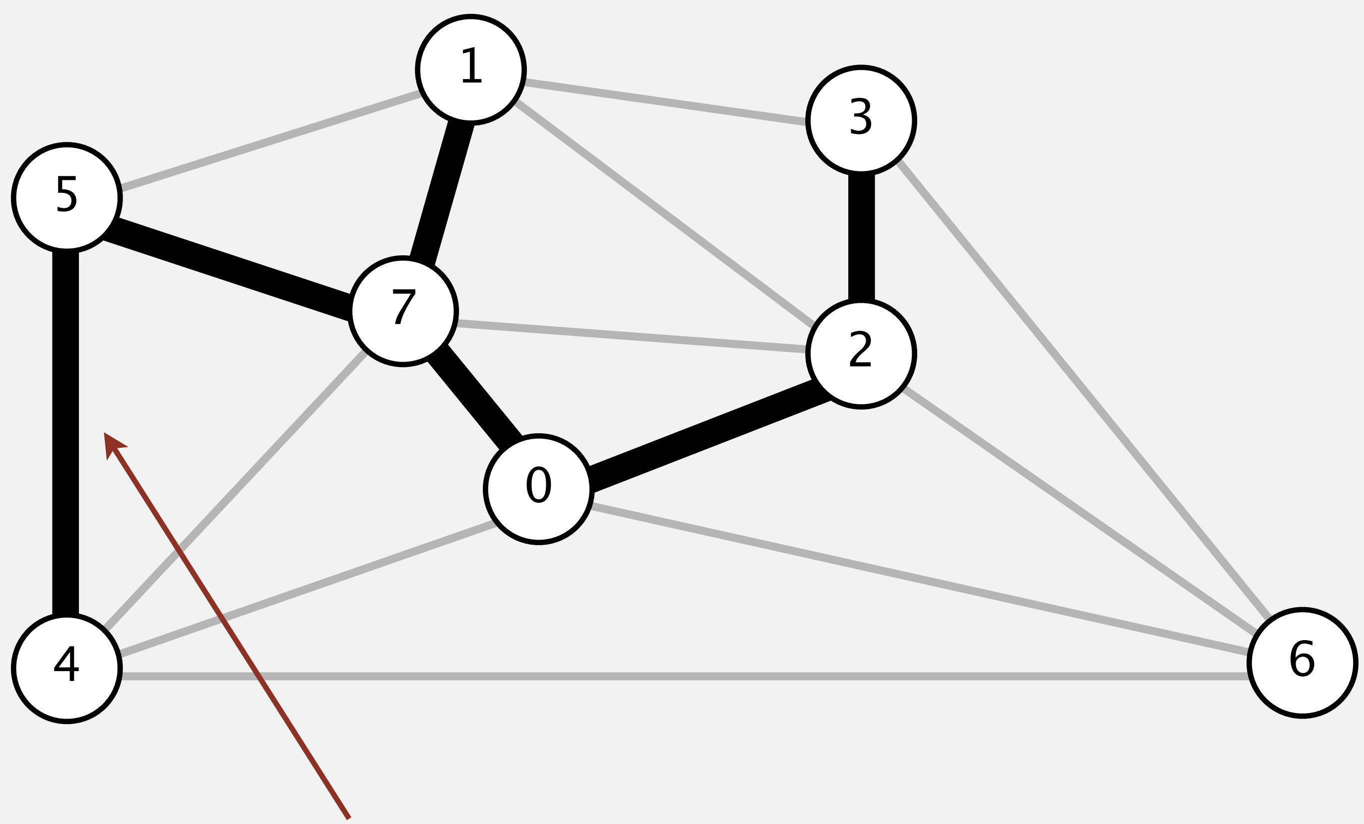
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34

not in MST → 2-7 0.34

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



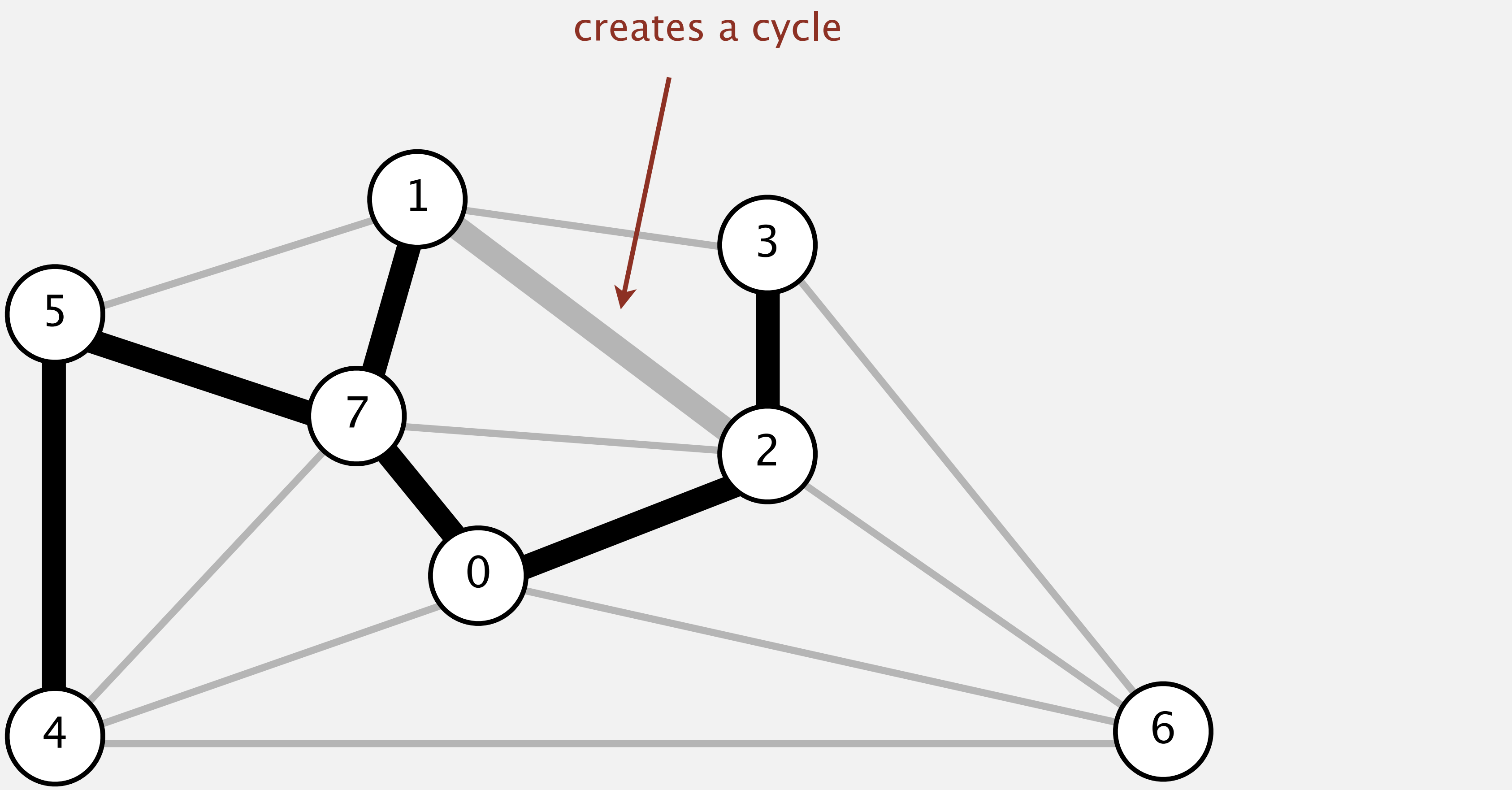
does not create a cycle

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
in MST → 4-5	0.35

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.

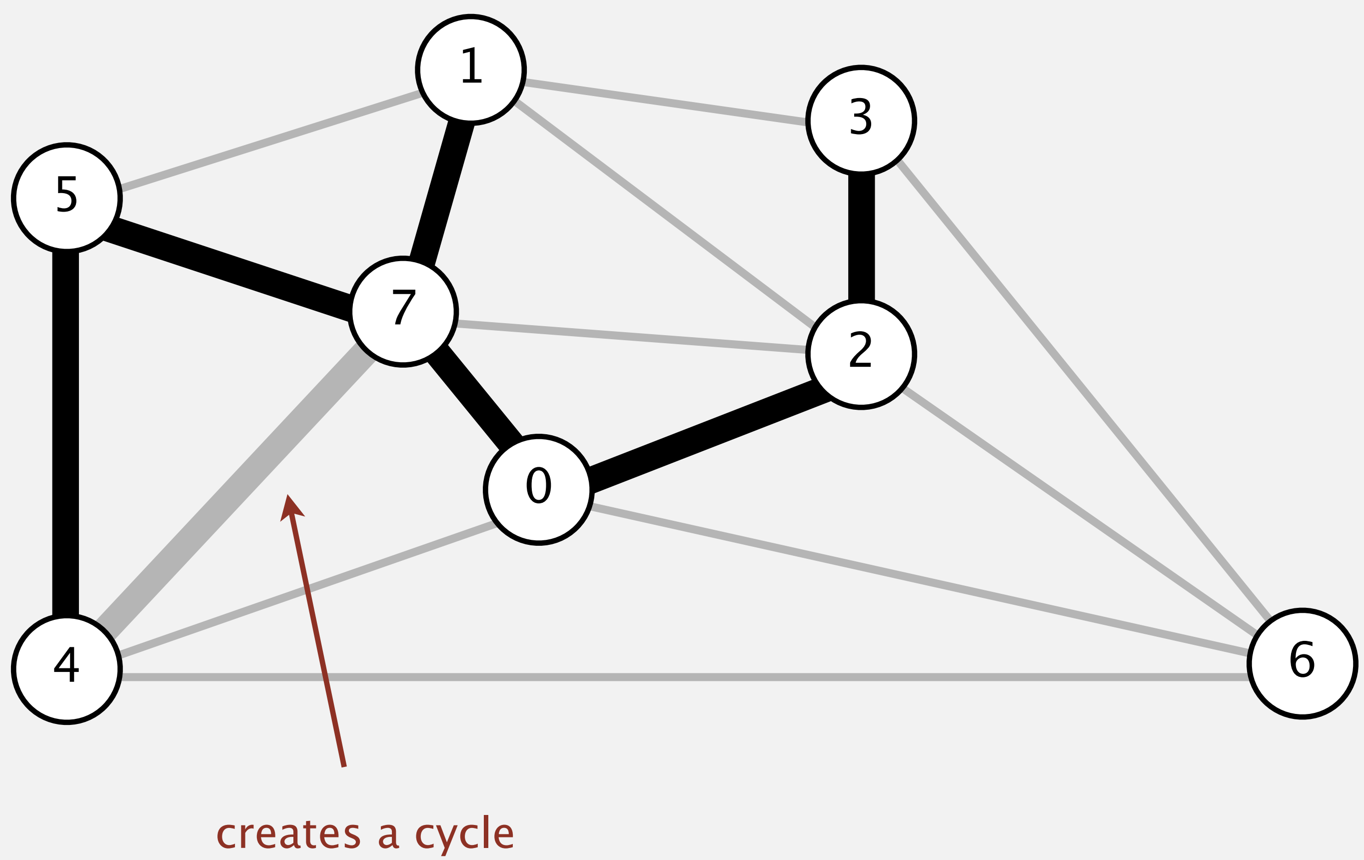


0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.

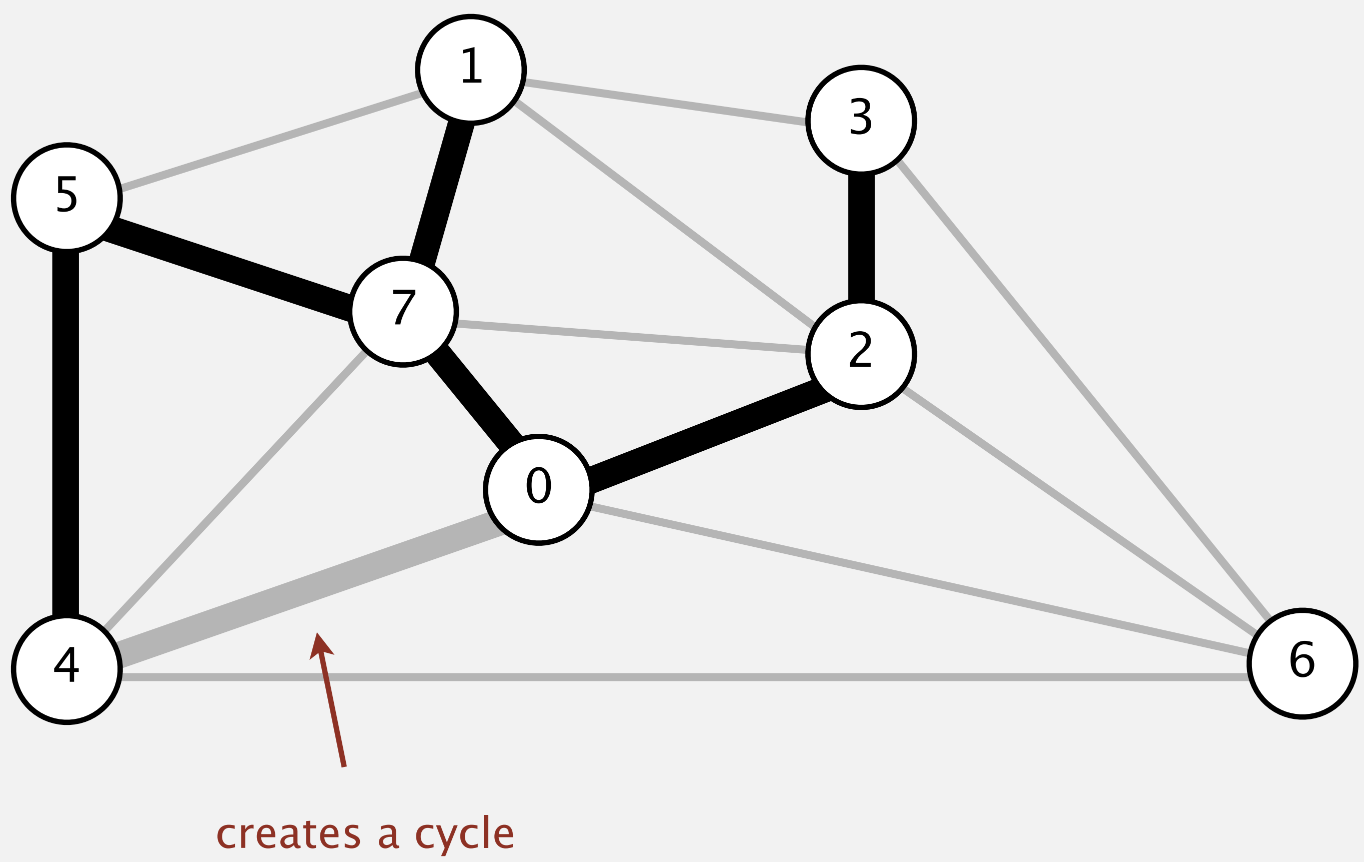


	0-7	0.16
	2-3	0.17
	1-7	0.19
	0-2	0.26
	5-7	0.28
	1-3	0.29
	1-5	0.32
	2-7	0.34
	4-5	0.35
	1-2	0.36
not in MST →	4-7	0.37

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.

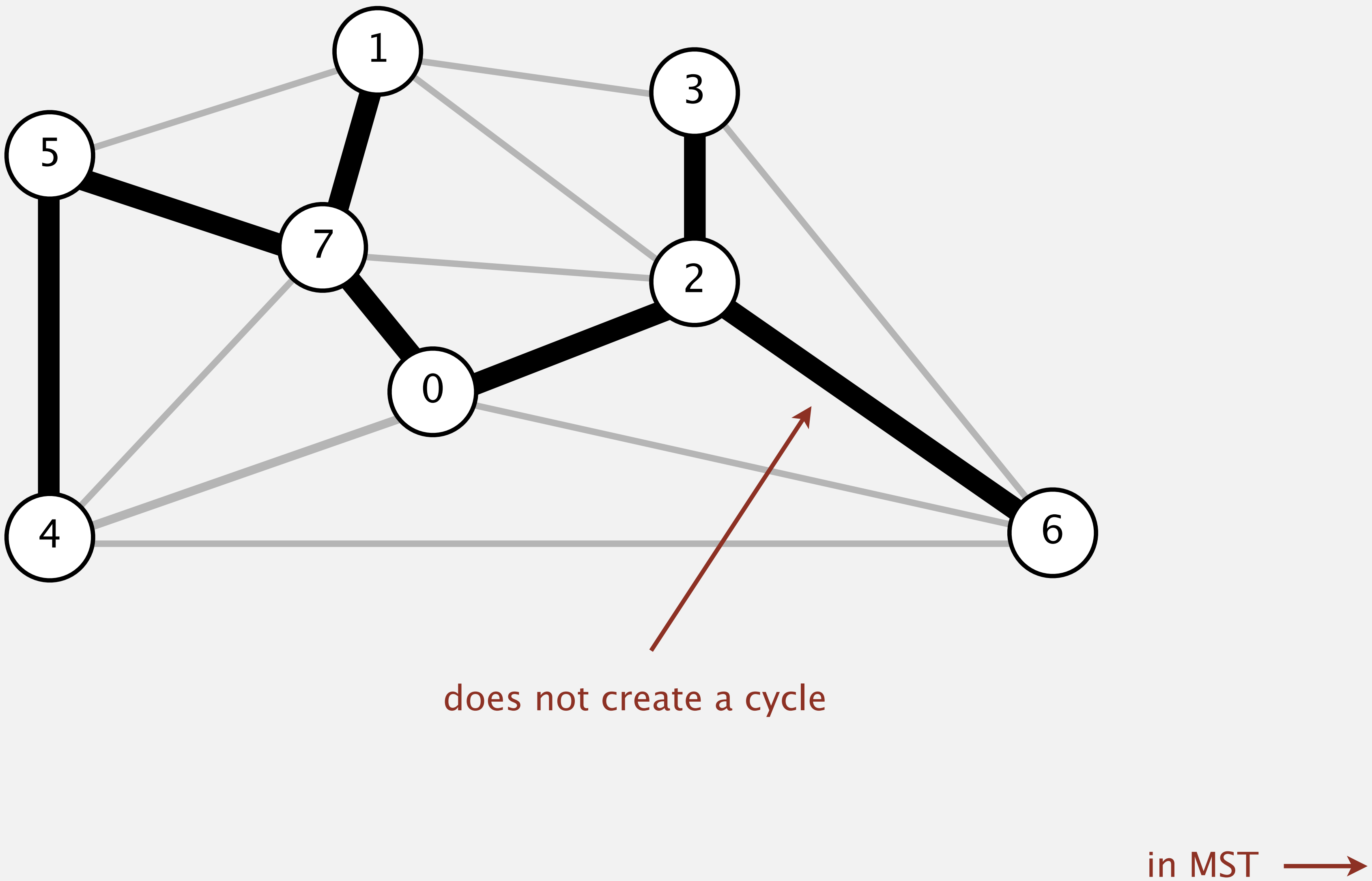


	0-7	0.16
	2-3	0.17
	1-7	0.19
	0-2	0.26
	5-7	0.28
	1-3	0.29
	1-5	0.32
	2-7	0.34
	4-5	0.35
	1-2	0.36
	4-7	0.37
not in MST →	0-4	0.38

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.

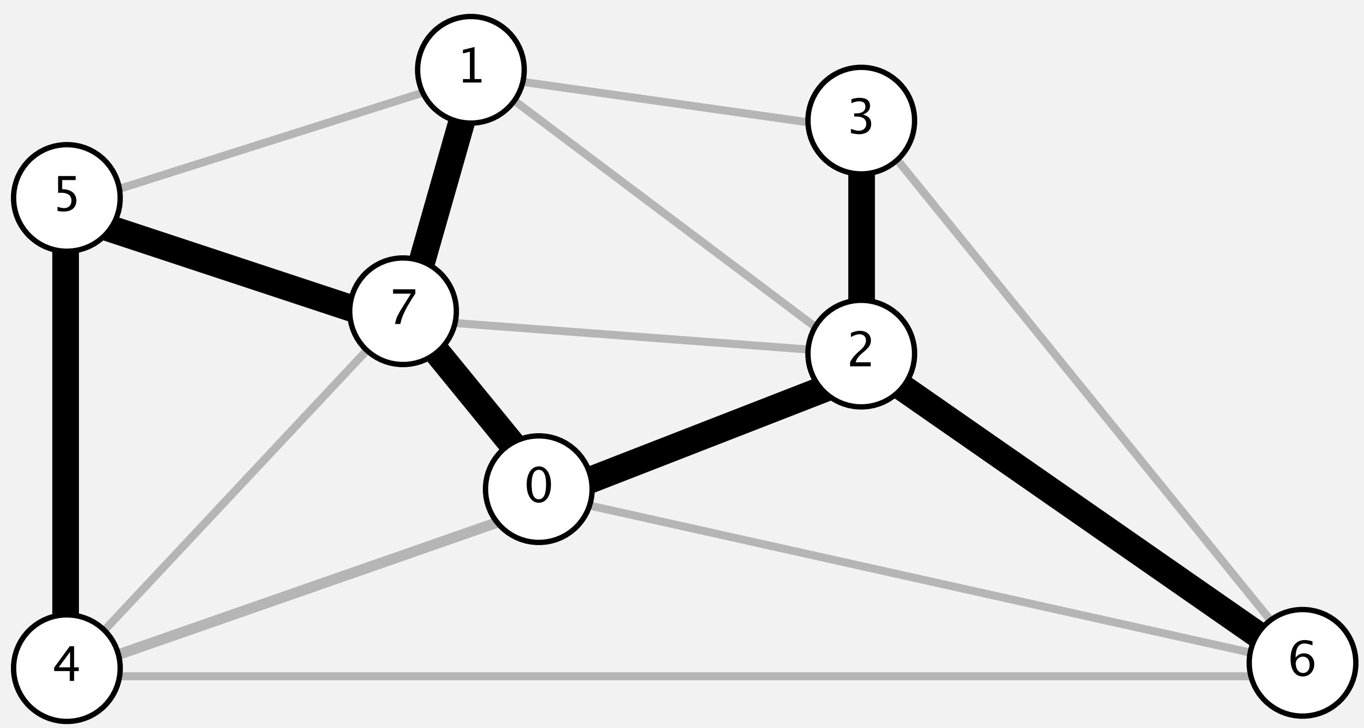


0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree *T* unless doing so would create a cycle.



a minimum spanning tree

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

Kruskal's algorithm: visualization

Kruskal's algorithm: visualization

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

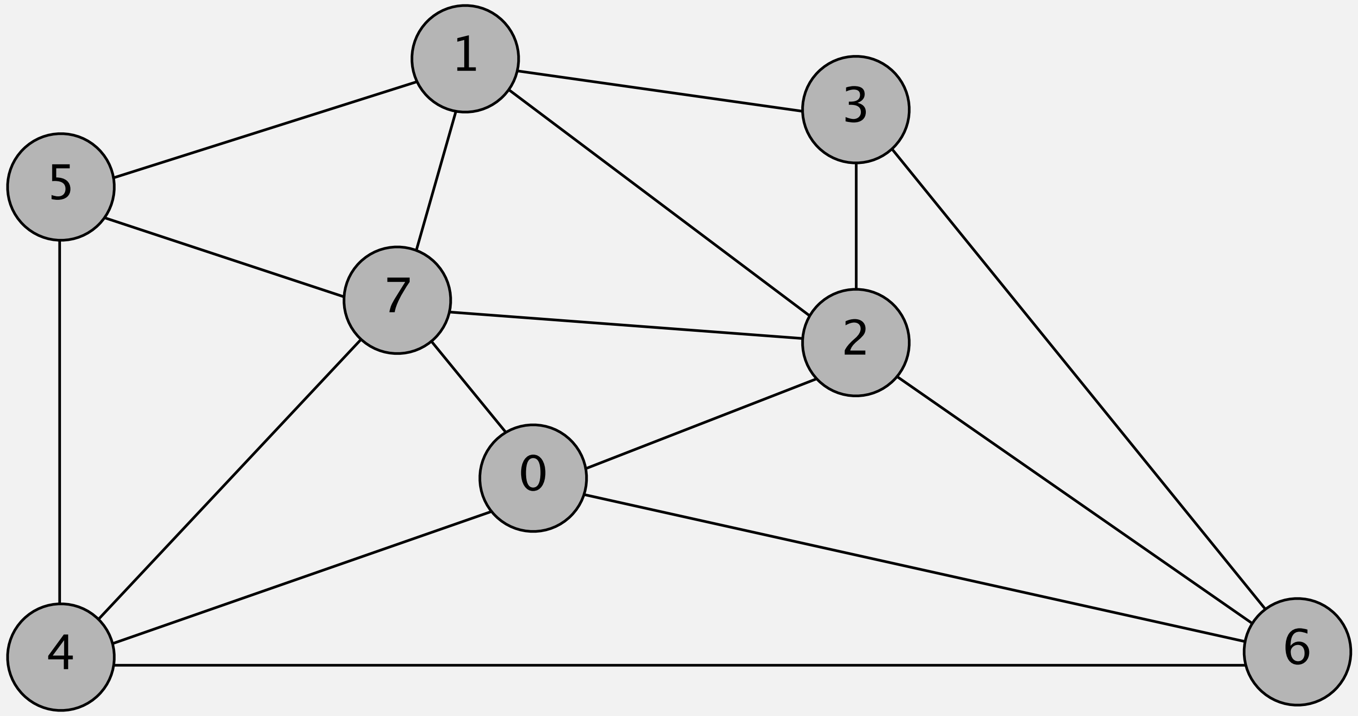
http://algs4.cs.princeton.edu

Prim's Algorithm Demo

- *Prim's algorithm*
- *lazy implementation*
- *eager implementation*

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

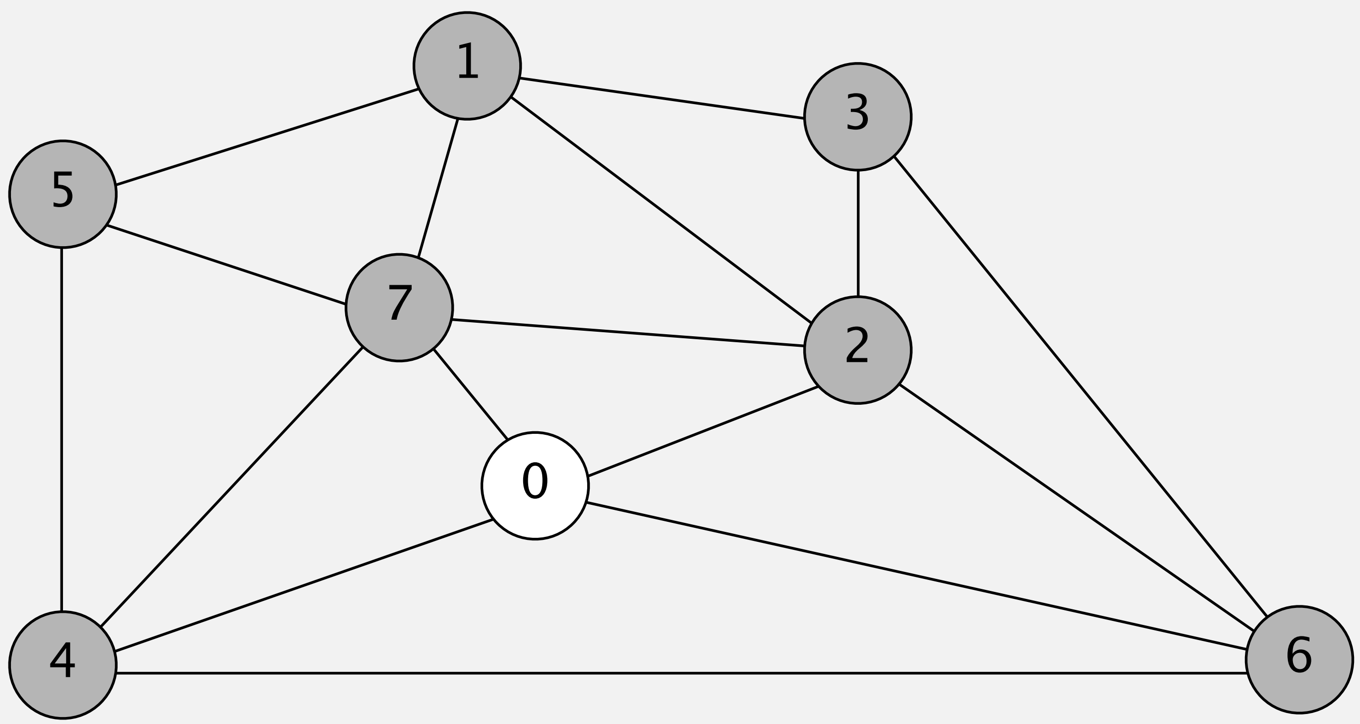


an edge-weighted graph

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

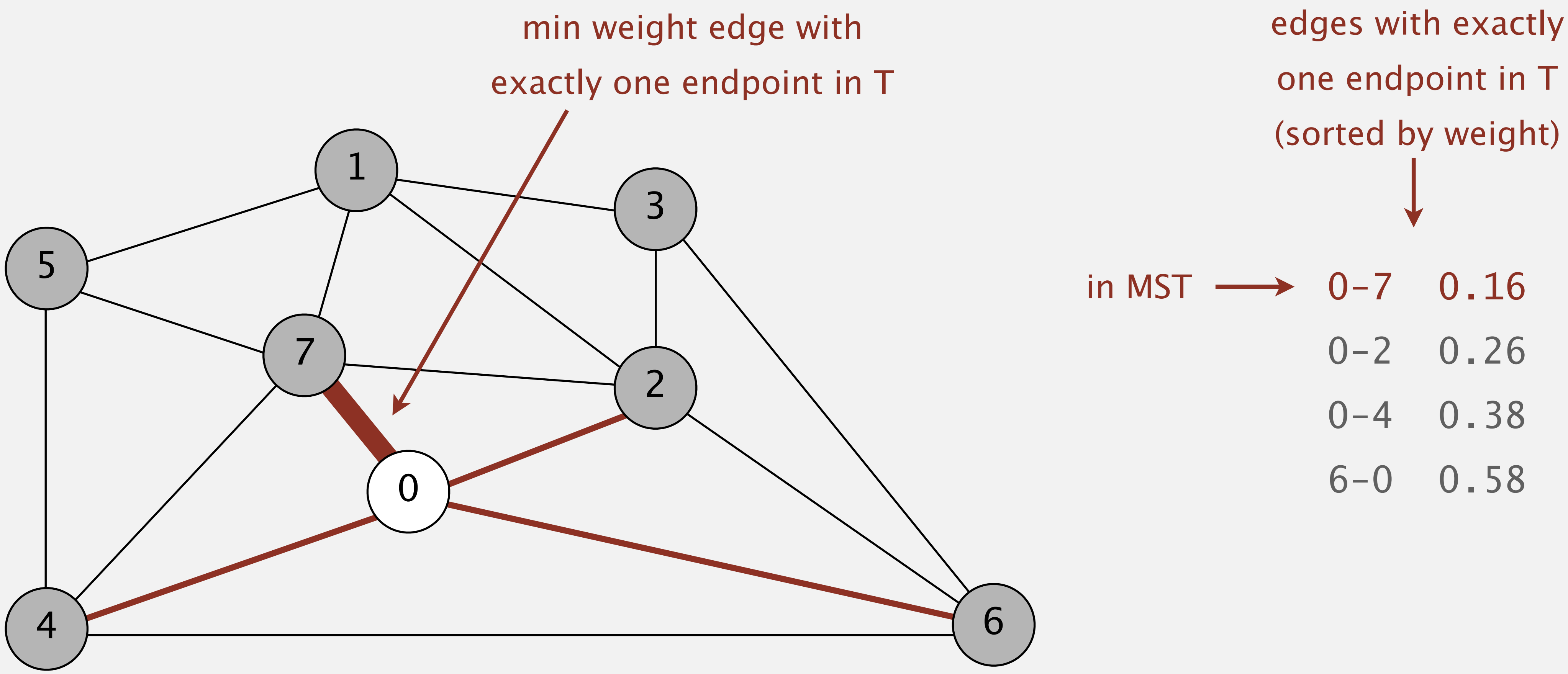
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



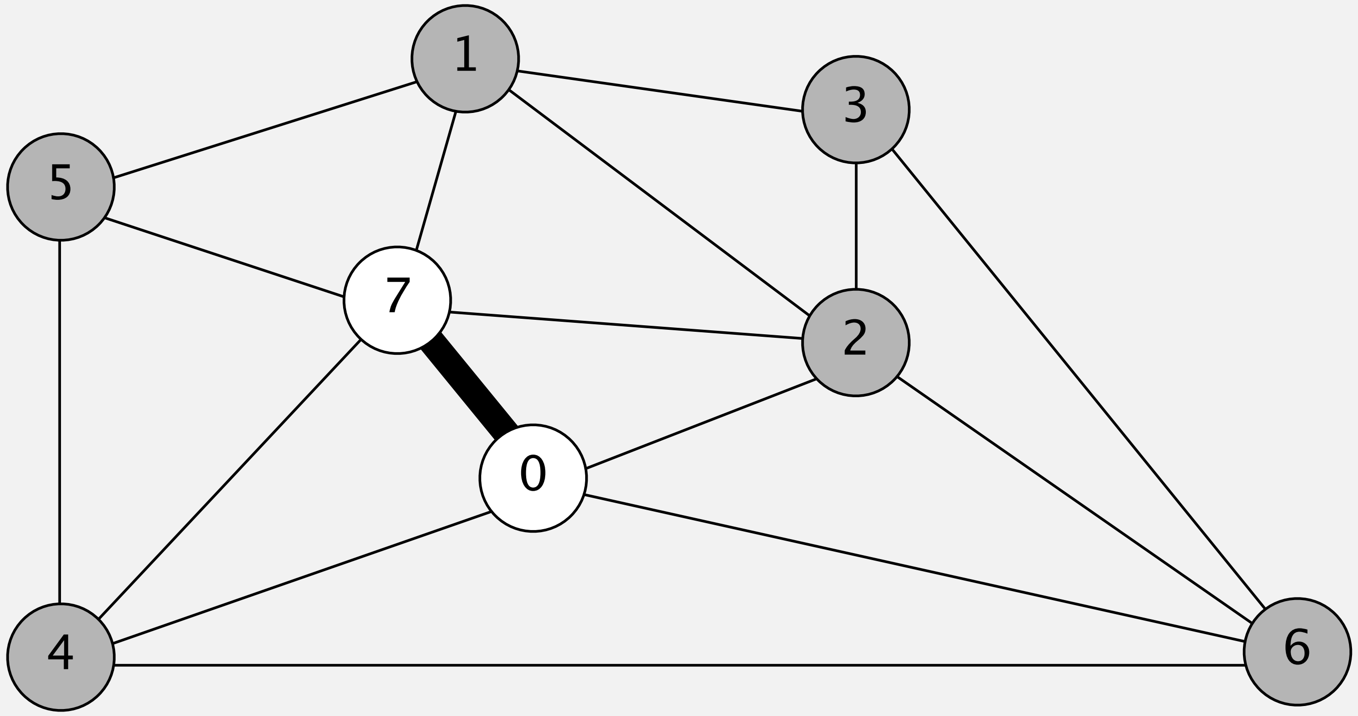
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

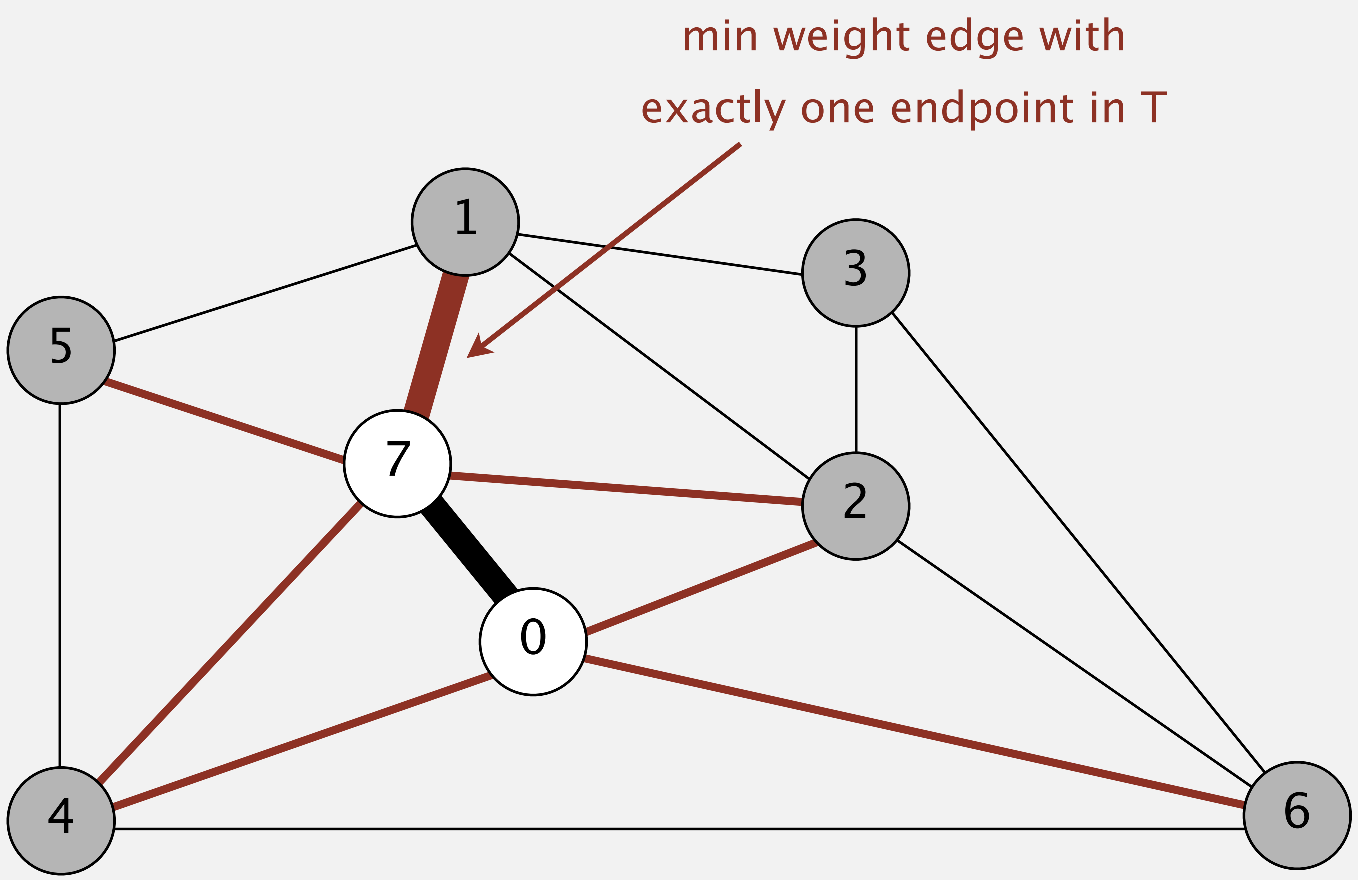


MST edges

0–7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



edges with exactly one endpoint in T (sorted by weight)

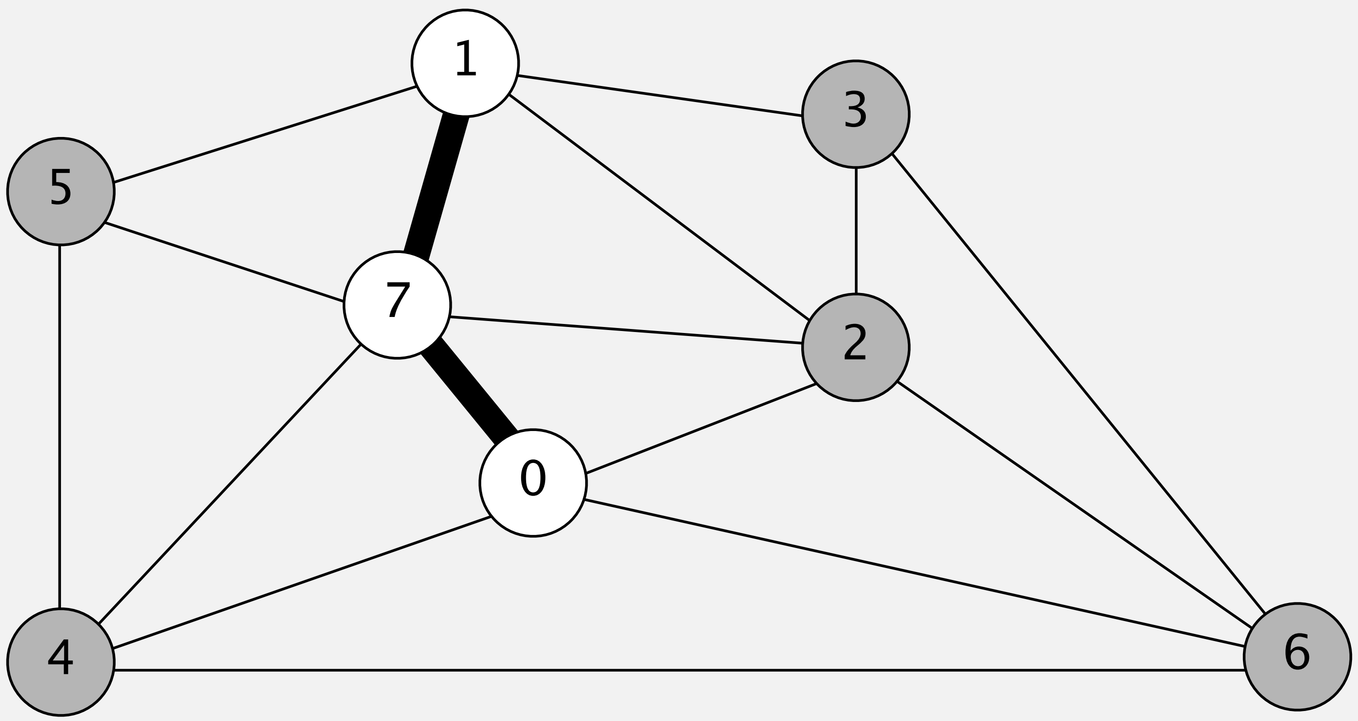
in MST →	1-7	0.19
	0-2	0.26
	5-7	0.28
	2-7	0.34
	4-7	0.37
	0-4	0.38
	6-0	0.58

MST edges

0-7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

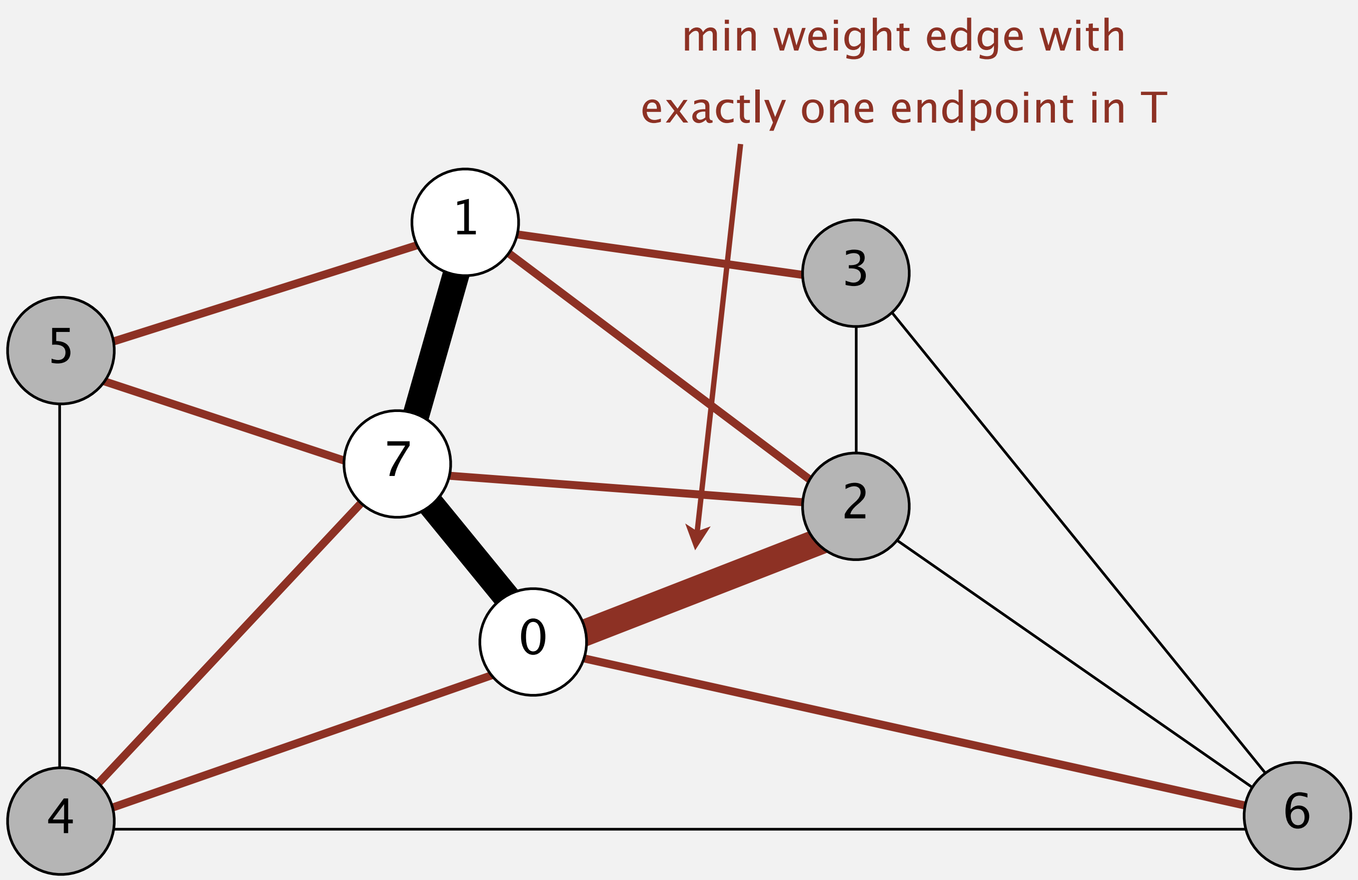


MST edges

0–7 1–7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



edges with exactly one endpoint in T (sorted by weight)

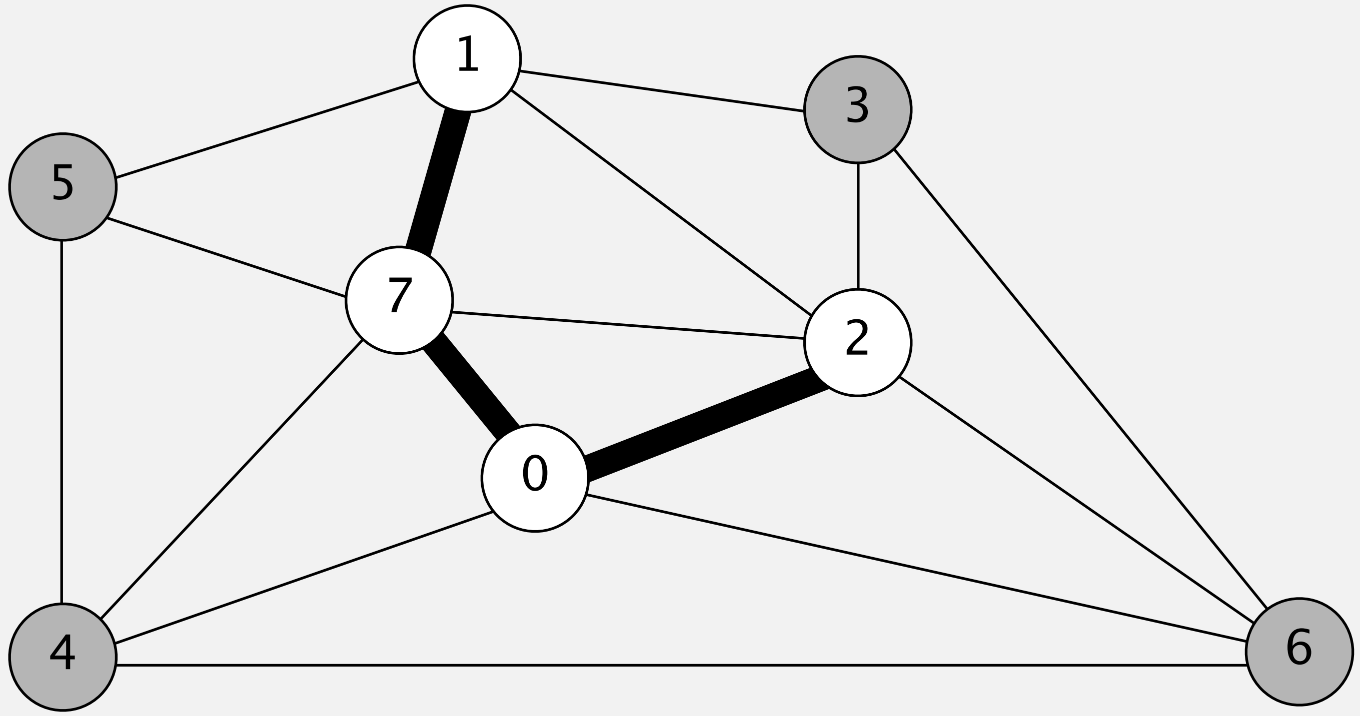
in MST →	0-2	0.26
	5-7	0.28
	1-3	0.29
	1-5	0.32
	2-7	0.34
	1-2	0.36
	4-7	0.37
	0-4	0.38
	6-0	0.58

MST edges

0-7 1-7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

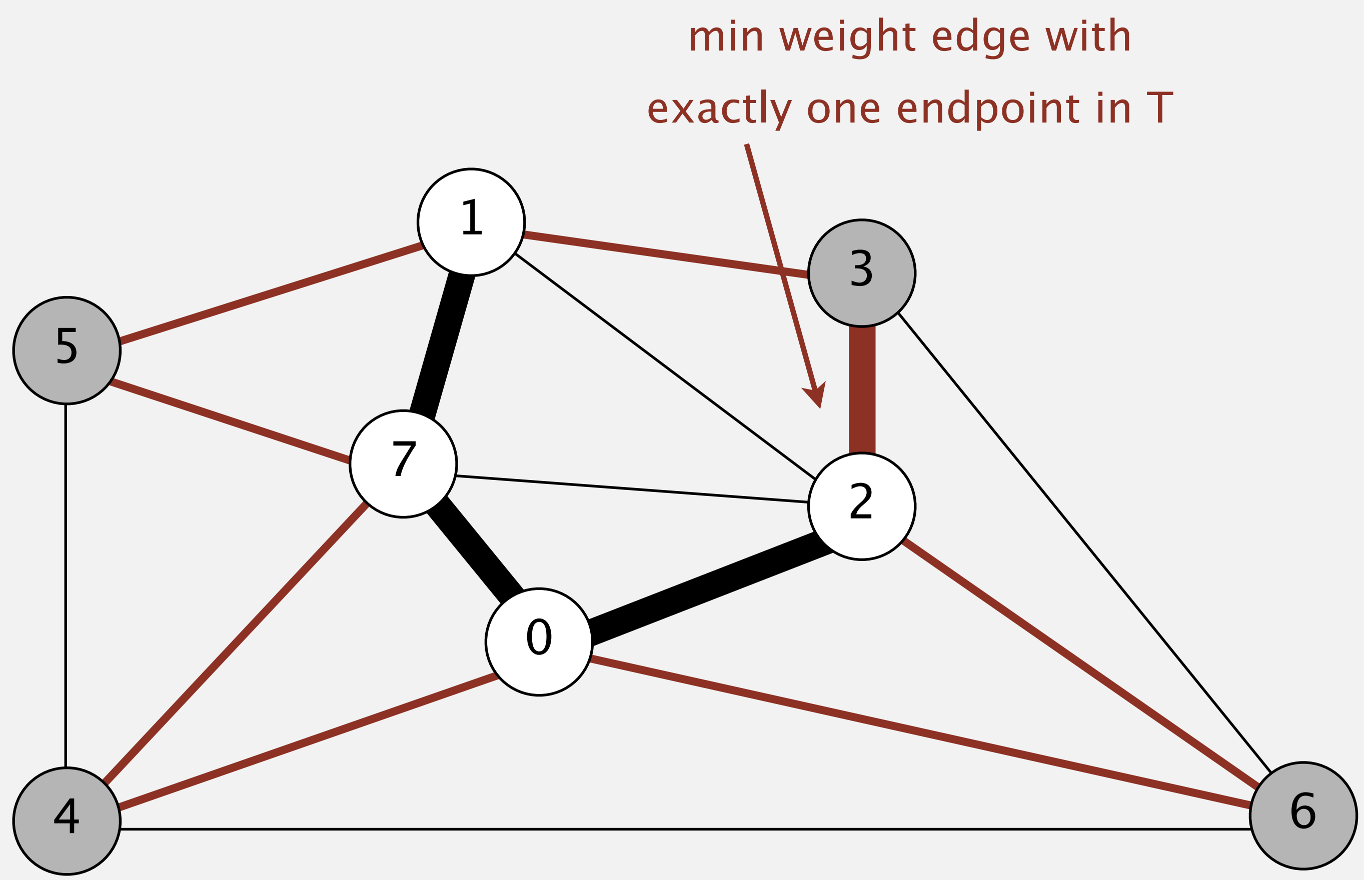


MST edges

0–7   1–7   0–2

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



edges with exactly one endpoint in T (sorted by weight)

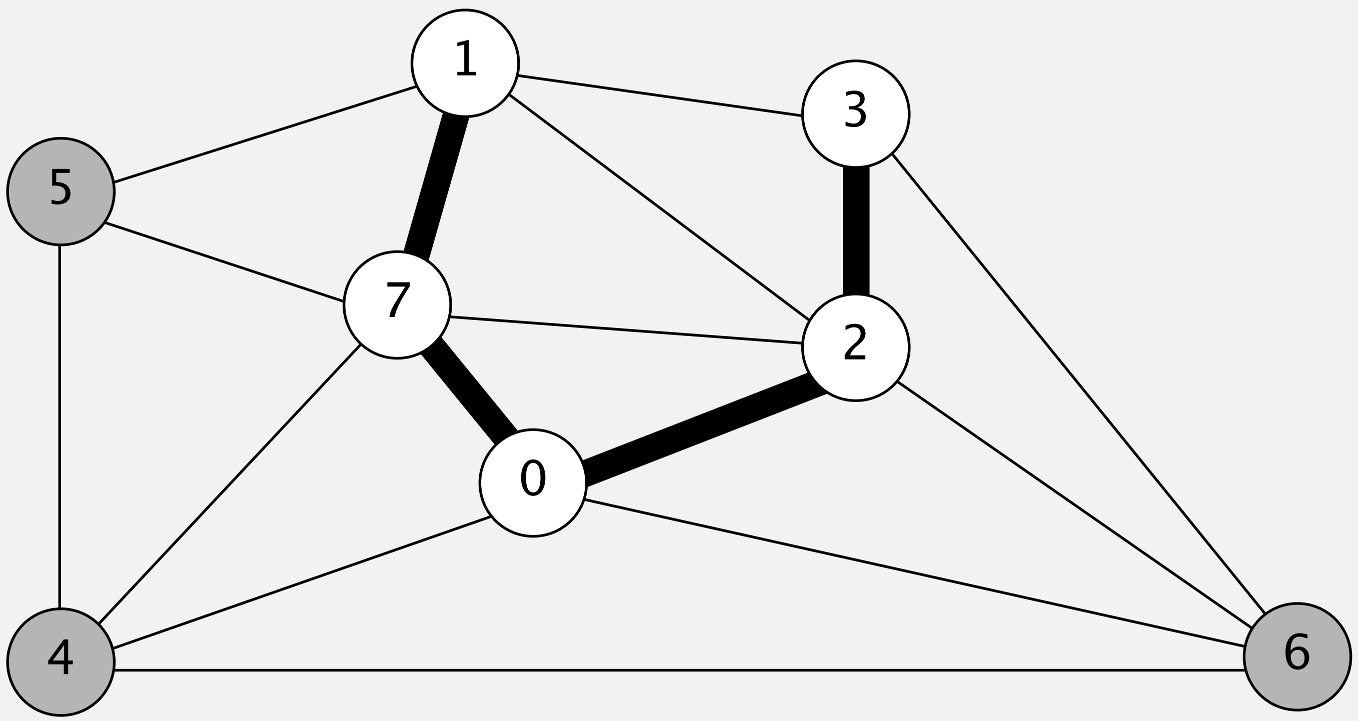
in MST →	2-3	0.17
	5-7	0.28
	1-3	0.29
	1-5	0.32
	4-7	0.37
	0-4	0.38
	6-2	0.40
	6-0	0.58

MST edges

0-7 1-7 0-2

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



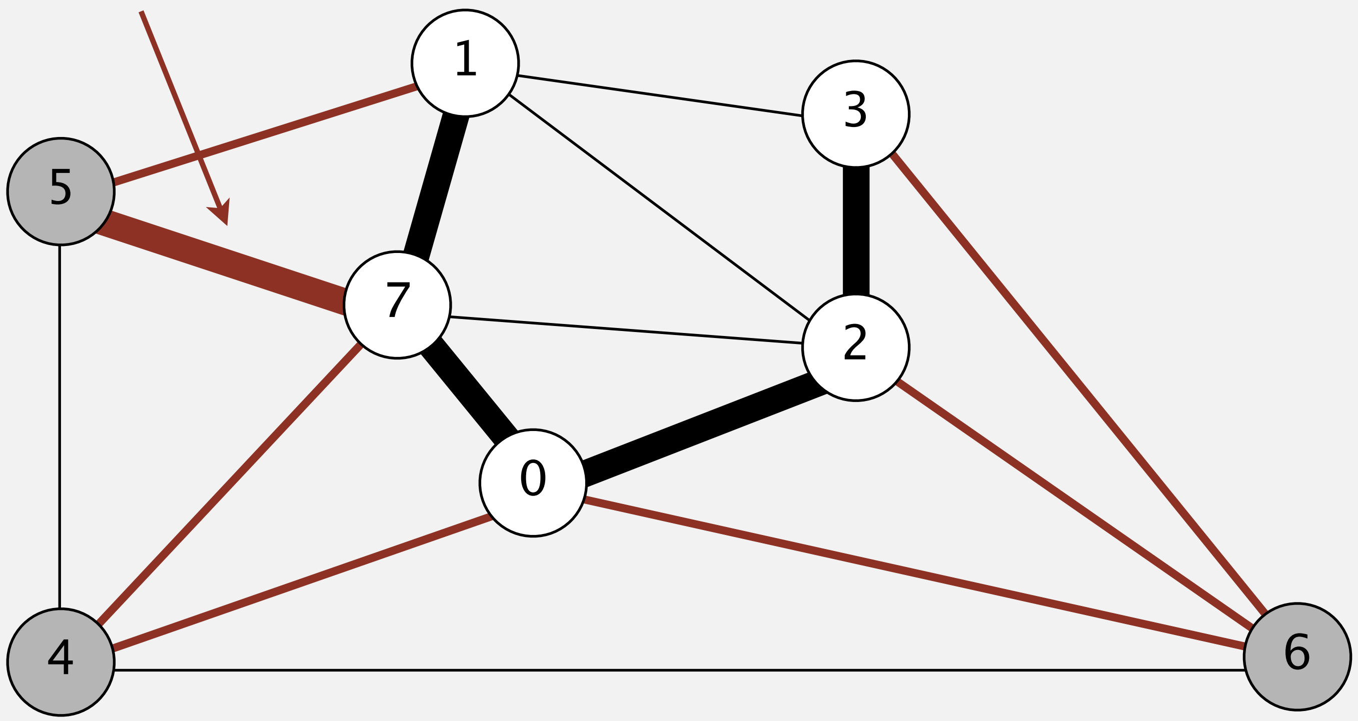
MST edges

0–7 1–7 0–2 2–3

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

min weight edge with exactly one endpoint in T



edges with exactly one endpoint in T (sorted by weight)

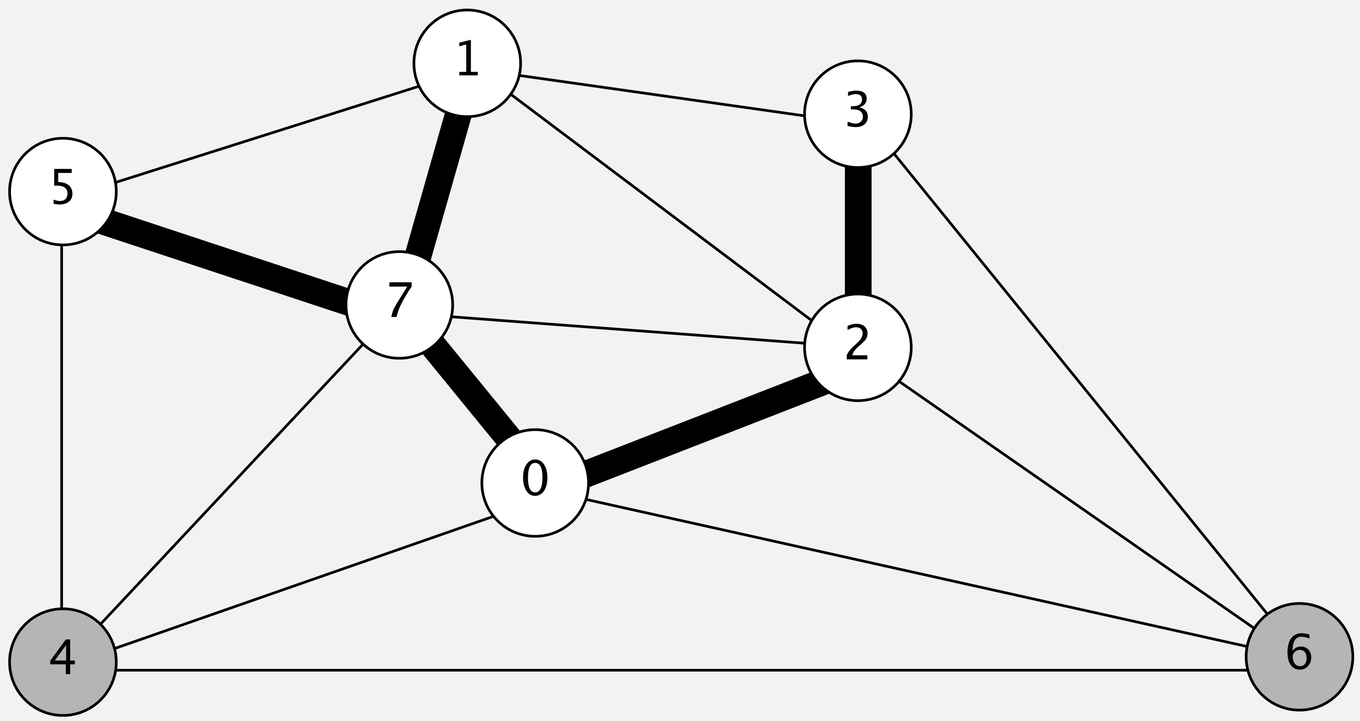
in MST →	5-7	0.28
	1-5	0.32
	4-7	0.37
	0-4	0.38
	6-2	0.40
	3-6	0.52
	6-0	0.58

MST edges

0-7 1-7 0-2 2-3

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



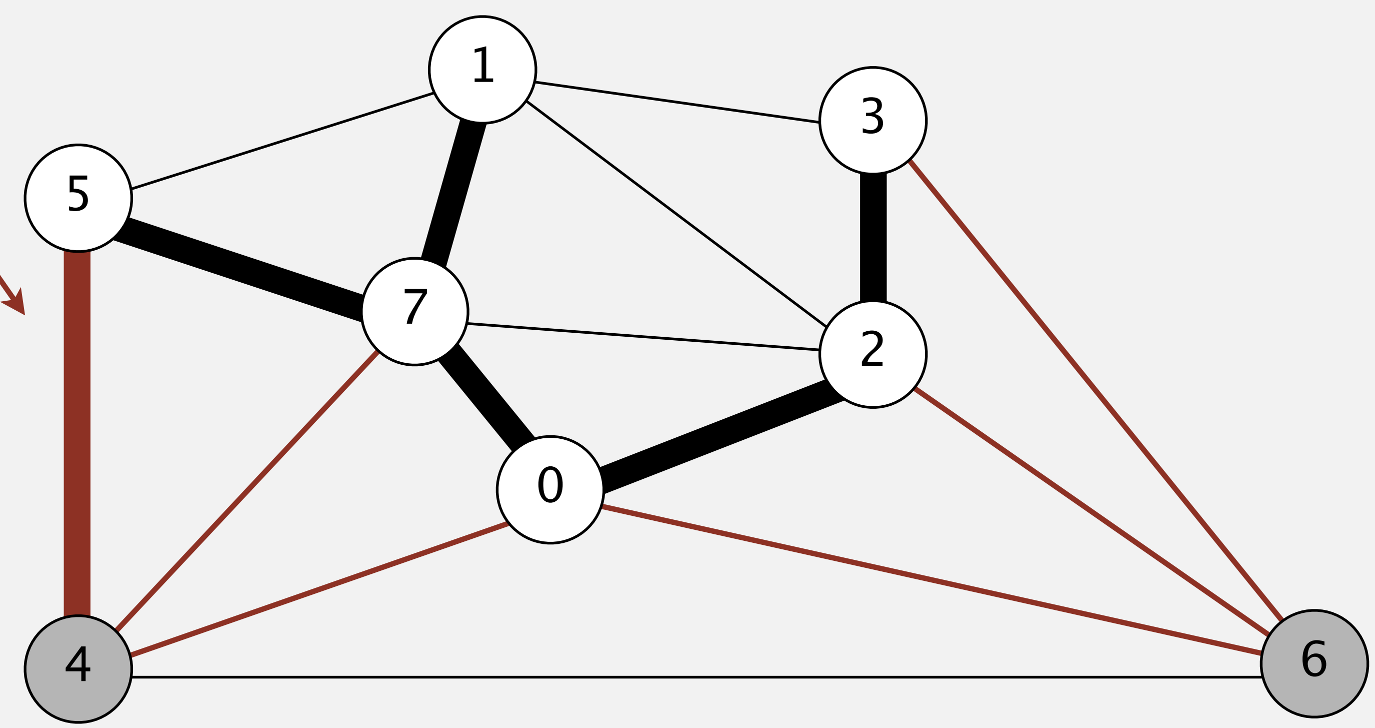
MST edges

0–7 1–7 0–2 2–3 5–7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

min weight edge with exactly one endpoint in T



edges with exactly one endpoint in T (sorted by weight)

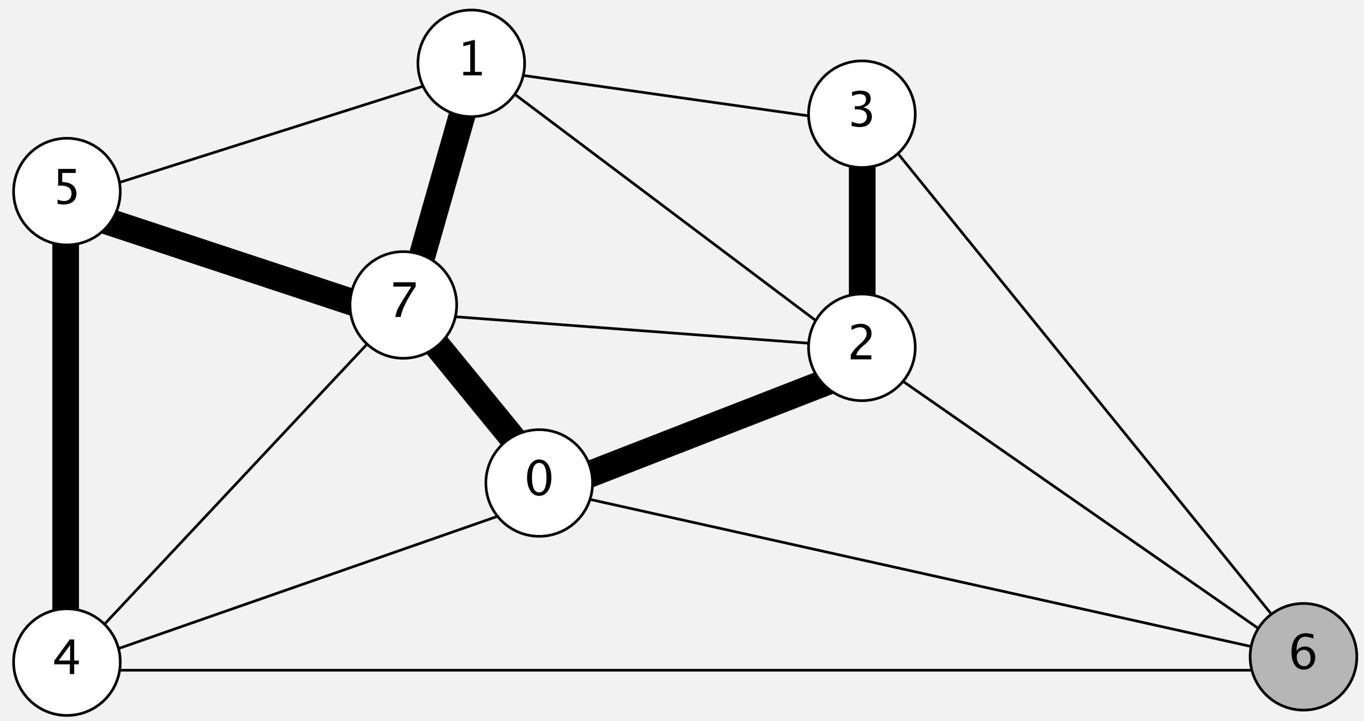
in MST → 4-5	0.35
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

MST edges

0-7 1-7 0-2 2-3 5-7

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.

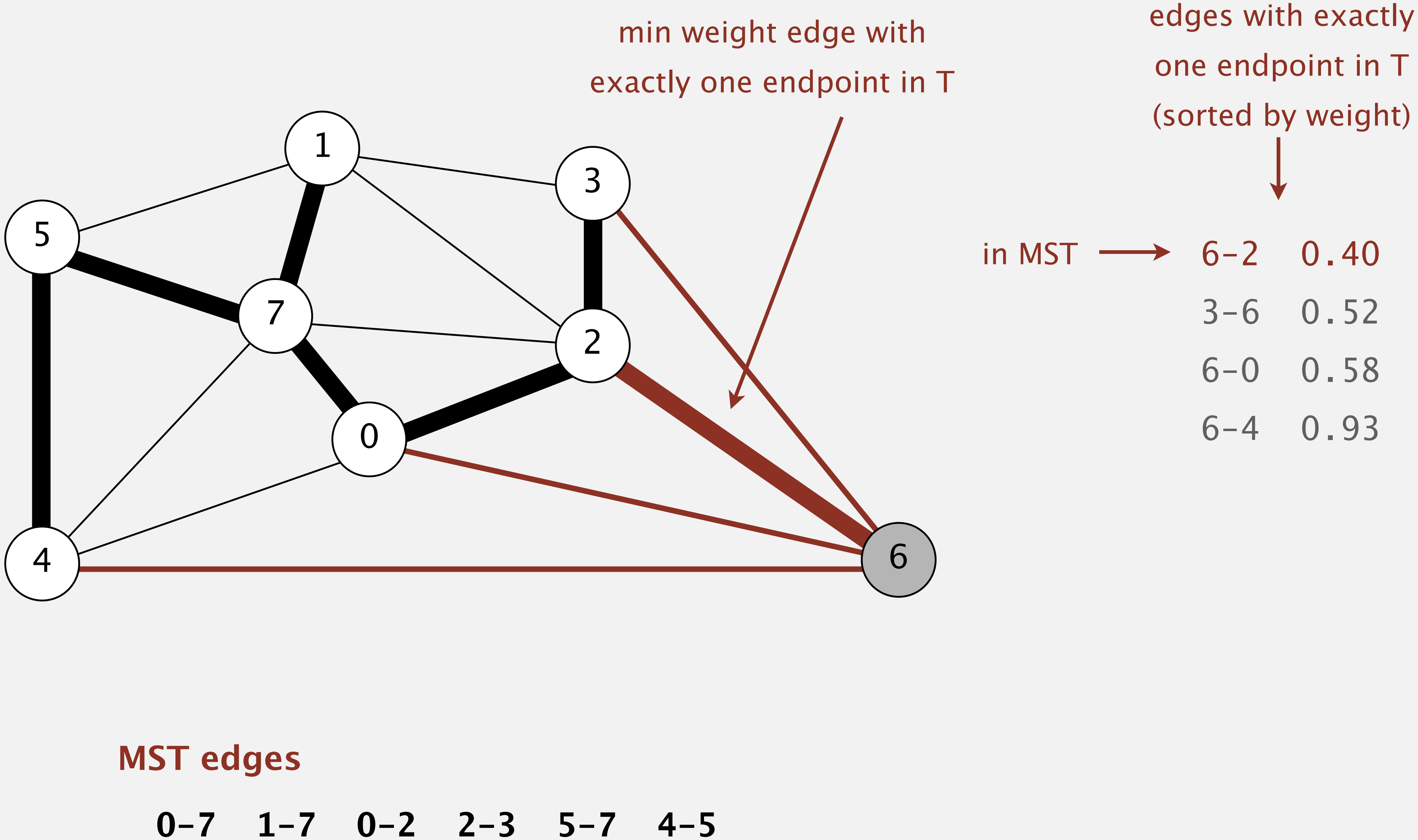


MST edges

0–7 1–7 0–2 2–3 5–7 4–5

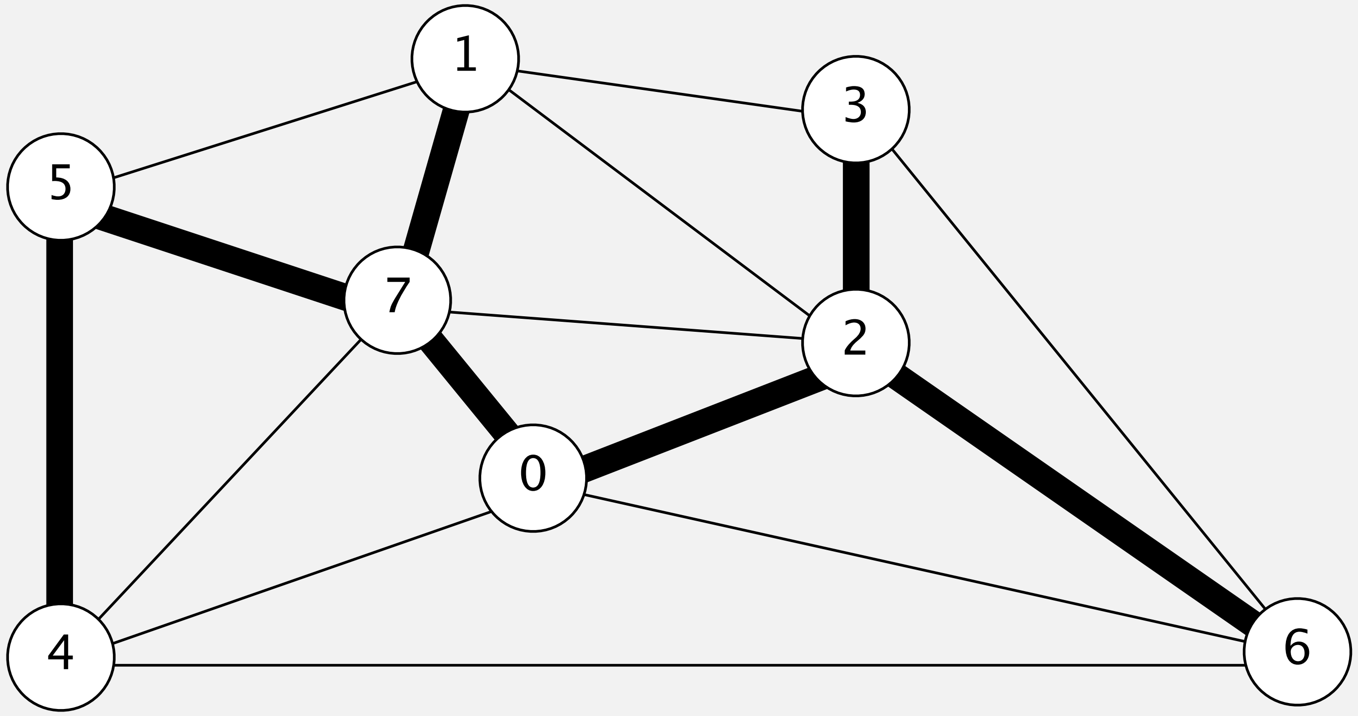
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



Prim's algorithm demo

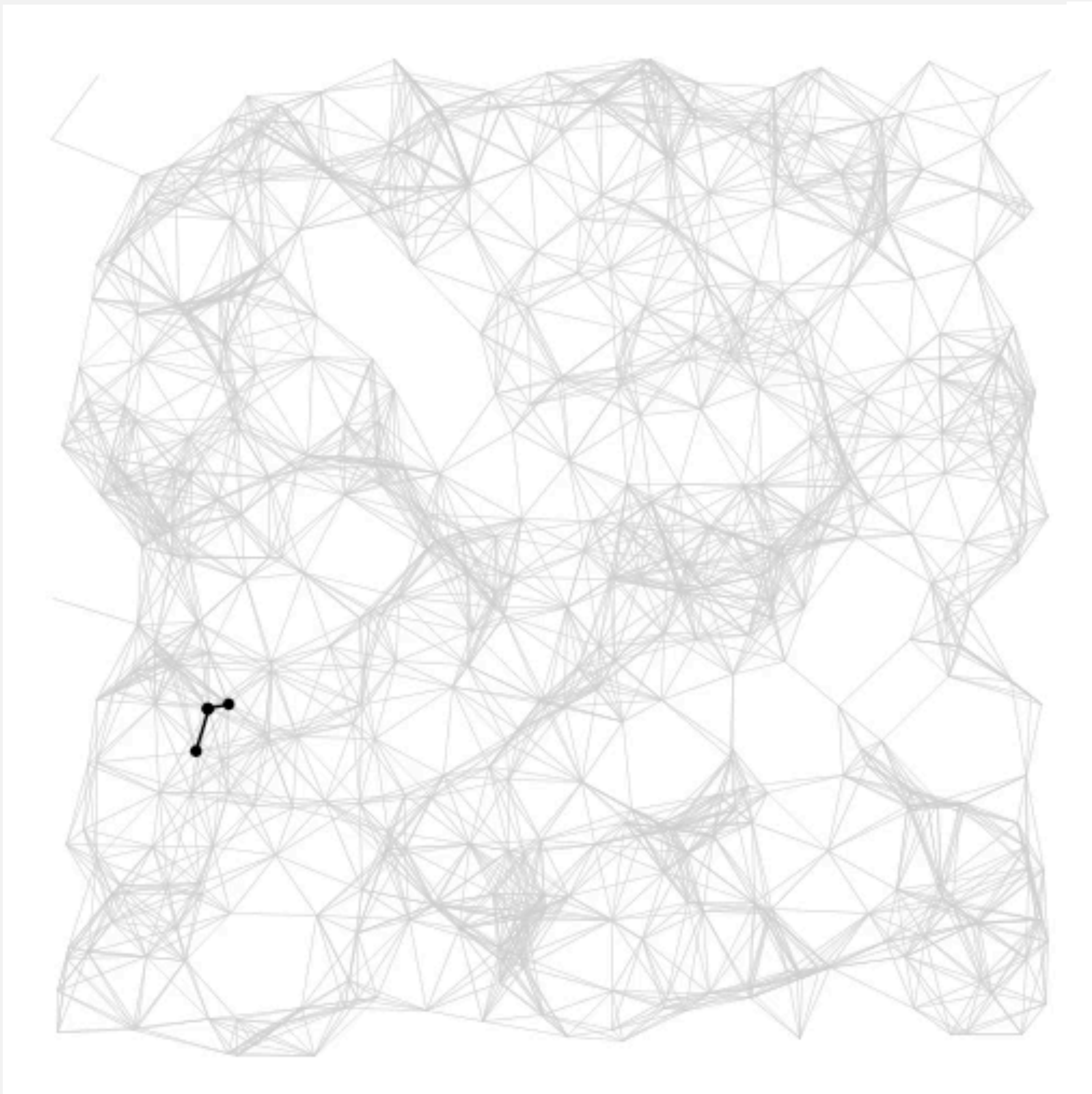
- Start with vertex 0 and greedily grow tree *T*.
- Add to *T* the min weight edge with exactly one endpoint in *T*.
- Repeat until *V* - 1 edges.



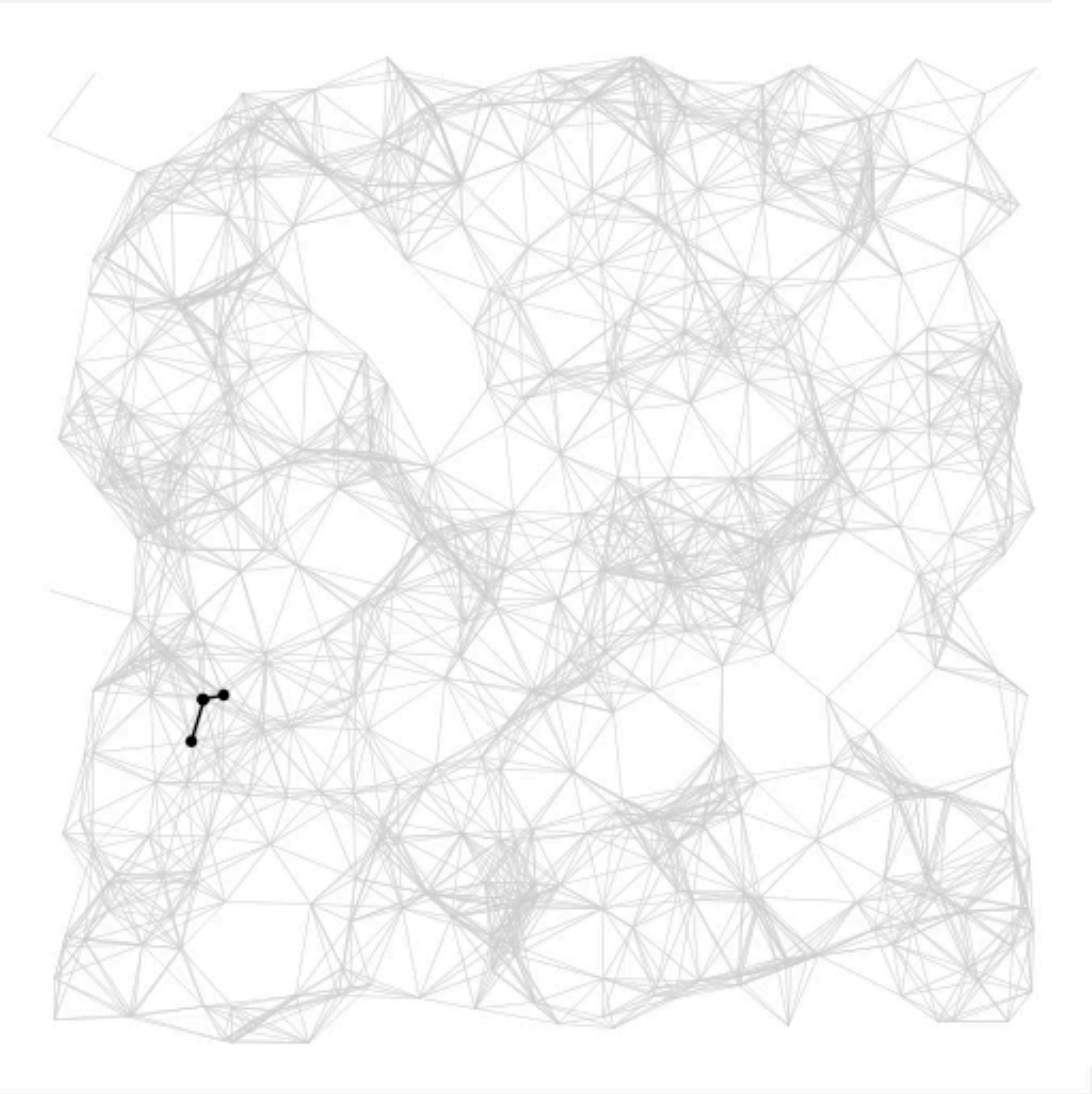
MST edges

0–7 1–7 0–2 2–3 5–7 4–5 6–2

Prim's algorithm: visualization



Prim's algorithm: visualization

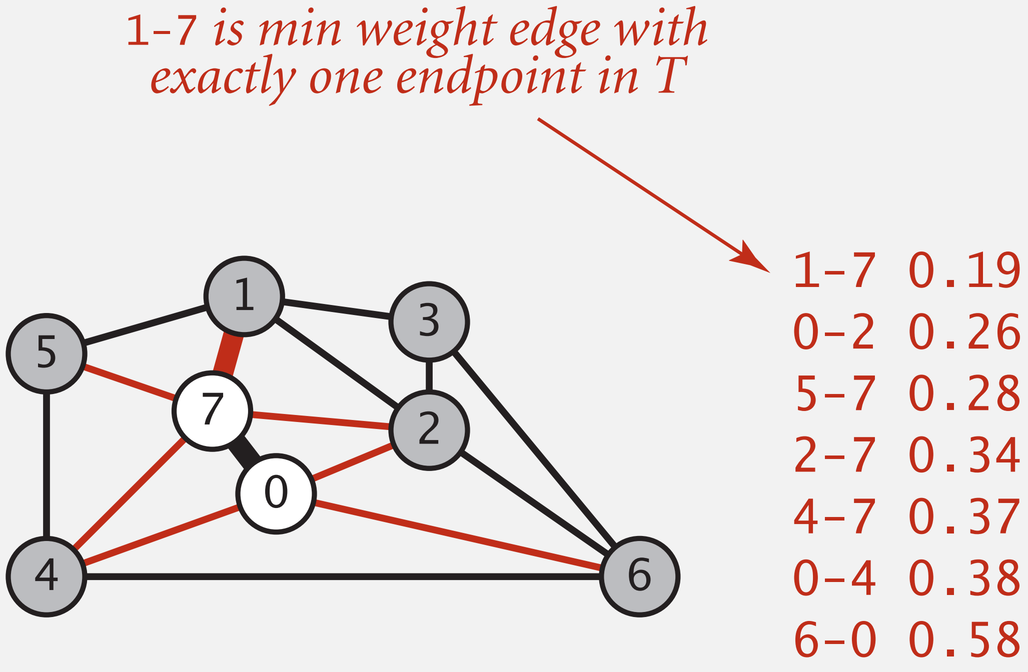


Prim's algorithm: implementation challenge

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

How difficult?

- E
- V
- $\log E$
- $\log^* E$
- 1

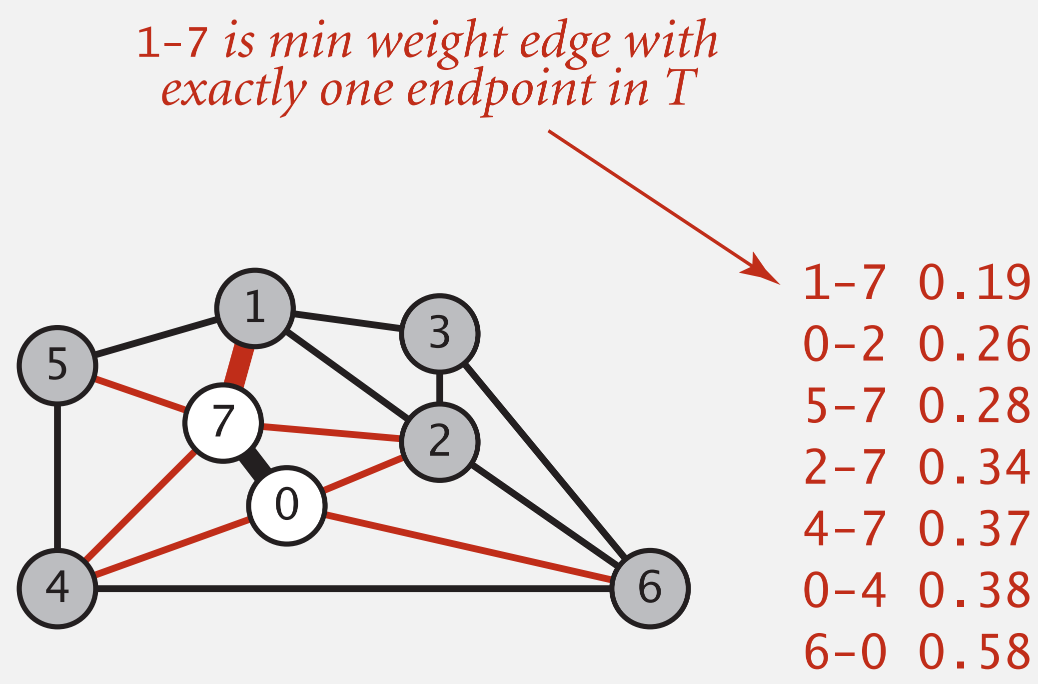


Prim's algorithm: implementation challenge

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

How difficult?

- E ← try all edges
- V
- $\log E$
- $\log^* E$
- 1

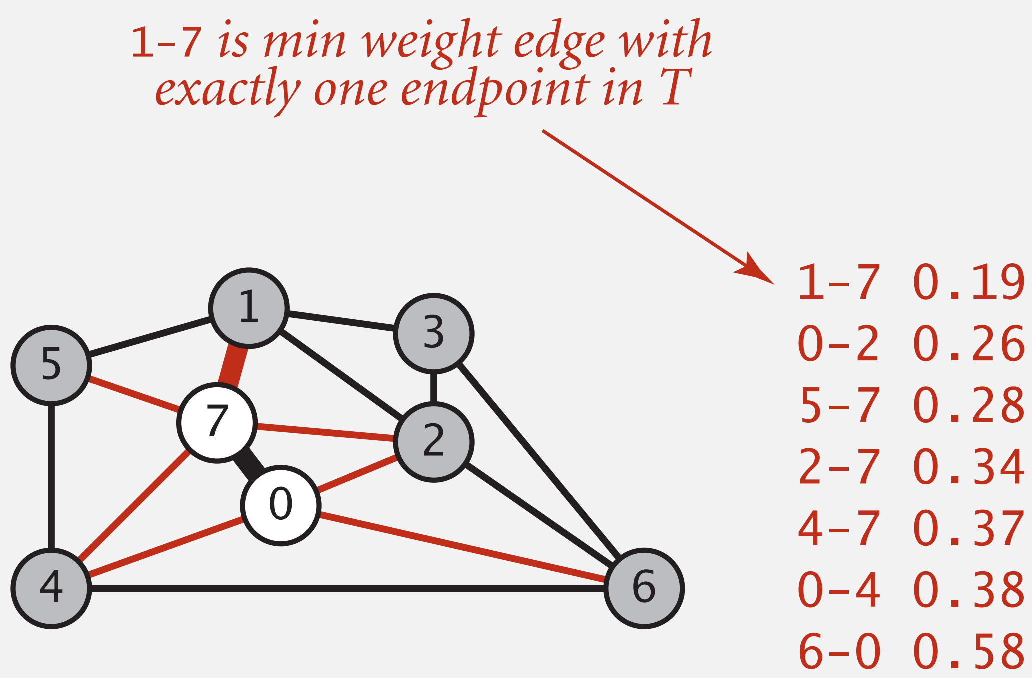


Prim's algorithm: implementation challenge

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

How difficult?

- E ← try all edges
- V
- $\log E$ ← use a priority queue!
- $\log^* E$
- 1



Prim's algorithm: lazy implementation

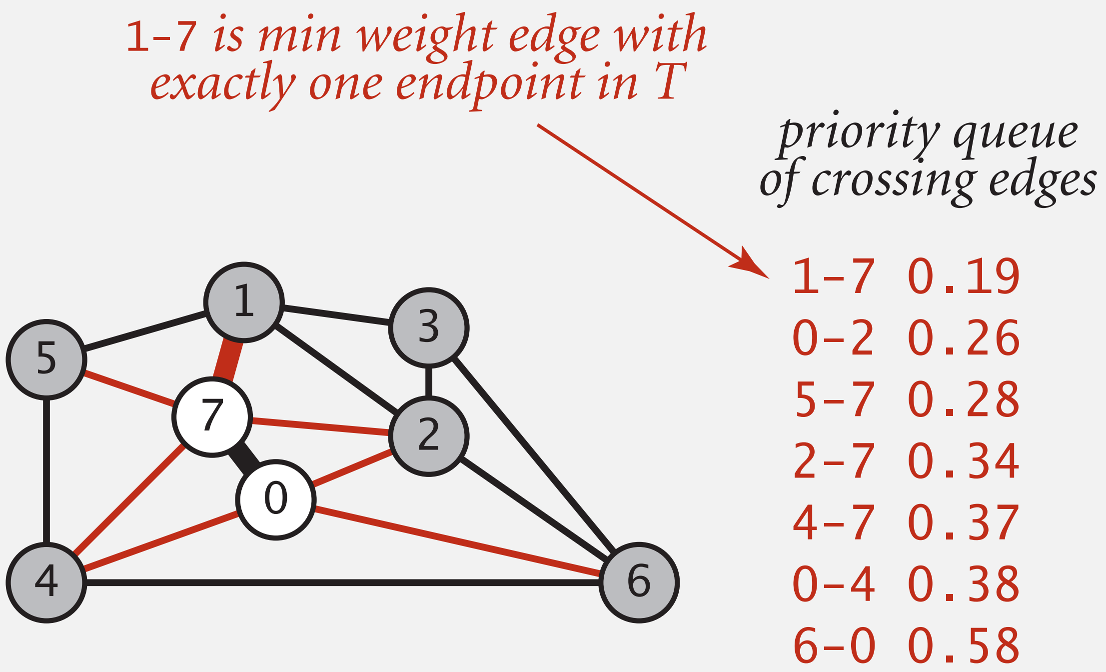
Challenge. Find the min weight edge with exactly one endpoint in *T*.

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.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v\text{–}w$ to add to T.

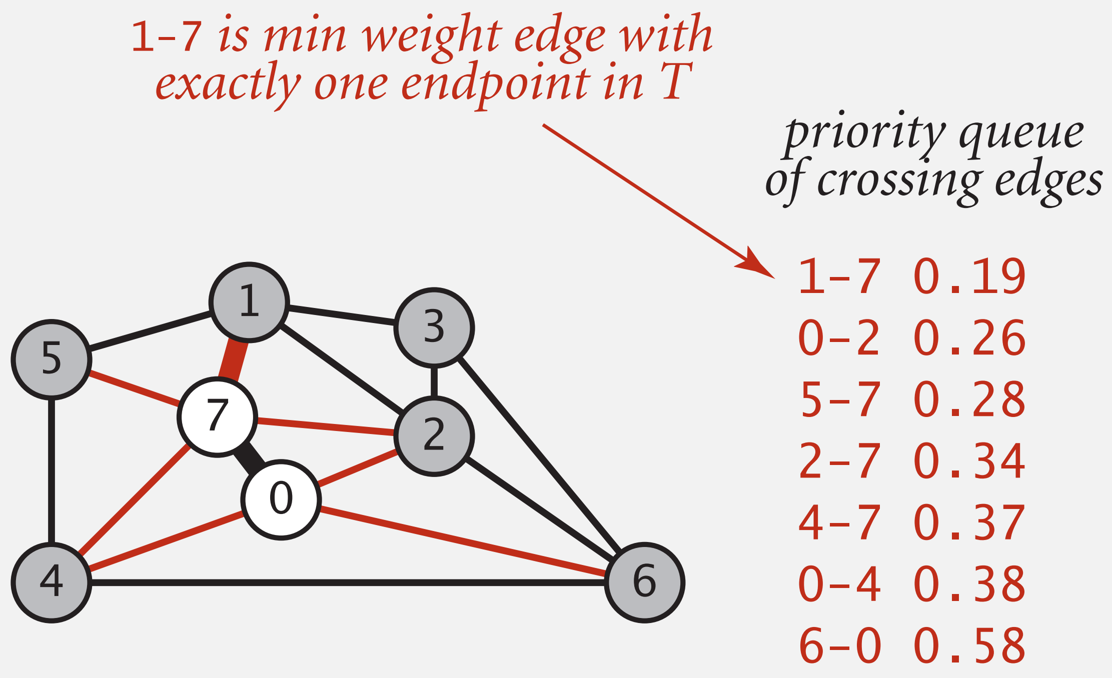


Prim's algorithm: lazy implementation

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*.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge *e* = *v*–*w* to add to *T*.
- Disregard if both endpoints *v* and *w* are marked (both in *T*).

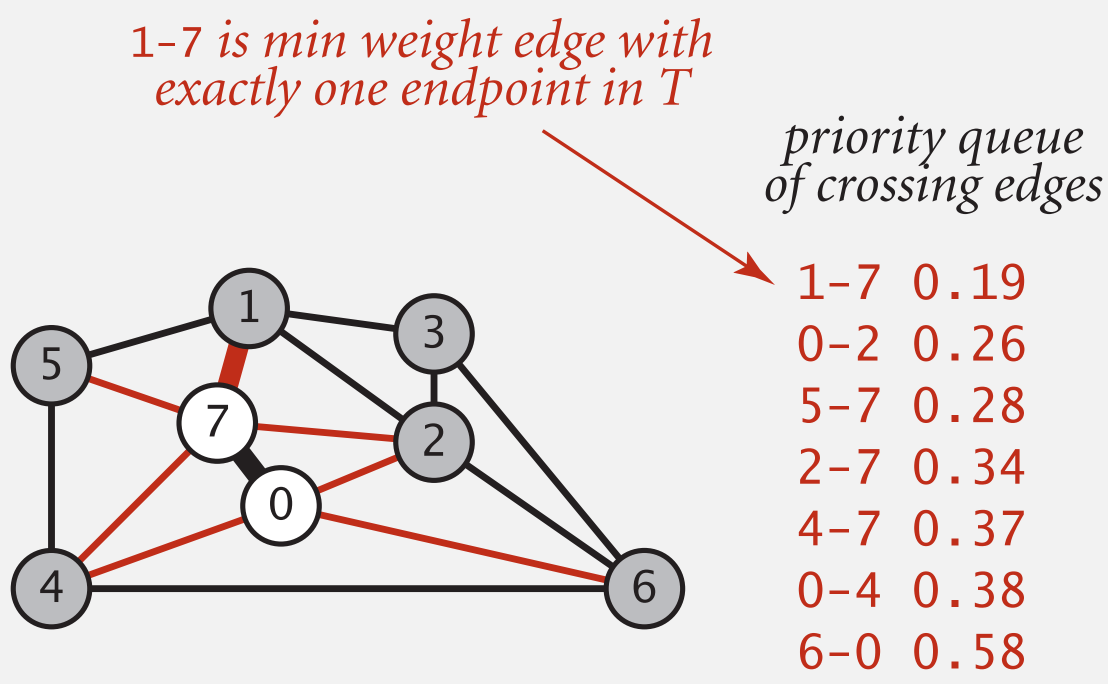


Prim's algorithm: lazy implementation

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*.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge *e* = *v*–*w* to add to *T*.
- Disregard if both endpoints *v* and *w* are marked (both in *T*).
- Otherwise, let *w* be the unmarked vertex (not in *T*):
  - add to PQ any edge incident to *w* (assuming other endpoint not in *T*)
  - add *e* to *T* and mark *w*



Lazy implementation of Prim's algorithm

Prim(graph G)

```
PQ = empty priority queue of edges
color all vertices grey
Visit(0)
while(|A| < n - 1)
    (u,v) = PQ.DeleteMin()
    if u or v is grey
        A = A ∪ (u, v)
    if u is grey
        Visit(u)
    else // v is grey
        Visit(v)
```

Visit(vertex u)

```
color u black
for all edges (u,v)
    if v is grey
        PQ.insert((u,v))
```

Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Lazy Prim's algorithm: running time

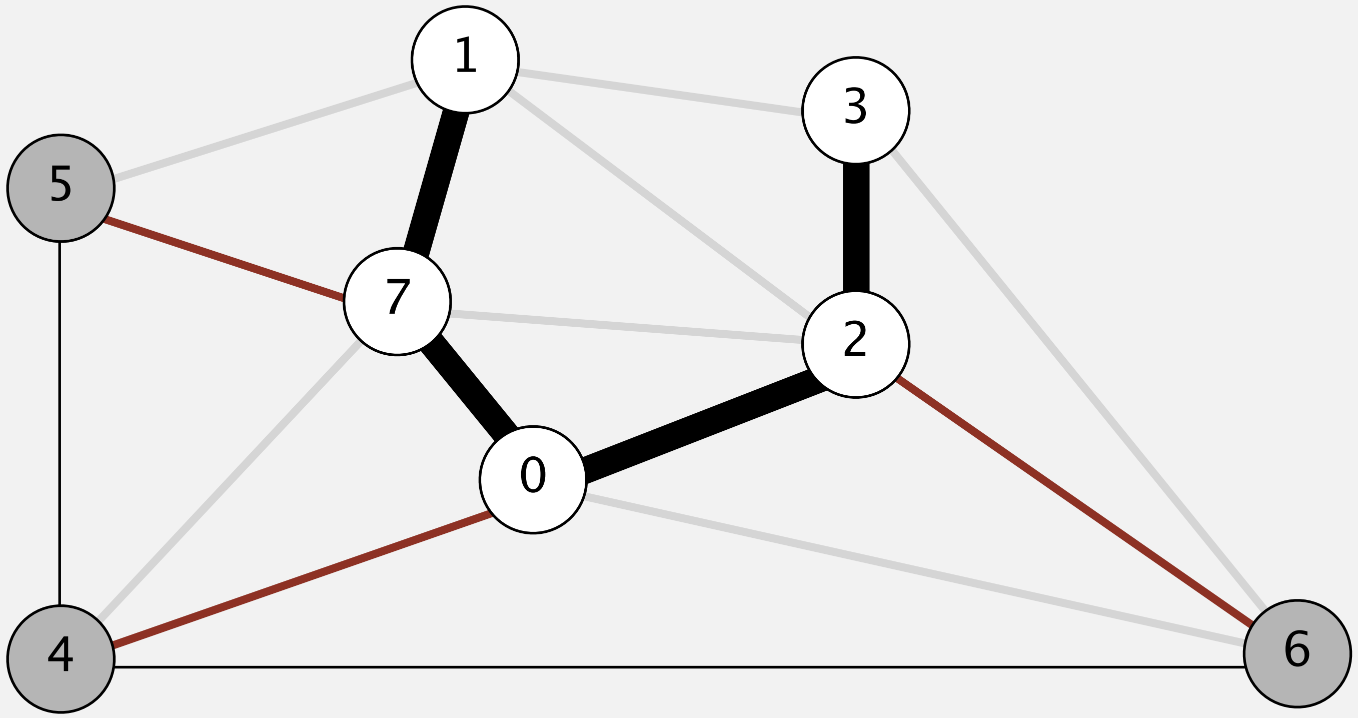
Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

operation	frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

Prim's algorithm: eager implementation

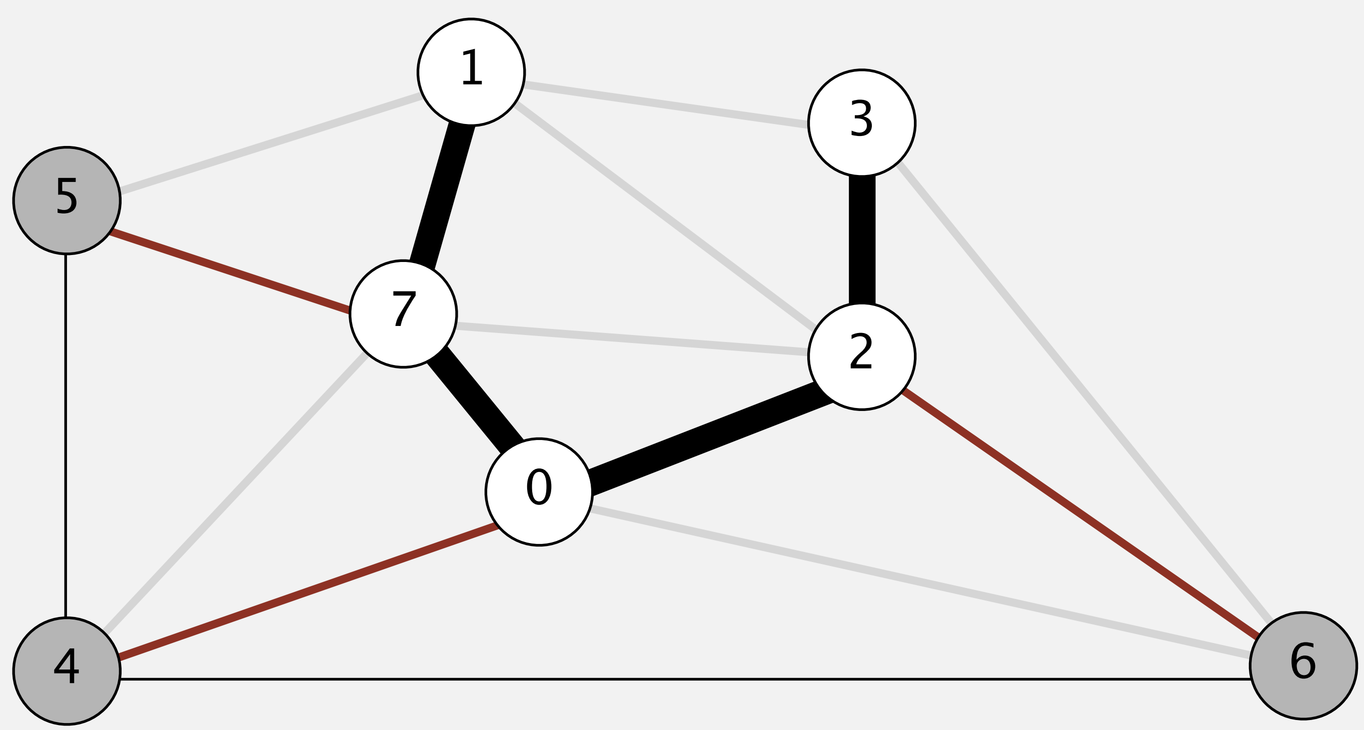
Challenge. Find min weight edge with exactly one endpoint in T.



Prim's algorithm: eager implementation

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

Observation. For each vertex v, need only min weight edge connecting v to T.

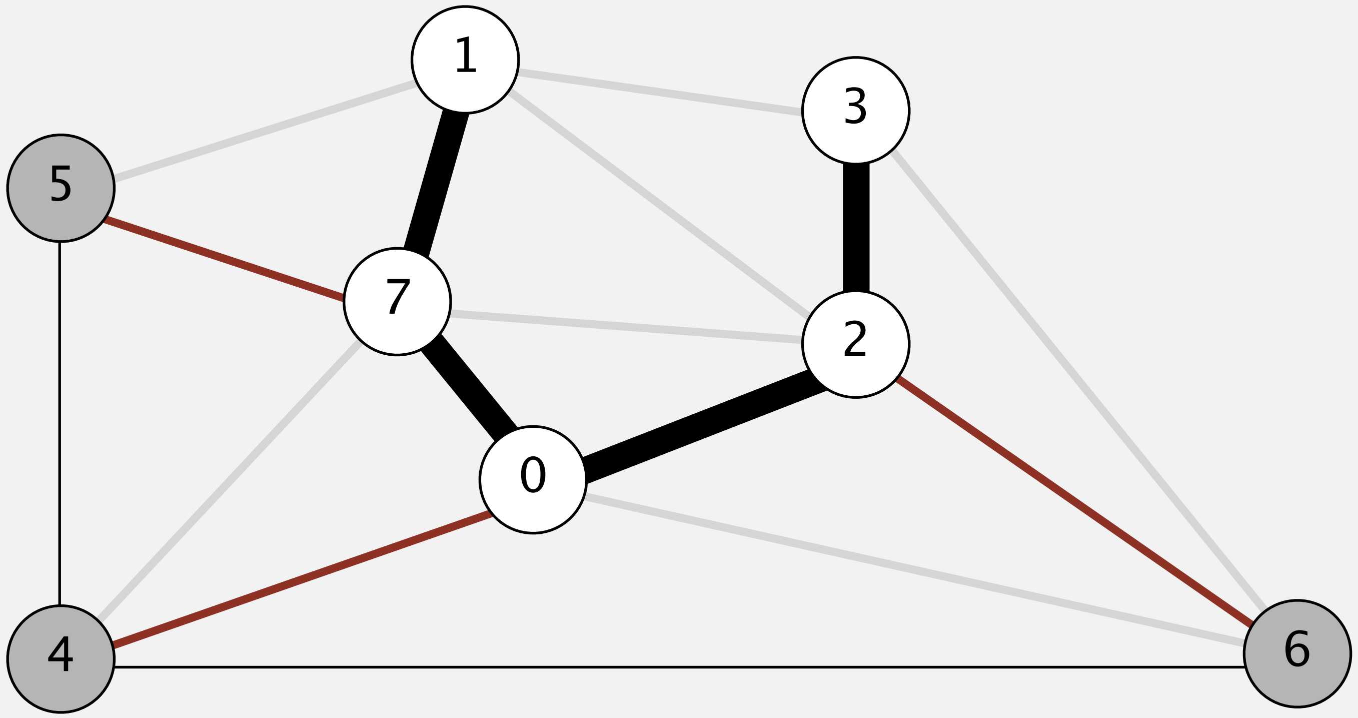


Prim's algorithm: eager implementation

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

Observation. For each vertex v, need only min weight edge connecting v to T.

- MST includes at most one edge connecting v to T. Why?

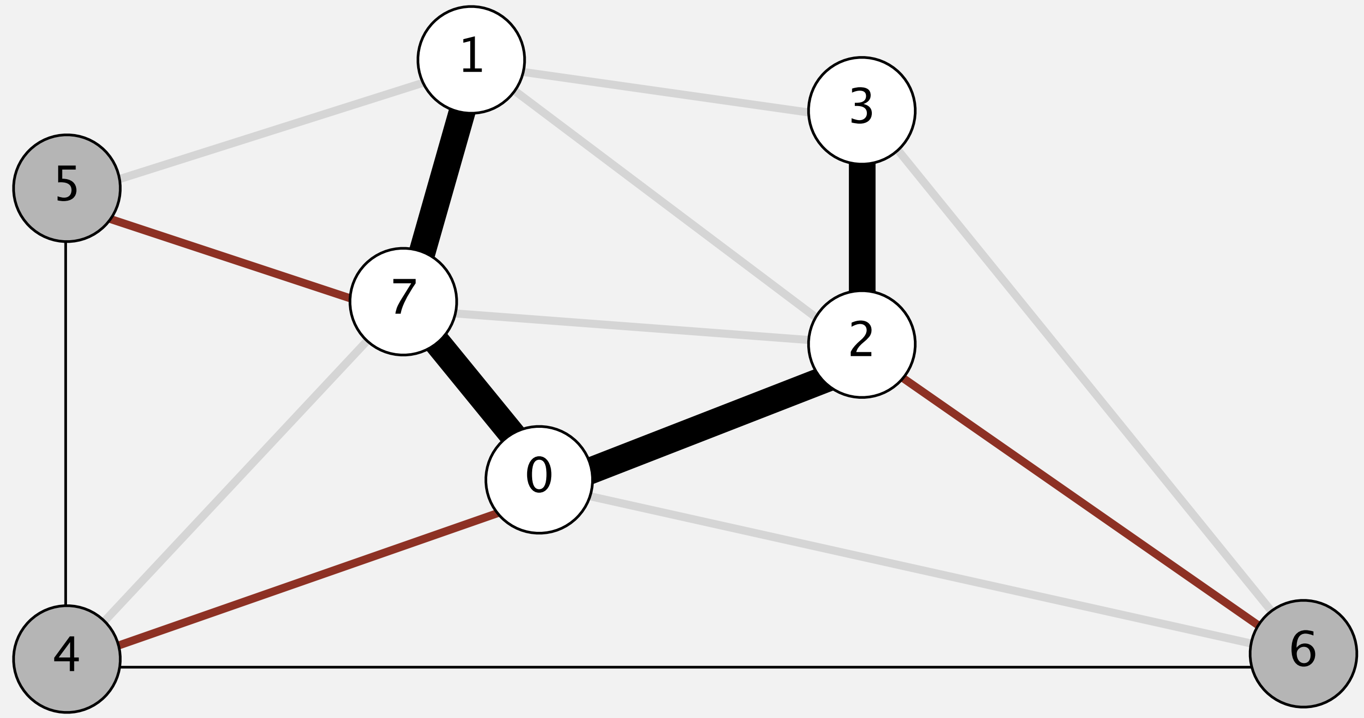


Prim's algorithm: eager implementation

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

Observation. For each vertex v, need only min weight edge connecting v to T.

- MST includes at most one edge connecting v to T. Why?
- If MST includes such an edge, it can take cheapest such edge. Why?



Prim's algorithm: eager implementation

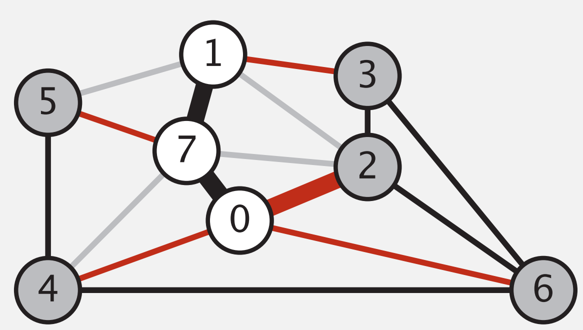
Challenge. Find min weight edge with exactly one endpoint in *T*.

Prim's algorithm: eager implementation

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

pq has at most one entry per vertex

Eager solution. Maintain a PQ of vertices connected by an edge to *T*, where priority of vertex *v* = weight of min weight edge connecting *v* to *T*.



0		
1	1-7	0.19
2	0-2	0.26
3	1-3	0.29
4	0-4	0.38
5	5-7	0.28
6	6-0	0.58
7	0-7	0.16

red: on PQ

black: on MST

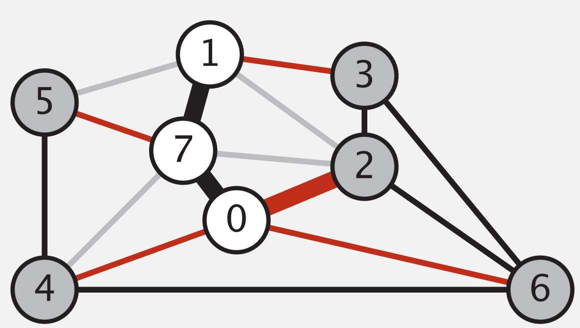
Prim's algorithm: eager implementation

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

pq has at most one entry per vertex

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of min weight edge connecting v to T.

- Delete min vertex v and add its associated edge $e = v–w$ to T.




```
0
1 1-7 0.19
2 0-2 0.26
3 1-3 0.29
4 0-4 0.38
5 5-7 0.28
6 6-0 0.58
7 0-7 0.16
```

red: on PQ

black: on MST

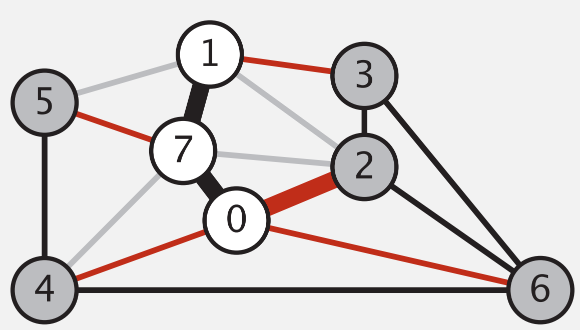
Prim's algorithm: eager implementation

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

pq has at most one entry per vertex

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of min weight edge connecting v to T.

- Delete min vertex v and add its associated edge $e = v–w$ to T.
- Update PQ by considering all edges $e = v–x$ incident to v
  - ignore if x is already in T
  - add x to PQ if not already on it
  - decrease priority of x if $v–x$ becomes min weight edge connecting x to T



0		
1	1-7	0.19
2	0-2	0.26
3	1-3	0.29
4	0-4	0.38
5	5-7	0.28
6	6-0	0.58
7	0-7	0.16

red: on PQ

black: on MST

Eager implementation of Prim's algorithm

```
Prim(graph G)
    PQ = empty priority queue of vertices
    cost = array of size n
    edge = array of size n
    color all vertices grey
    Visit(0)
    while(PQ not empty)
        u = PQ.DeleteMin()
        A = A ∪ edge[u]
        Visit(u)
```

```
Visit(vertex u)
    color u black
    for all edges (u,v)
        if v is grey
            color v red
            PQ.insert(v, w(u,v))
            cost[v] = w(u,v)
            edge[v] = (u,v)
        elseif (v is red) and (w(u,v) < cost[v])
            PQ.DecreaseKey(v, w(u,v))
            cost[v] = w(u,v)
            edge[v] = (u,v)
```

Prim's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

Prim's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

Prim's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2

Bottom line.

- Array implementation optimal for dense graphs.

Prim's algorithm: which priority queue?

Depends on PQ implementation: *V* insert, *V* delete-min, *E* decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.

Prim's algorithm: which priority queue?

Depends on PQ implementation: *V* insert, *V* delete-min, *E* decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.

Prim's algorithm: which priority queue?

Depends on PQ implementation: *V* insert, *V* delete-min, *E* decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

Bottom line.

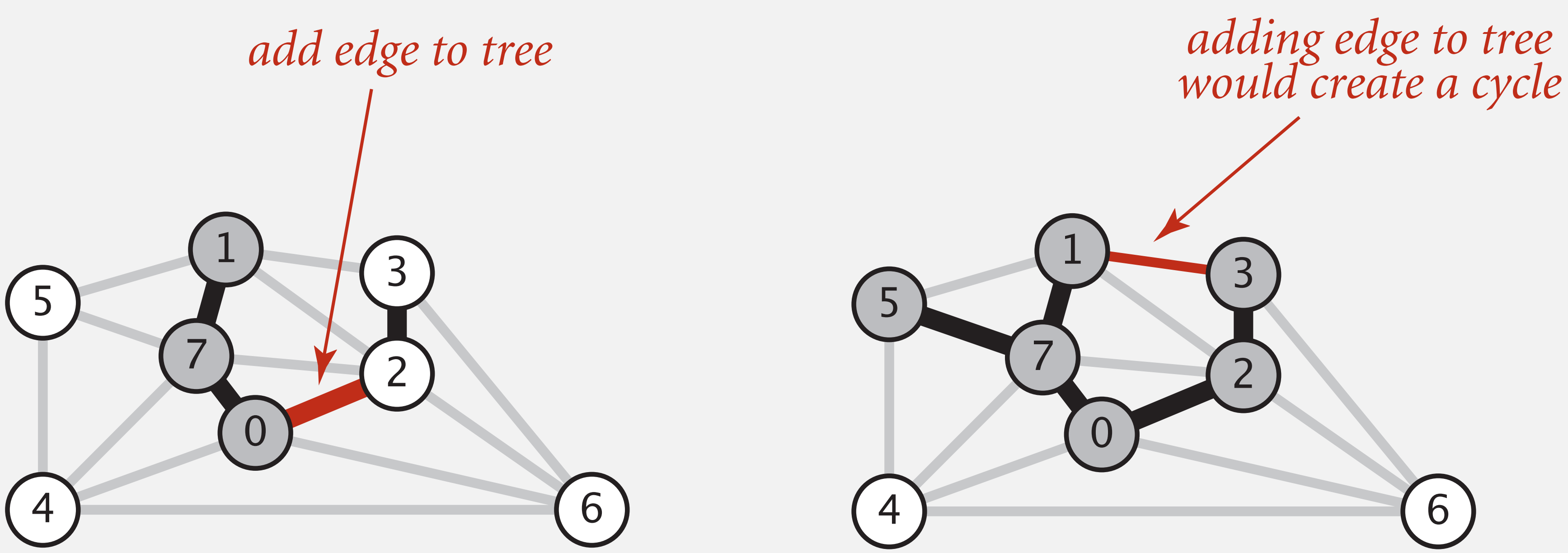
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v–w$ to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V
- $\log V$
- $\log^* V$
- 1

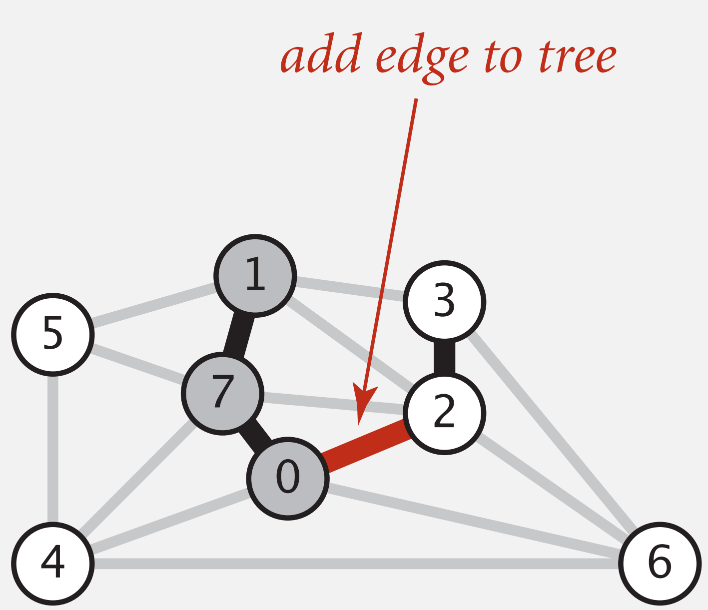


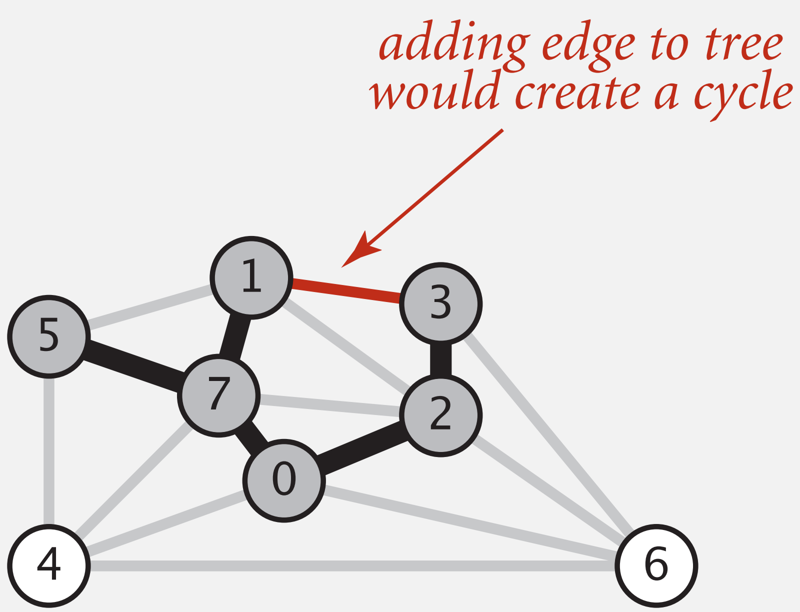
Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v\text{–}w$ to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v, check if w is reachable (T has at most V – 1 edges)
- $\log V$
- $\log^* V$
- 1



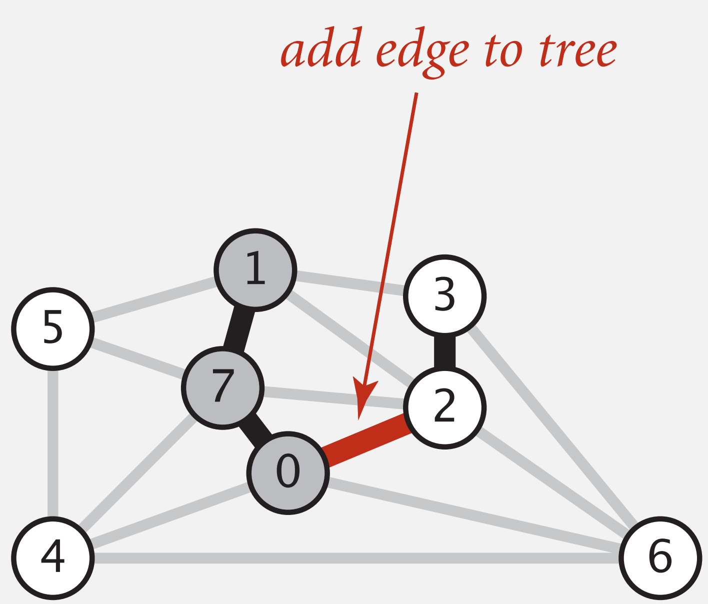


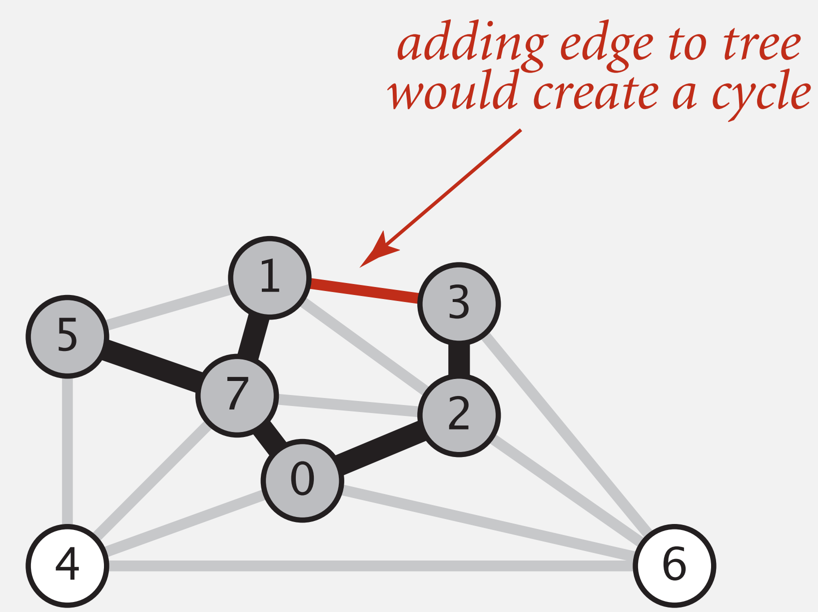
Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v–w$ to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v, check if w is reachable (T has at most V – 1 edges)
- $\log V$
- $\log^* V$ ← use the union–find data structure !
- 1





Kruskal's algorithm: implementation challenge

Challenge. Would adding edge *v*–*w* to tree *T* create a cycle? If not, add it.

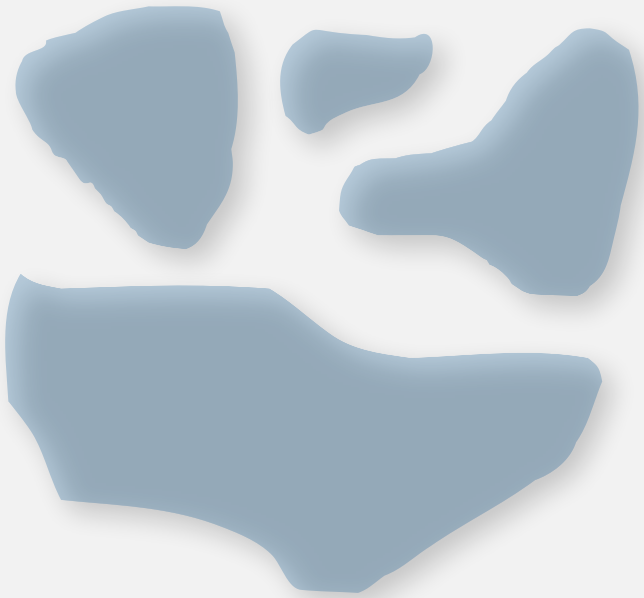
Efficient solution. Use the union–find data structure.

Kruskal's algorithm: implementation challenge

Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

Efficient solution. Use the union–find data structure.

- Maintain a set for each connected component in T.

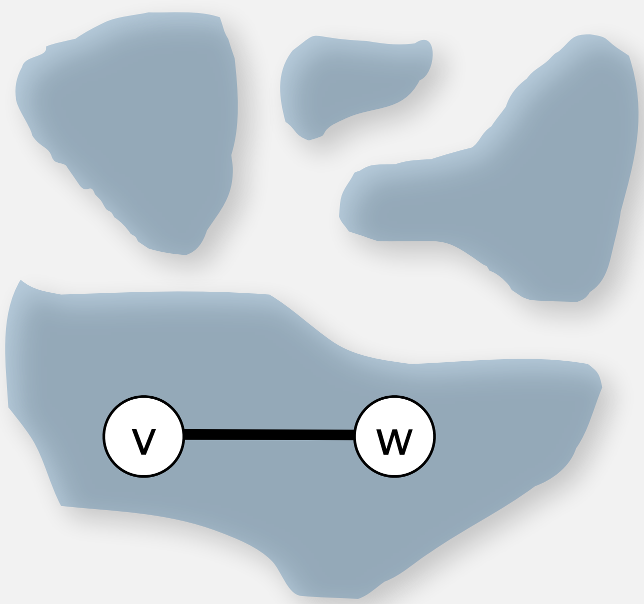


Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v–w$ to tree T create a cycle? If not, add it.

Efficient solution. Use the union-find data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding $v–w$ would create a cycle.



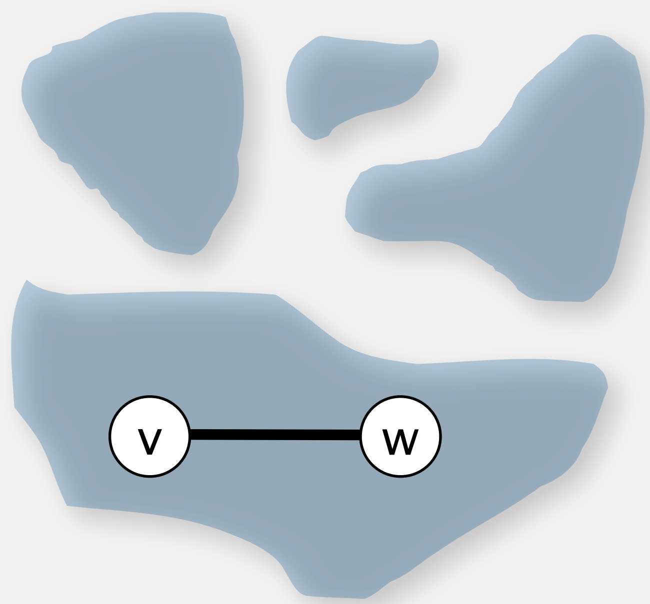
Case 1: adding v–w creates a cycle

Kruskal's algorithm: implementation challenge

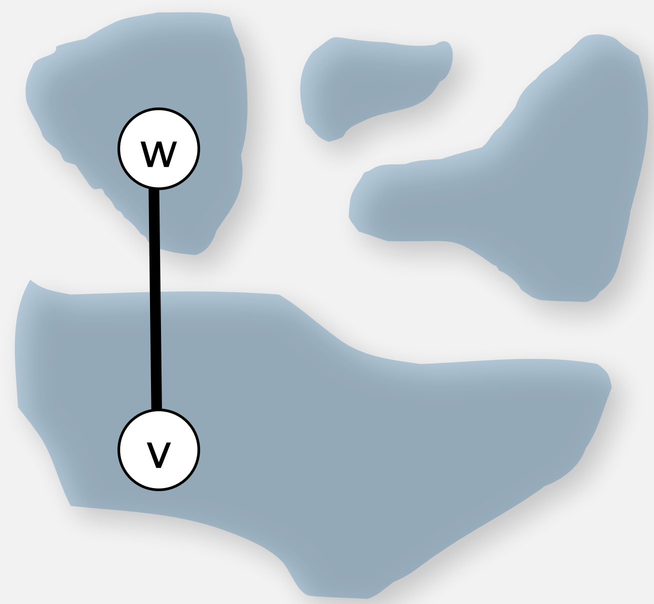
Challenge. Would adding edge *v*–*w* to tree *T* create a cycle? If not, add it.

Efficient solution. Use the union–find data structure.

- Maintain a set for each connected component in *T*.
- If *v* and *w* are in same set, then adding *v*–*w* would create a cycle.
- To add *v*–*w* to *T*, merge sets containing *v* and *w*.



Case 1: adding v–w creates a cycle



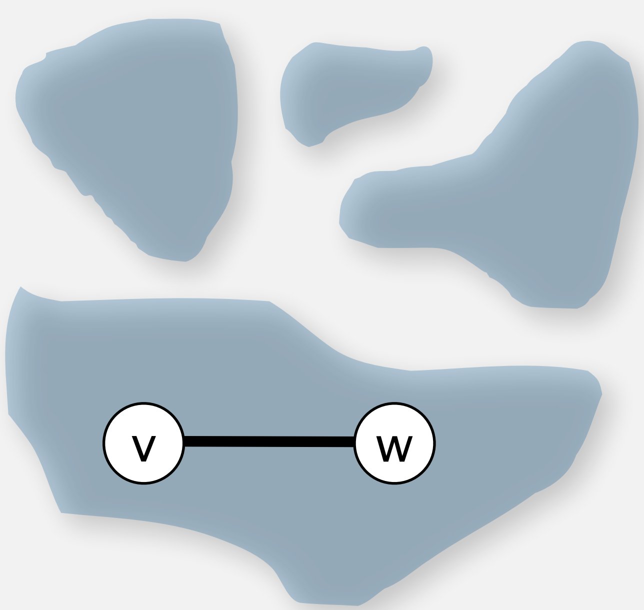
Case 2: add v–w to T and merge sets containing v and w

Kruskal's algorithm: implementation challenge

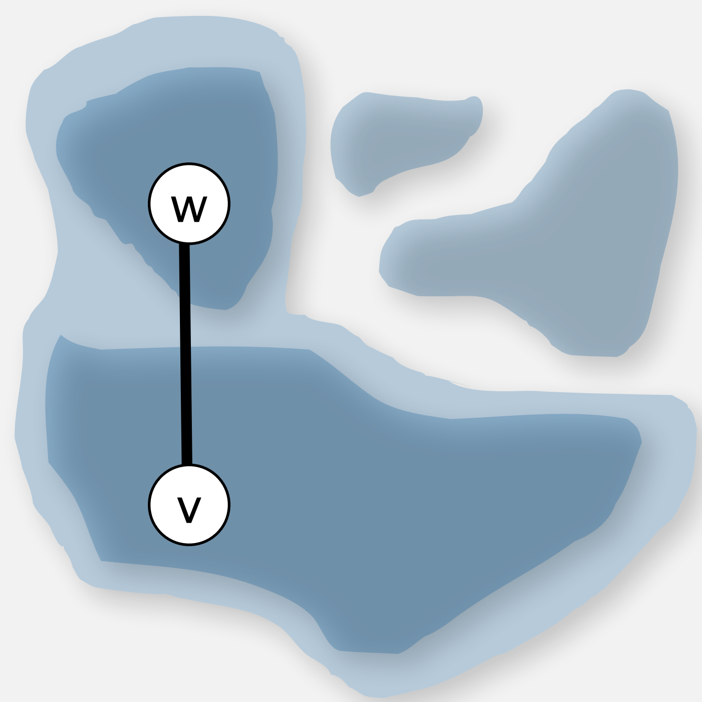
Challenge. Would adding edge $v–w$ to tree T create a cycle? If not, add it.

Efficient solution. Use the union–find data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding $v–w$ would create a cycle.
- To add $v–w$ to T, merge sets containing v and w.



Case 1: adding v–w creates a cycle



Case 2: add v–w to T and merge sets containing v and w

Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST(EdgeWeightedGraph G)
   {
      MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());

      UF uf = new UF(G.V());
      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();
         int v = e.either(), w = e.other(v);
         if (!uf.connected(v, w))
         {
            uf.union(v, w);
            mst.enqueue(e);
         }
      }
   }

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

build priority queue (or sort)

greedily add edges to MST

edge v–w does not create cycle

merge sets

add edge to MST

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

operation	frequency	time per op
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V$ †
connected	E	$\log^* V$ †

† amortized bound using weighted quick union with path compression

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

operation	frequency	time per op
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V$ †
connected	E	$\log^* V$ †

† amortized bound using weighted quick union with path compression

recall: $\log^* V \le 5$ in this universe

Remark. If edges are already sorted, order of growth is $E \log^* V$.

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V, \; E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle
2002	*optimal*	Pettie–Ramachandran

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle
2002	*optimal*	Pettie–Ramachandran
20xx	E	???

PRINCETON UNIVERSITY

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

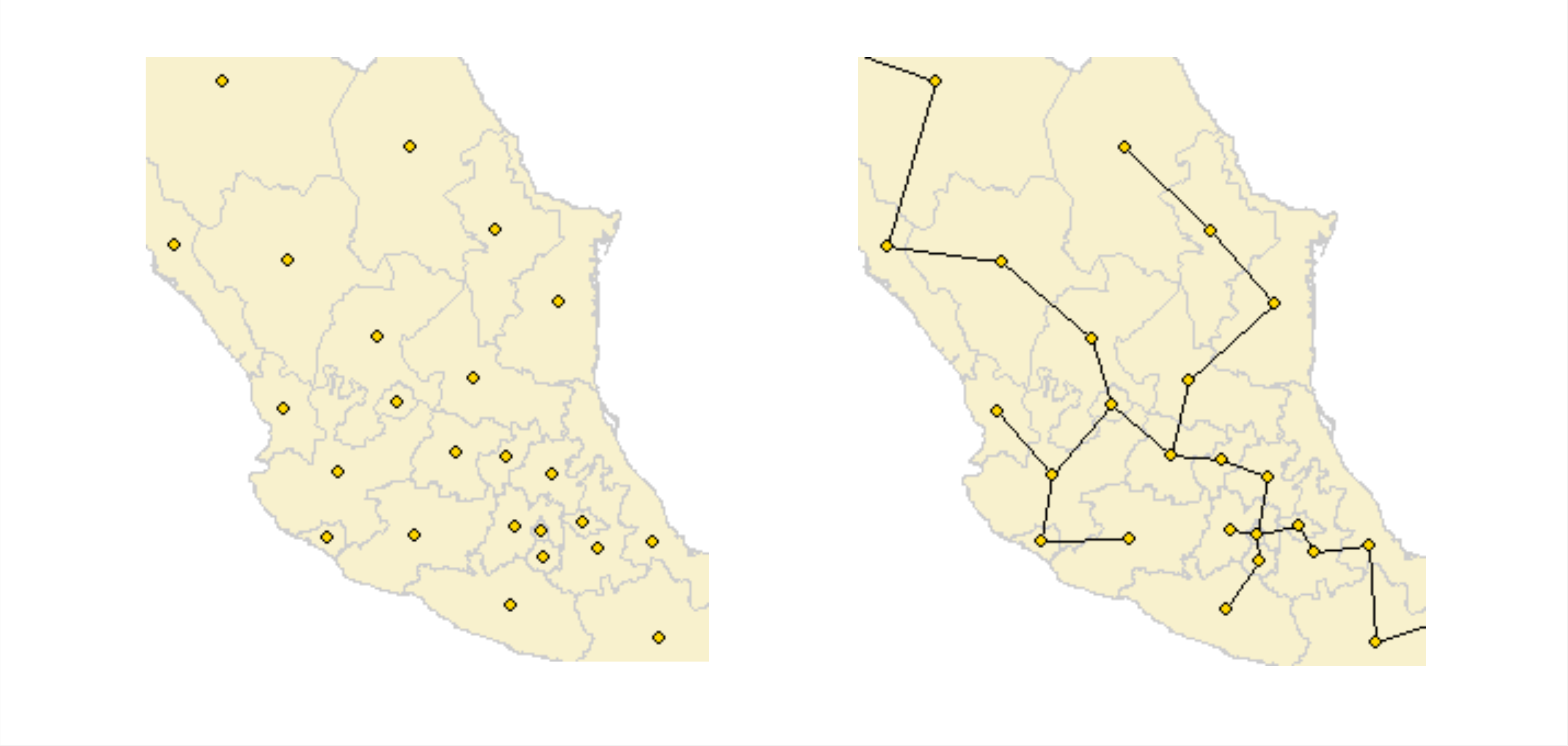
year	worst case	discovered by
1975	$E \log \log V$	Yao
1976	$E \log \log V$	Cheriton–Tarjan
1984	$E \log^* V,\ E + V \log V$	Fredman–Tarjan
1986	$E \log (\log^* V)$	Gabow–Galil–Spencer–Tarjan
1997	$E\, \alpha(V) \log \alpha(V)$	Chazelle
2000	$E\, \alpha(V)$	Chazelle
2002	*optimal*	Pettie–Ramachandran
20xx	E	???

PRINCETON UNIVERSITY

Remark. Linear-time randomized MST algorithm (Karger–Klein–Tarjan 1995).

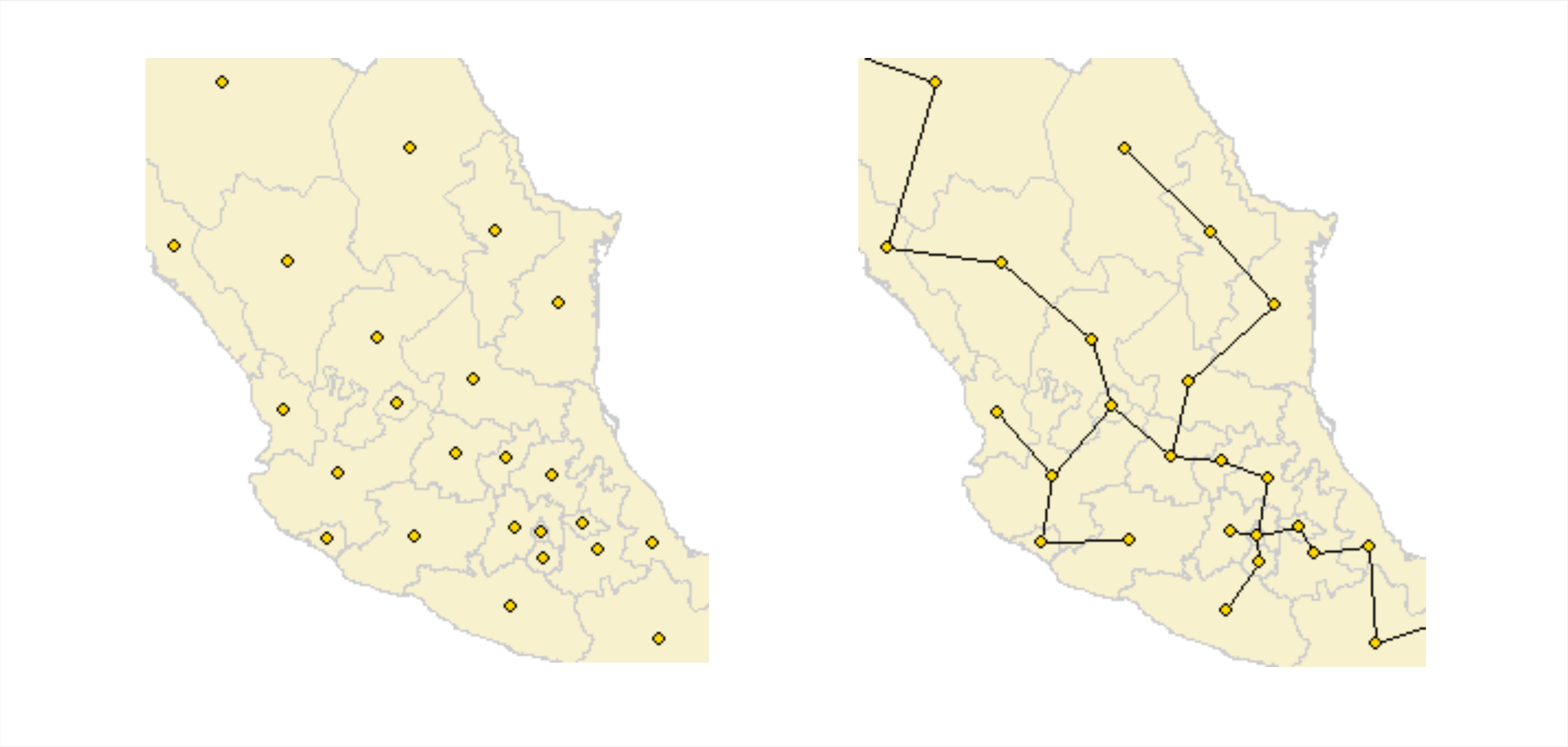
Euclidean MST

Given N points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.



Euclidean MST

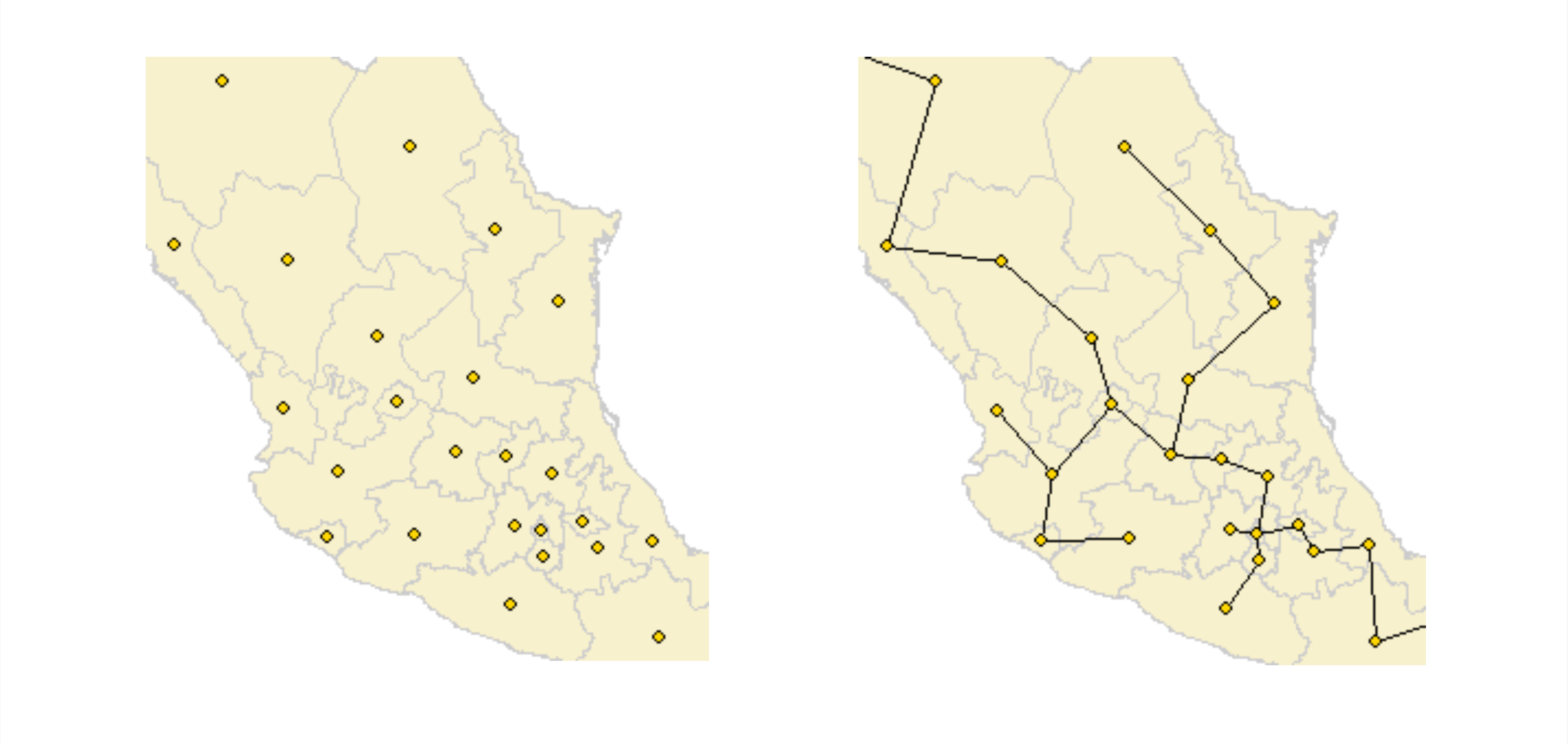
Given N points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.



Brute force. Compute $\sim N^2 / 2$ distances and run Prim's algorithm.

Euclidean MST

Given N points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.



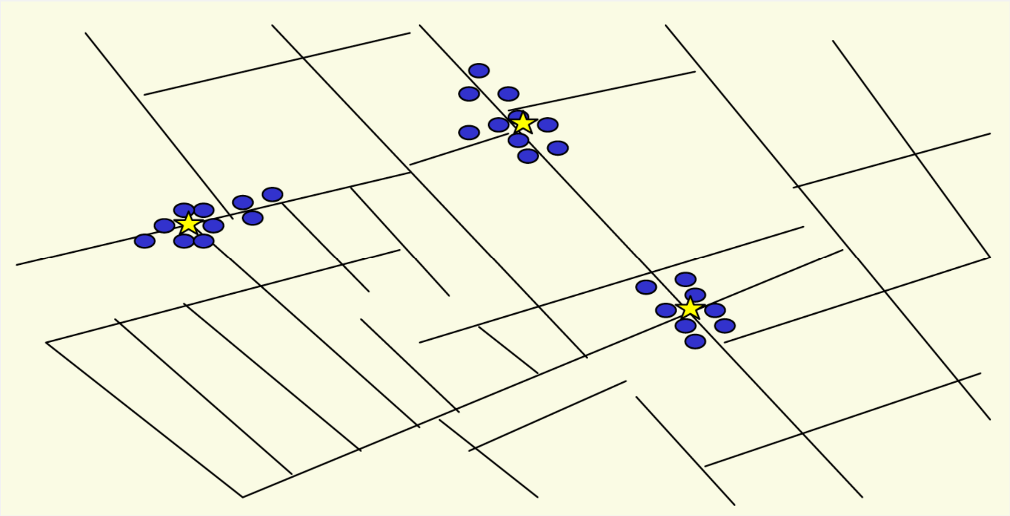
Brute force. Compute $\sim N^2 / 2$ distances and run Prim's algorithm.
Ingenuity. Exploit geometry and do it in $\sim c N \log N$.

Scientific application: clustering

k-clustering. Divide a set of objects classify into k coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Goal. Divide into clusters so that objects in different clusters are far apart.

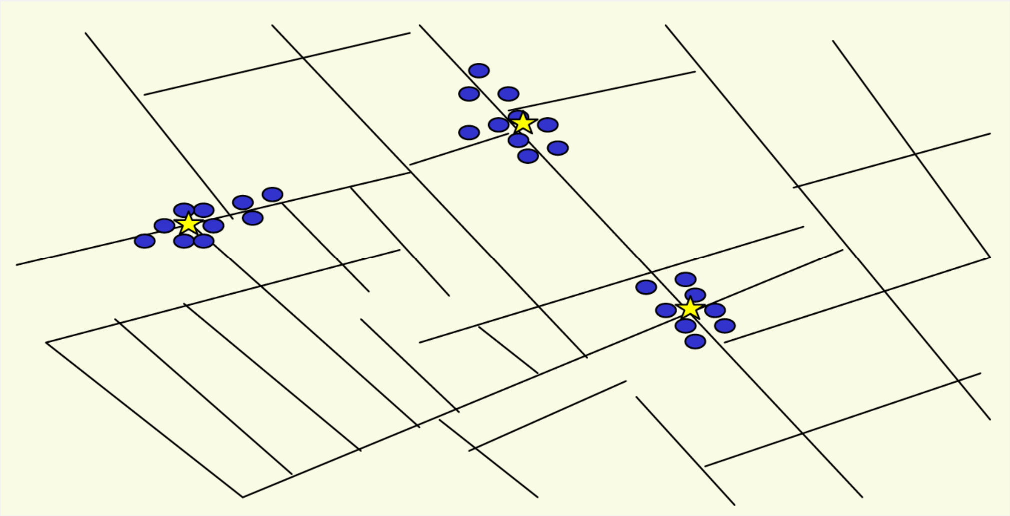


outbreak of cholera deaths in London in 1850s (Nina Mishra)

Scientific application: clustering

k-clustering. Divide a set of objects classify into k coherent groups.
Distance function. Numeric value specifying "closeness" of two objects.

Goal. Divide into clusters so that objects in different clusters are far apart.



outbreak of cholera deaths in London in 1850s (Nina Mishra)

Applications.

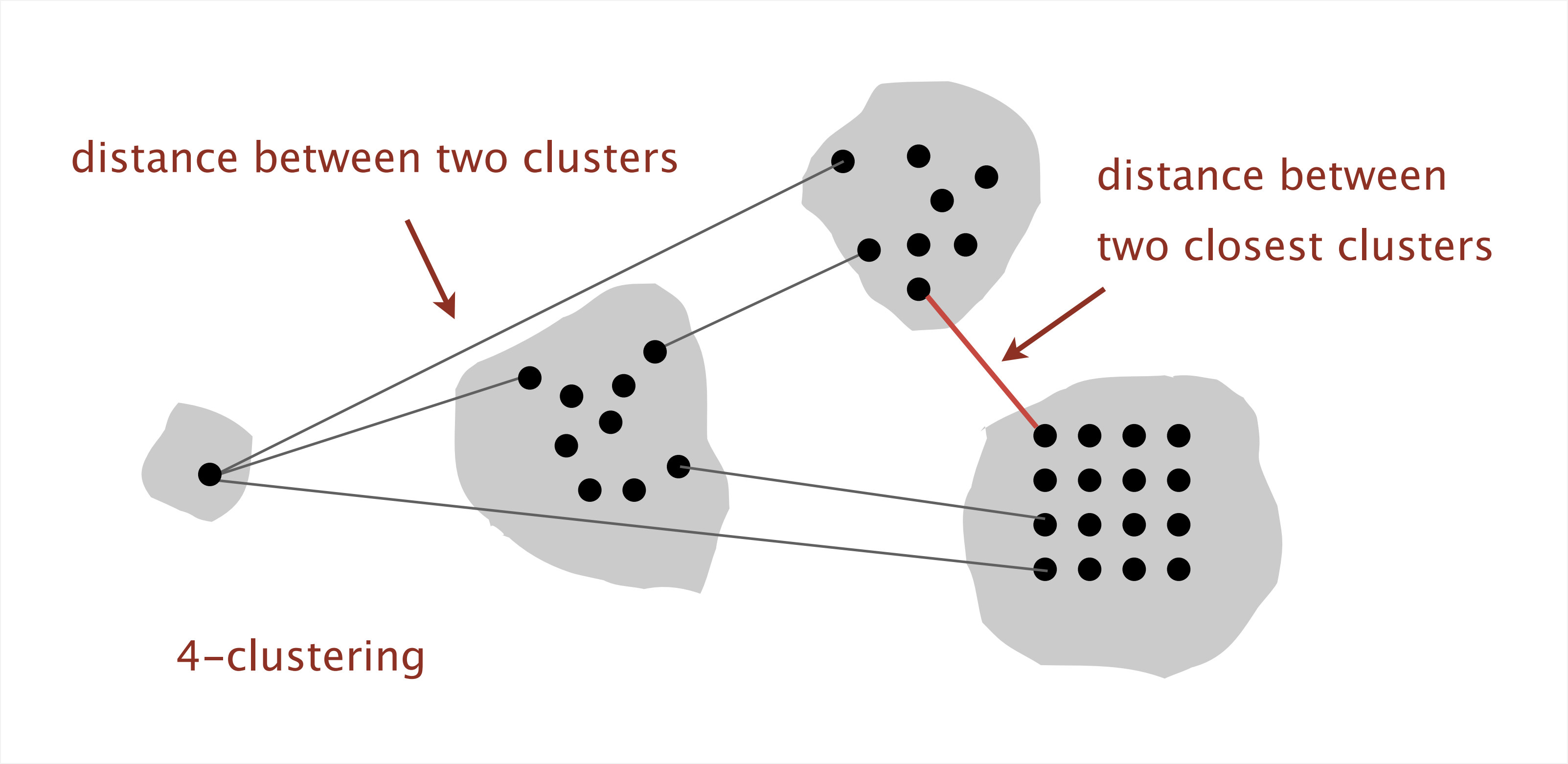
- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases.
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Single-link clustering

k-clustering. Divide a set of objects classify into k coherent groups.
Distance function. Numeric value specifying "closeness" of two objects.

Single link. Distance between two clusters equals the distance between the two closest objects (one in each cluster).

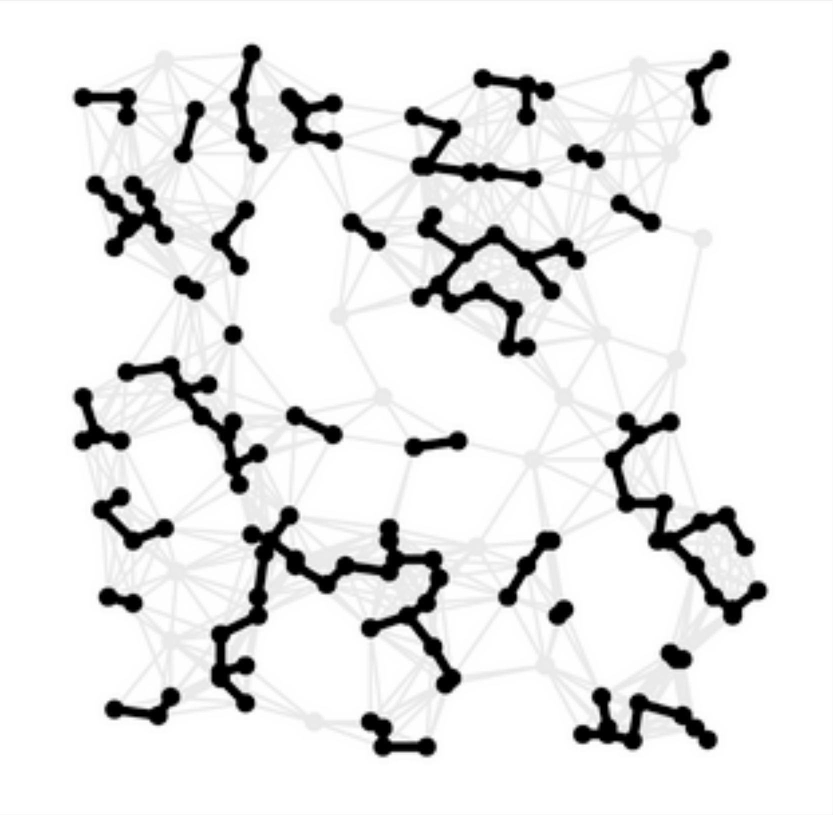
Single-link clustering. Given an integer k, find a k-clustering that maximizes the distance between two closest clusters.



Single-link clustering algorithm

"Well-known" algorithm in science literature for single-link clustering:

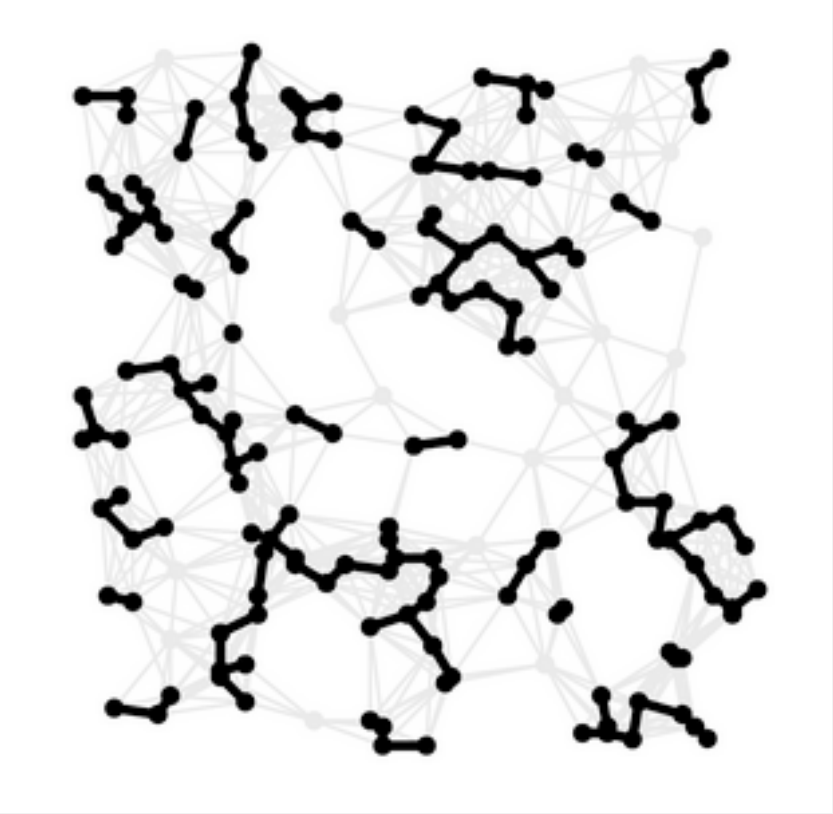
- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly k clusters.



Single-link clustering algorithm

"Well-known" algorithm in science literature for single-link clustering:

- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly k clusters.

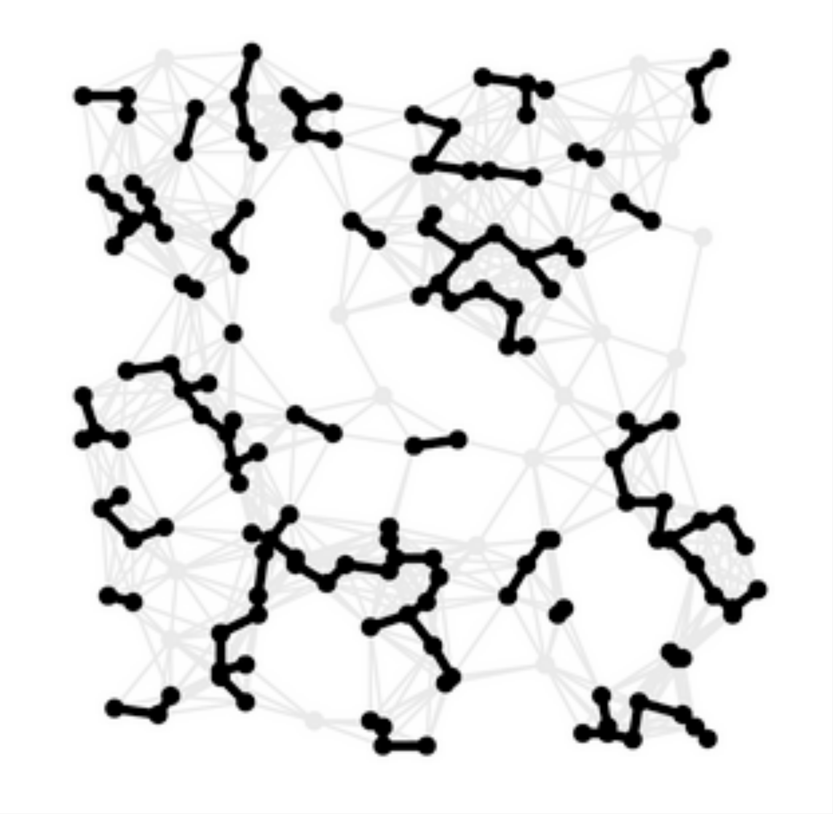


Observation. This is Kruskal's algorithm.
(stopping when k connected components)

Single-link clustering algorithm

"Well-known" algorithm in science literature for single-link clustering:

- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly k clusters.

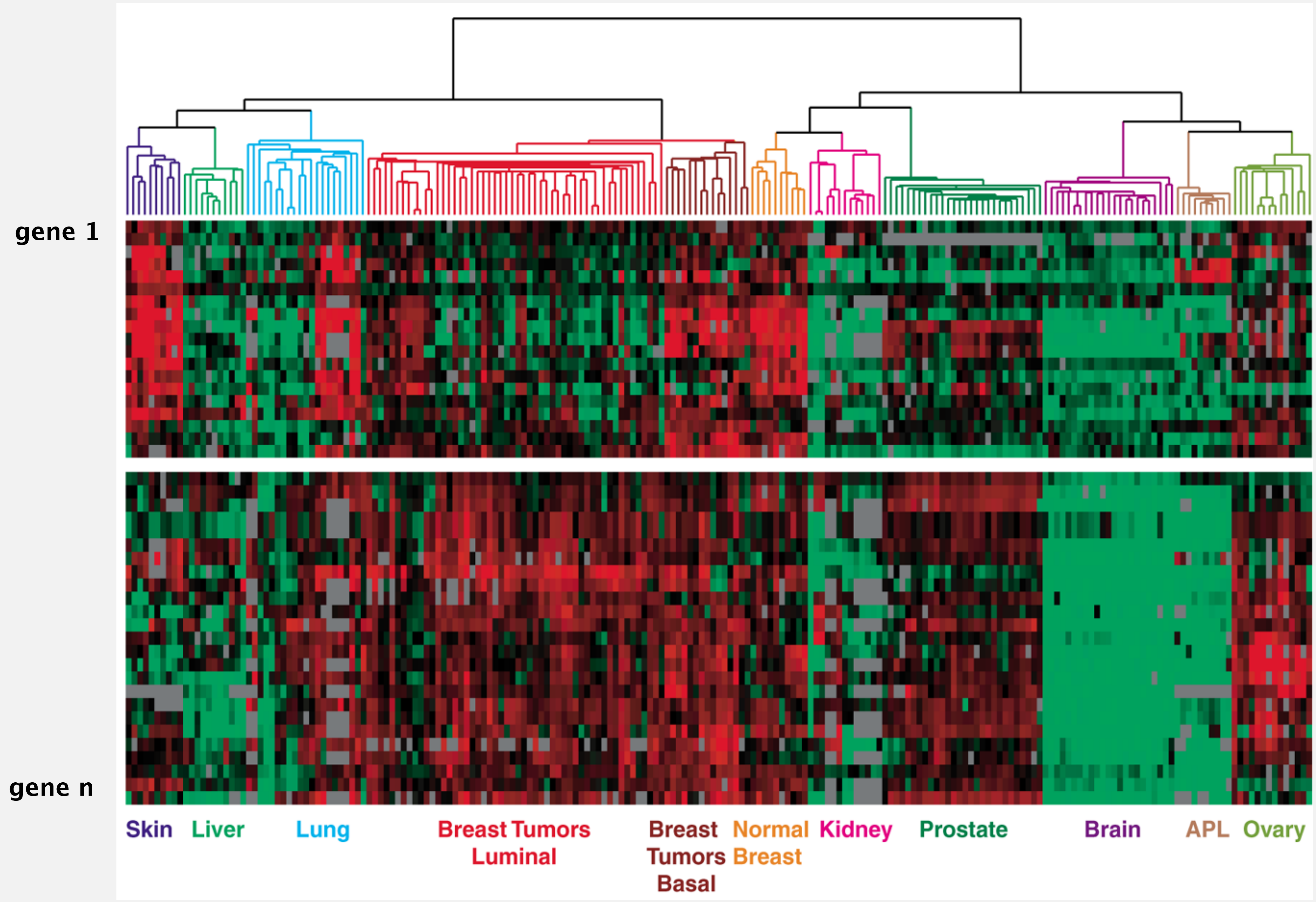


Observation. This is Kruskal's algorithm.
(stopping when k connected components)

Alternate solution. Run Prim; then delete $k-1$ max weight edges.

Dendrogram of cancers in human

Tumors in similar tissues cluster together.



Reference: Botstein & Brown group