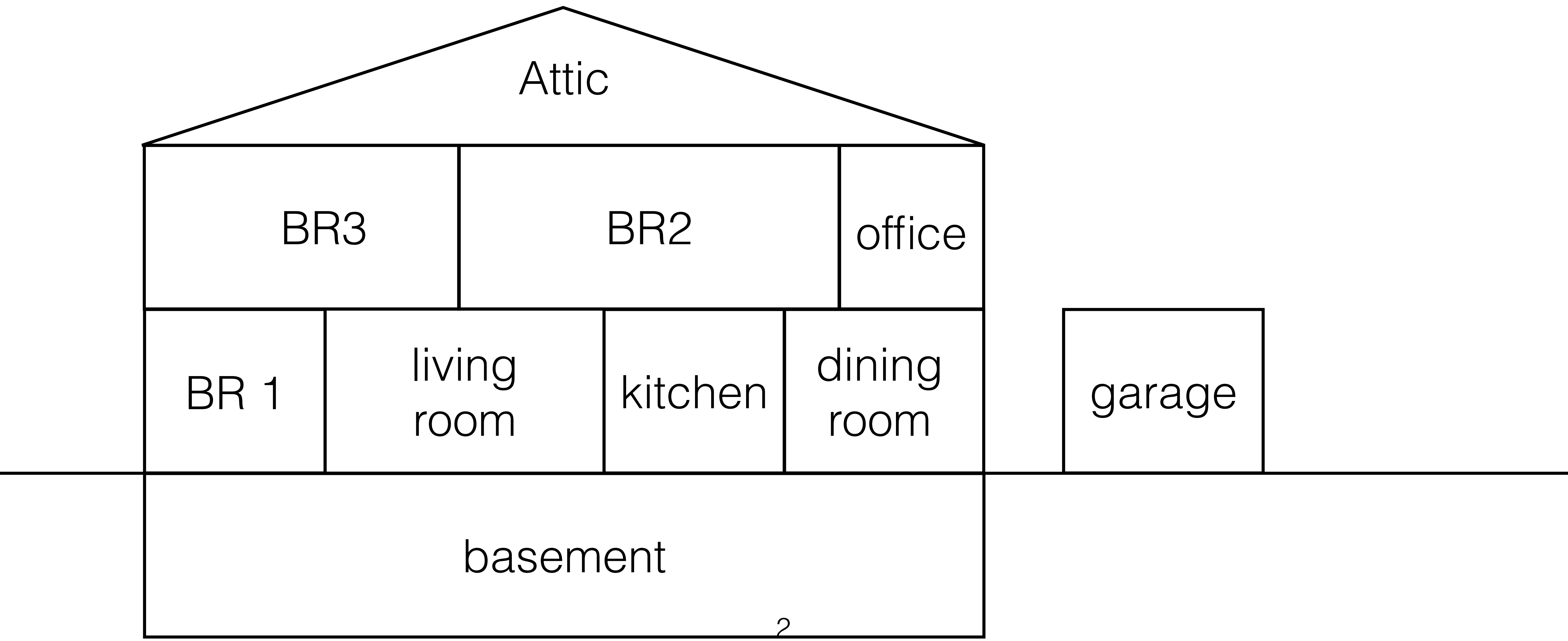
Honors Data Structures

Lecture 22: Minimum Spanning Trees

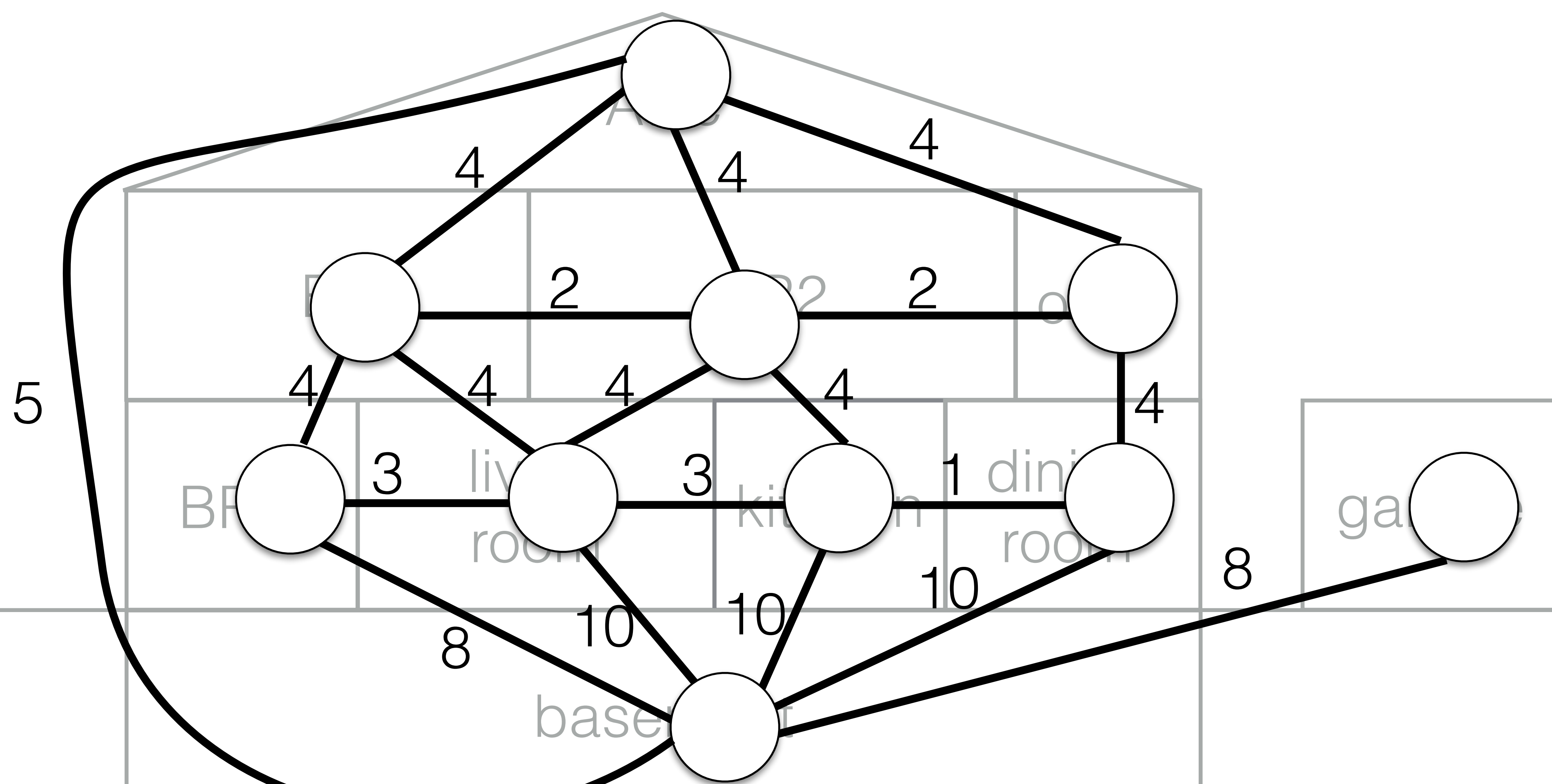
4/13/2022

Daniel Bauer

Designing a Home Network.

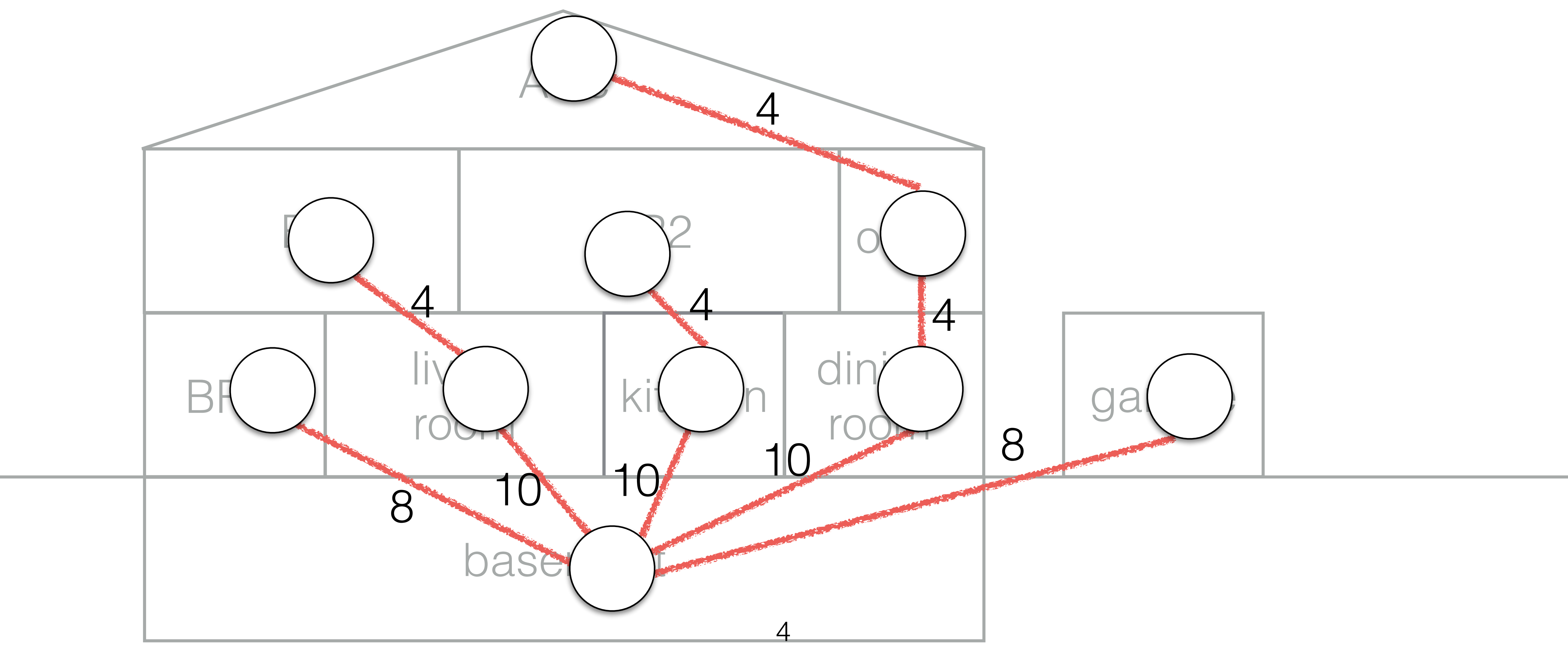


Designing a Home Network.



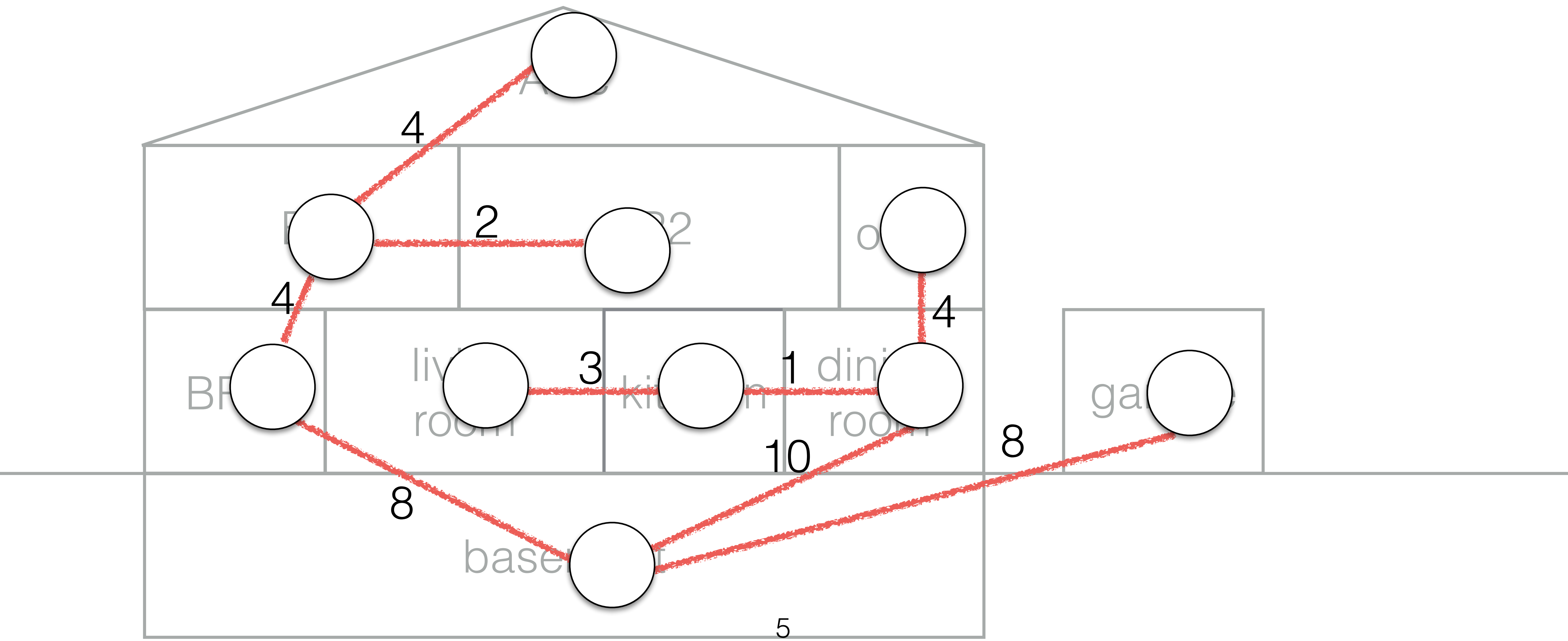
Designing a Home Network.

Total cost: 62



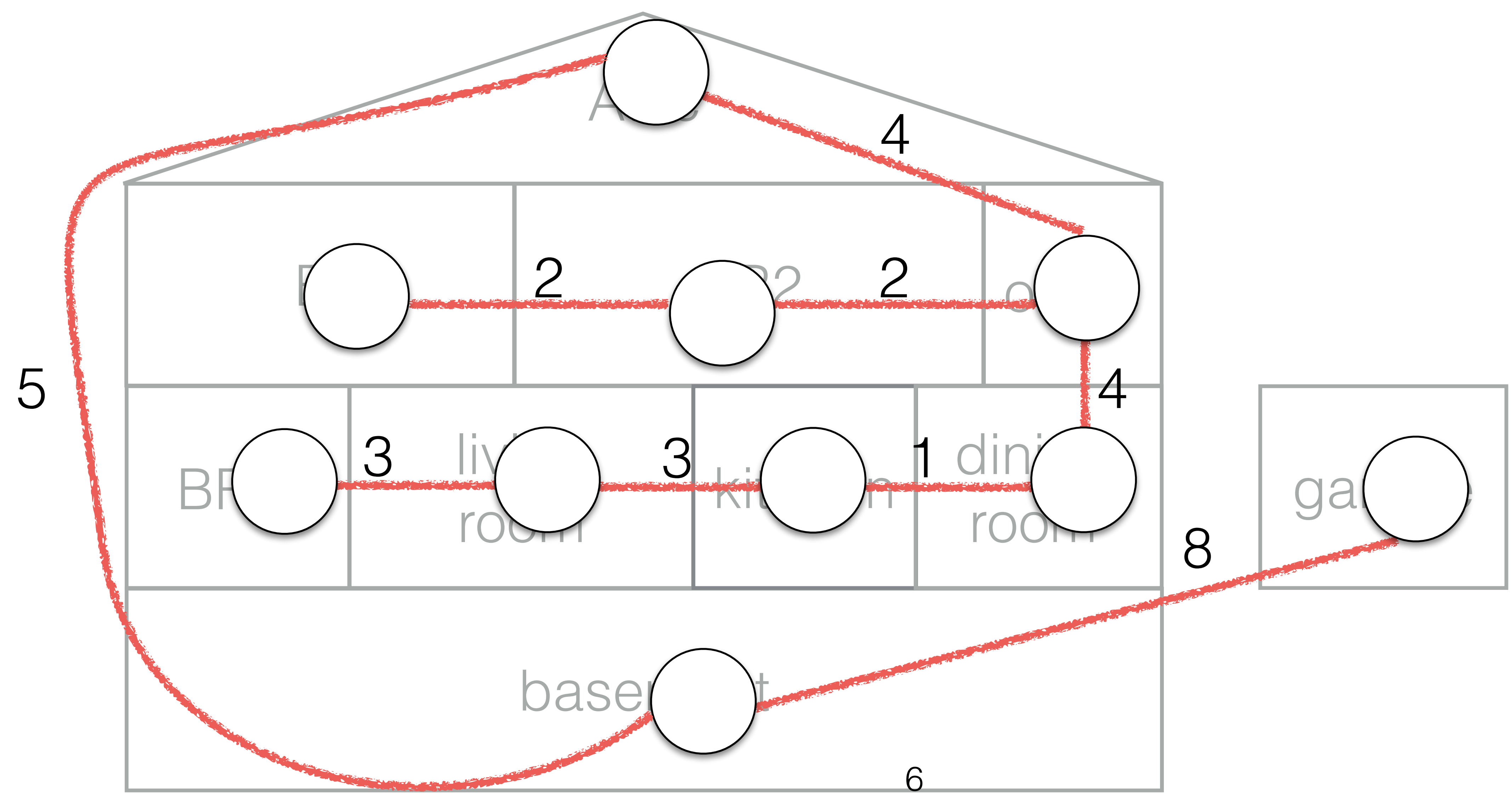
Designing a Home Network.

Total cost: 44



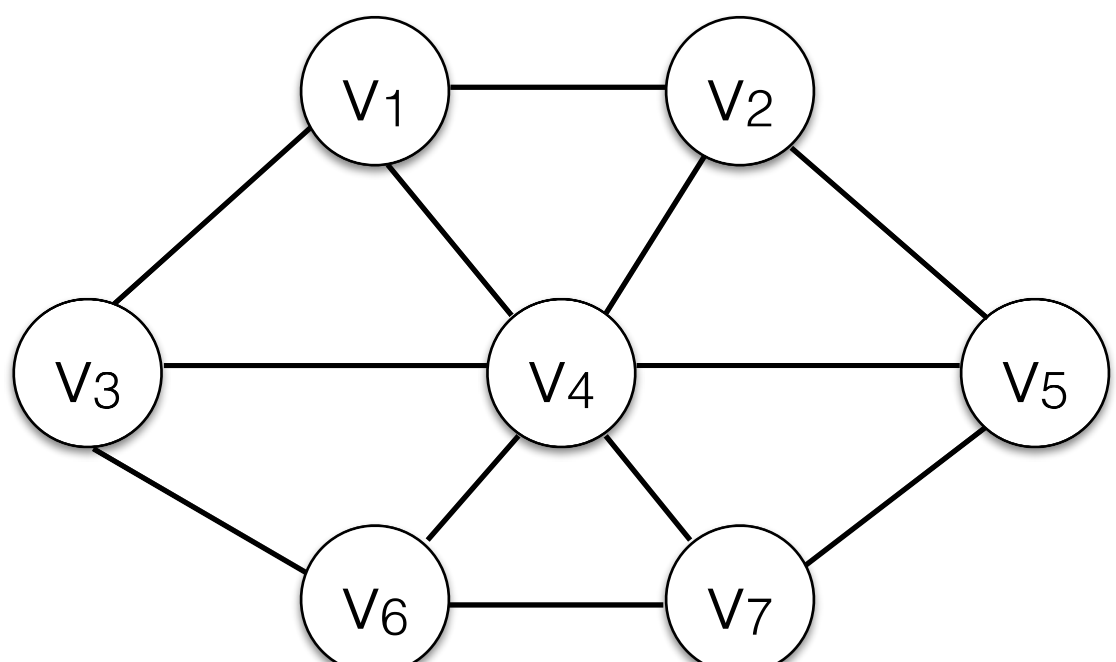
Designing a Home Network.

Total cost: 32



Spanning Trees

- Given an undirected, connected graph G=(V,E).
- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

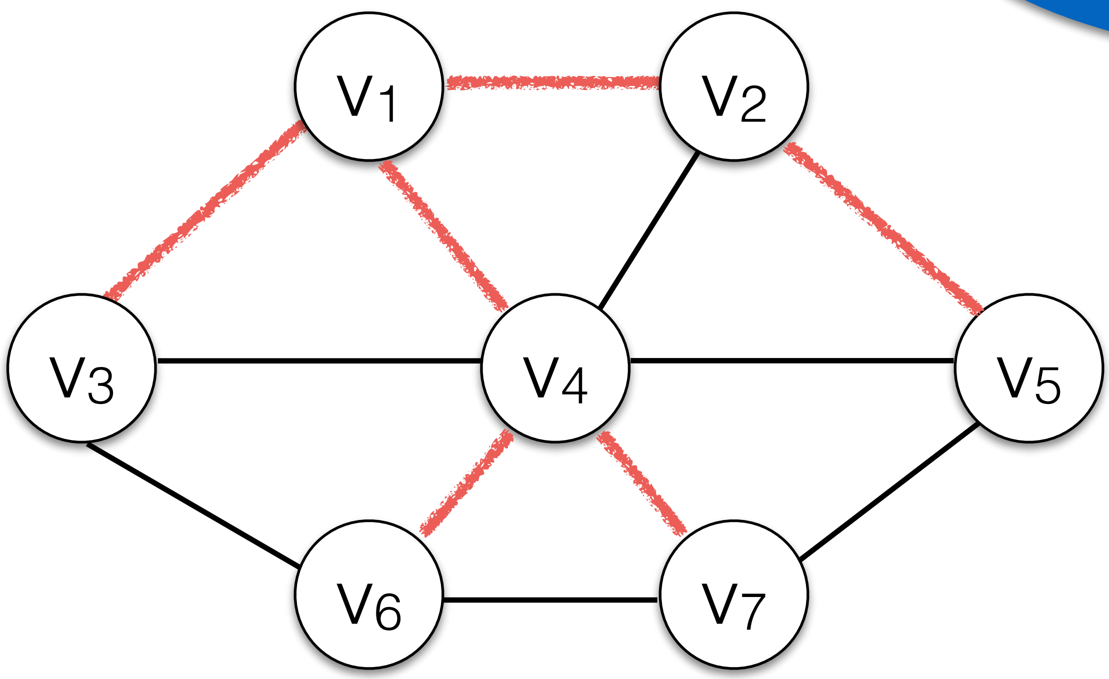


Spanning Trees

- Given an undirected, connected graph G=(V,E).

- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

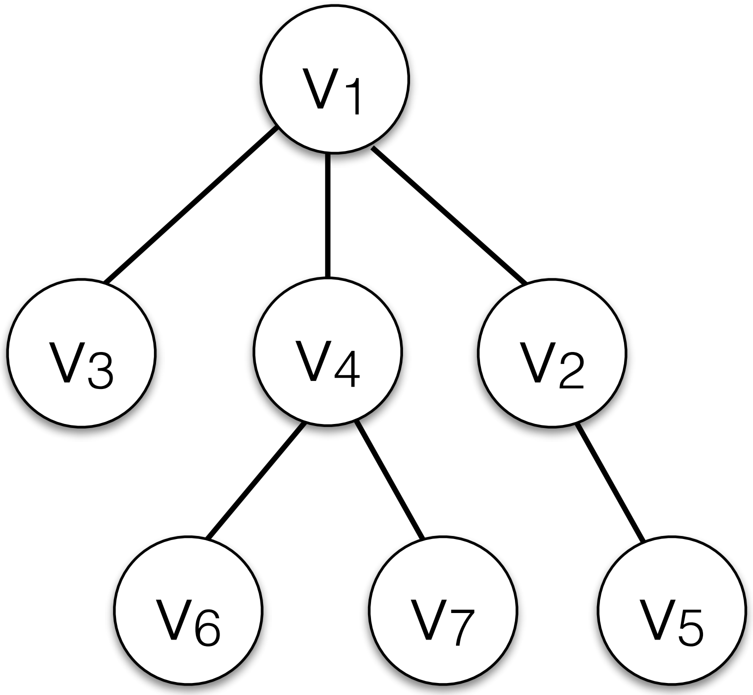
T is acyclic. There is a single path between any pair of vertices.

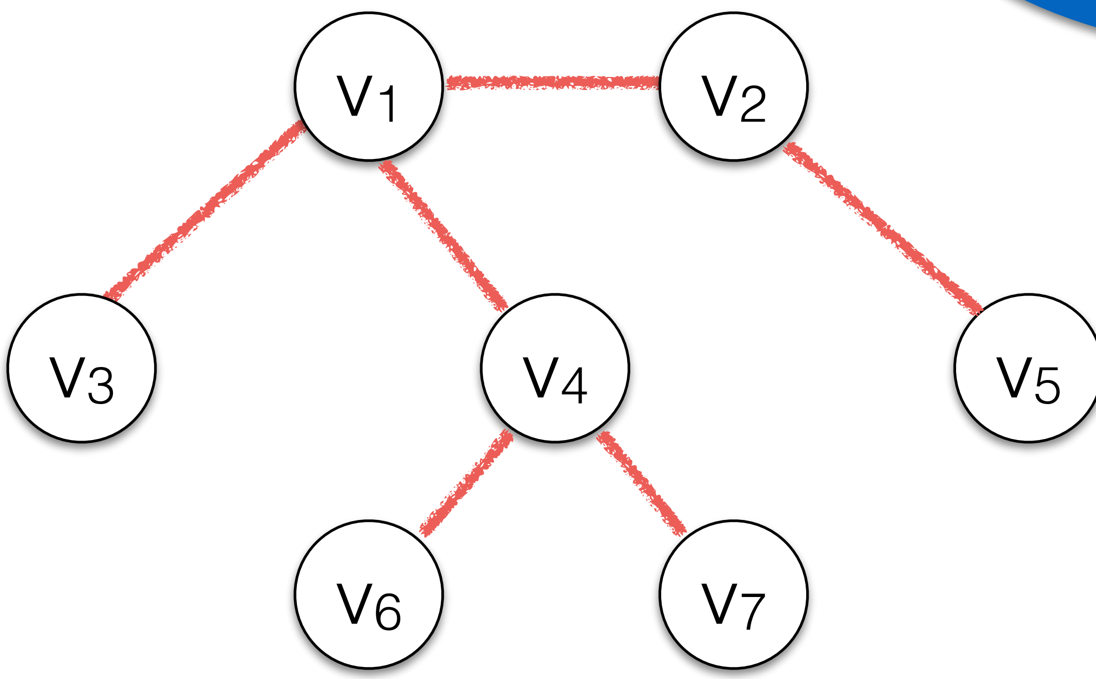


Spanning Trees

- Given an undirected, connected graph G=(V,E).
- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

T is acyclic. There is a single path between any pair of vertices.

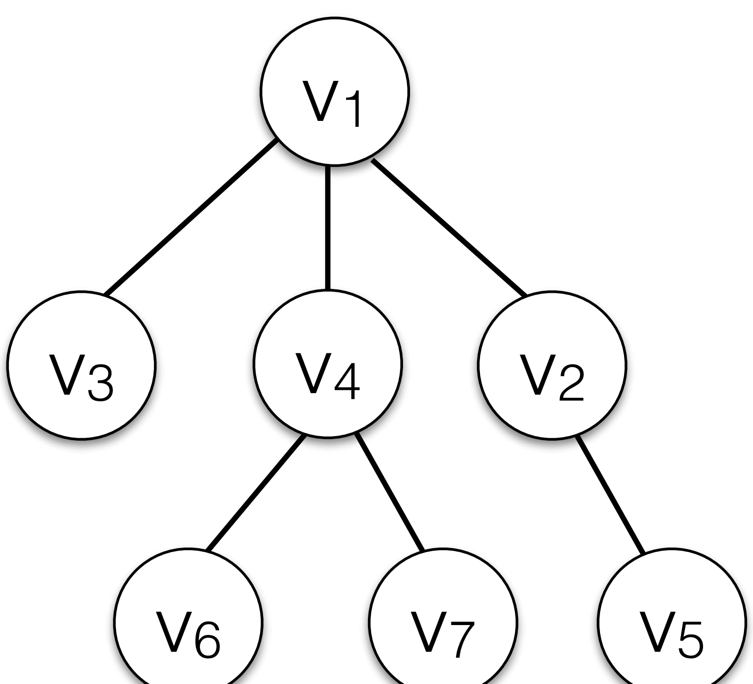


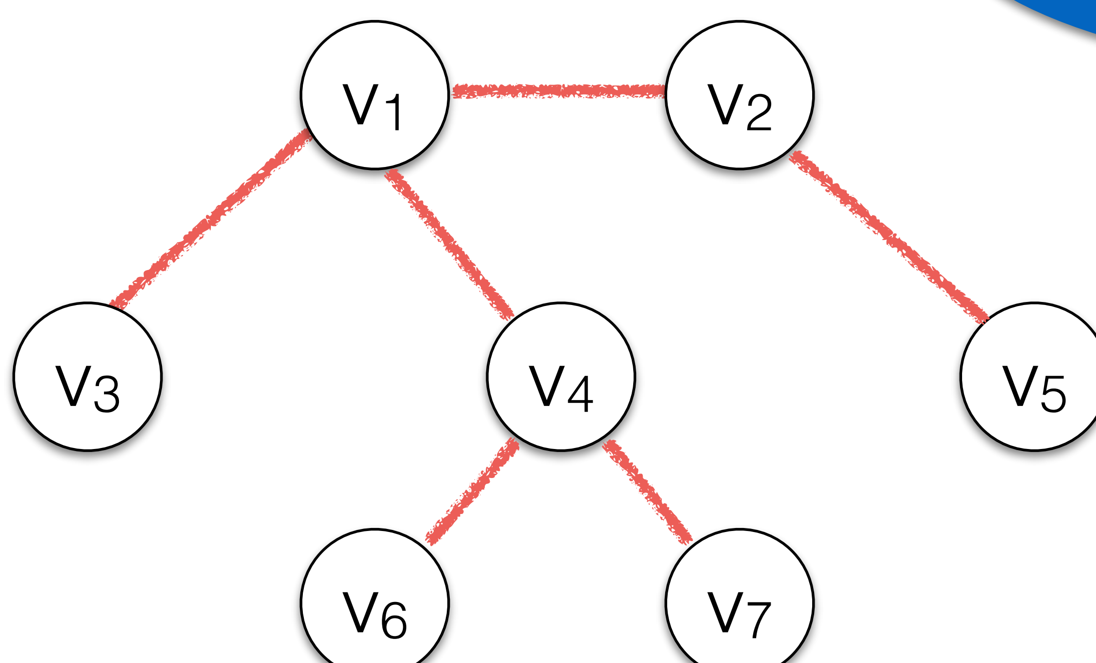


Spanning Trees

- Given an undirected, connected graph G=(V,E).
- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

T is acyclic. There is a single path between any pair of vertices.





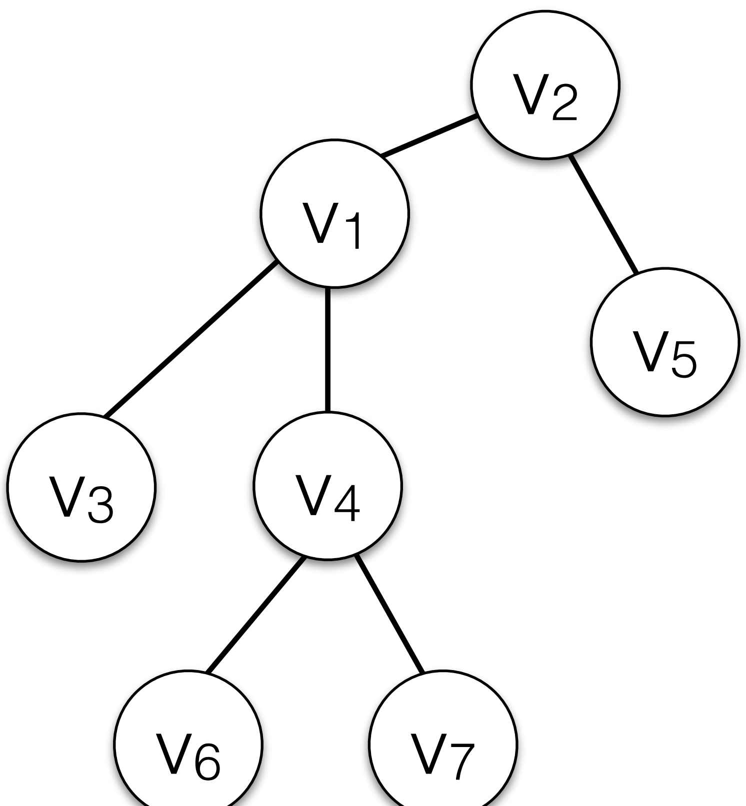
Any node can be the root of the spanning tree.

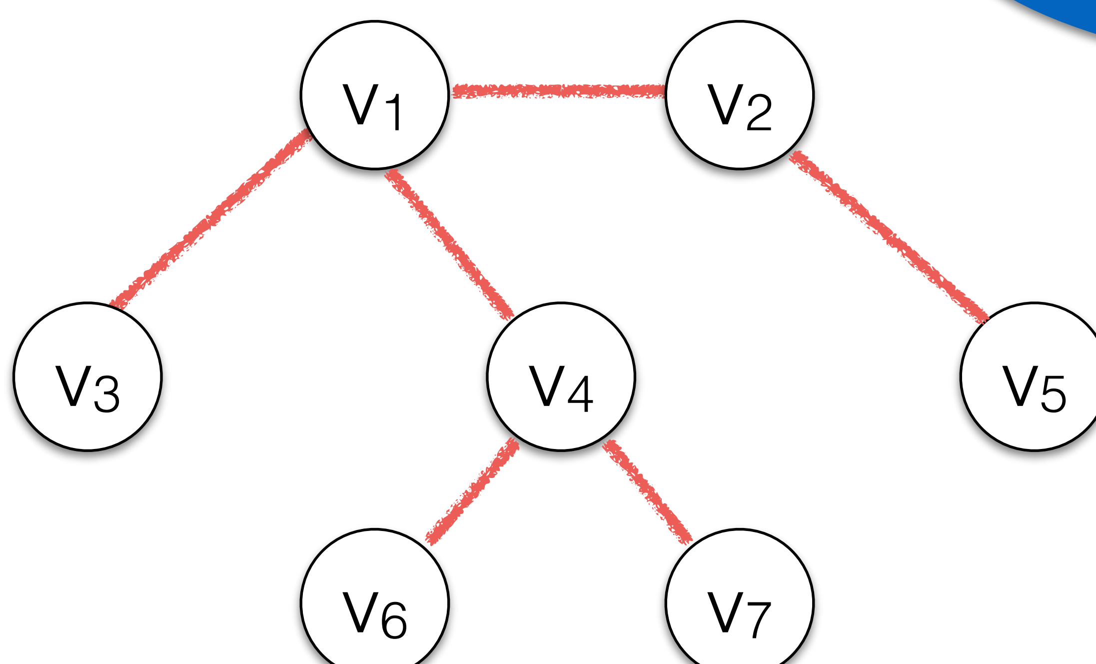
Spanning Trees

- Given an undirected, connected graph G=(V,E).

- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

T is acyclic. There is a single path between any pair of vertices.

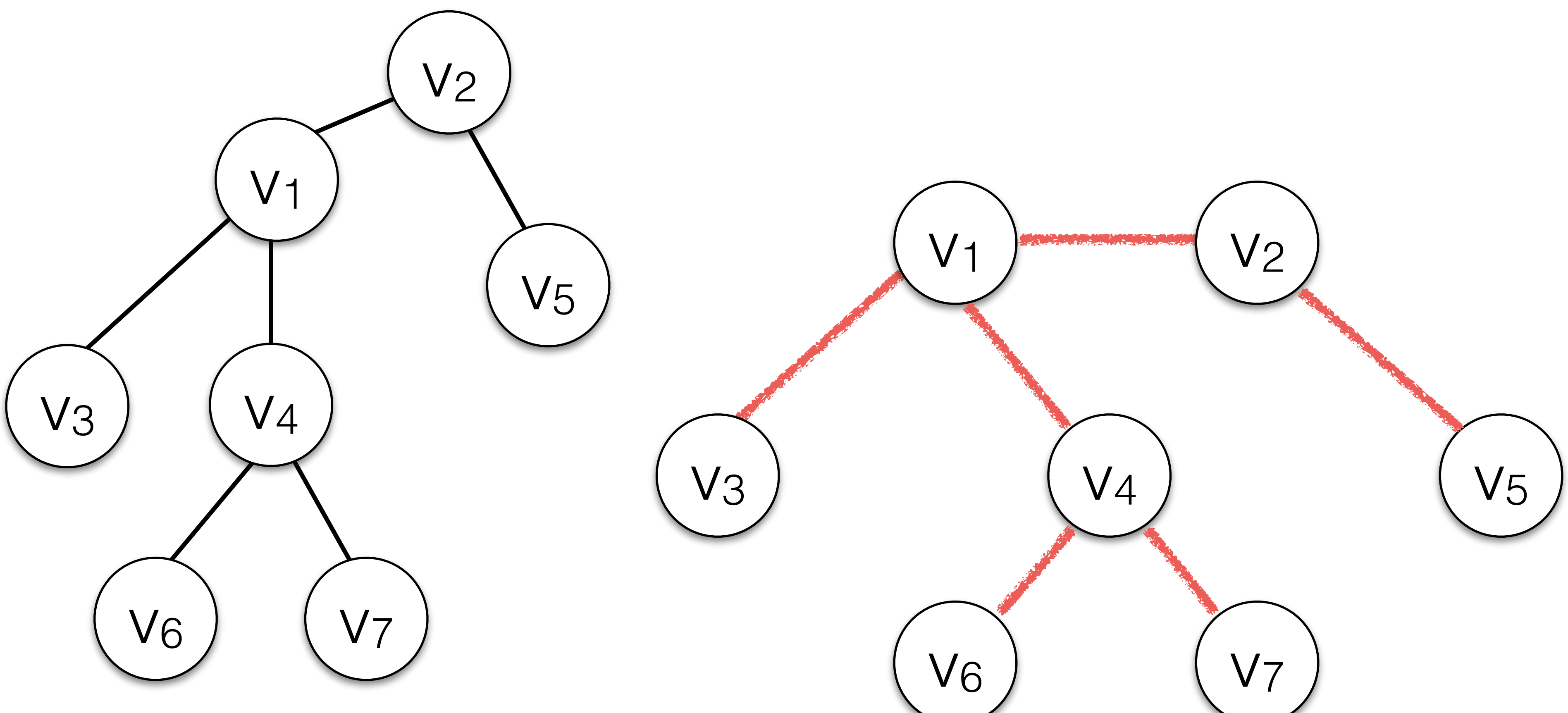




Any node can be the root of the spanning tree.

Spanning Trees

- Given an undirected, connected graph G=(V,E).
- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$



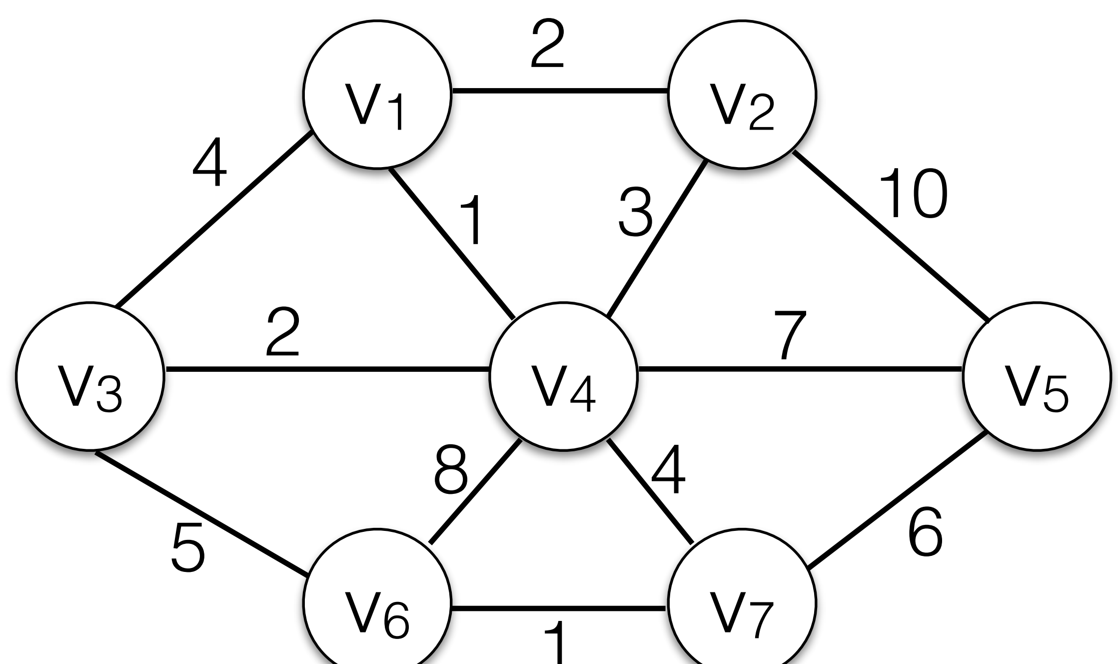
Number of edges in a spanning tree: |V|-1

Spanning Trees, Applications

- Constructing a computer/power networks (connect all vertices with the smallest amount of wire).
- Clustering Data.
- Dependency Parsing of Natural Language
  (directed graphs. This is harder).
- Constructing mazes.
- …
- Approximation algorithms for harder graph problems.
- …

Minimum Spanning Trees

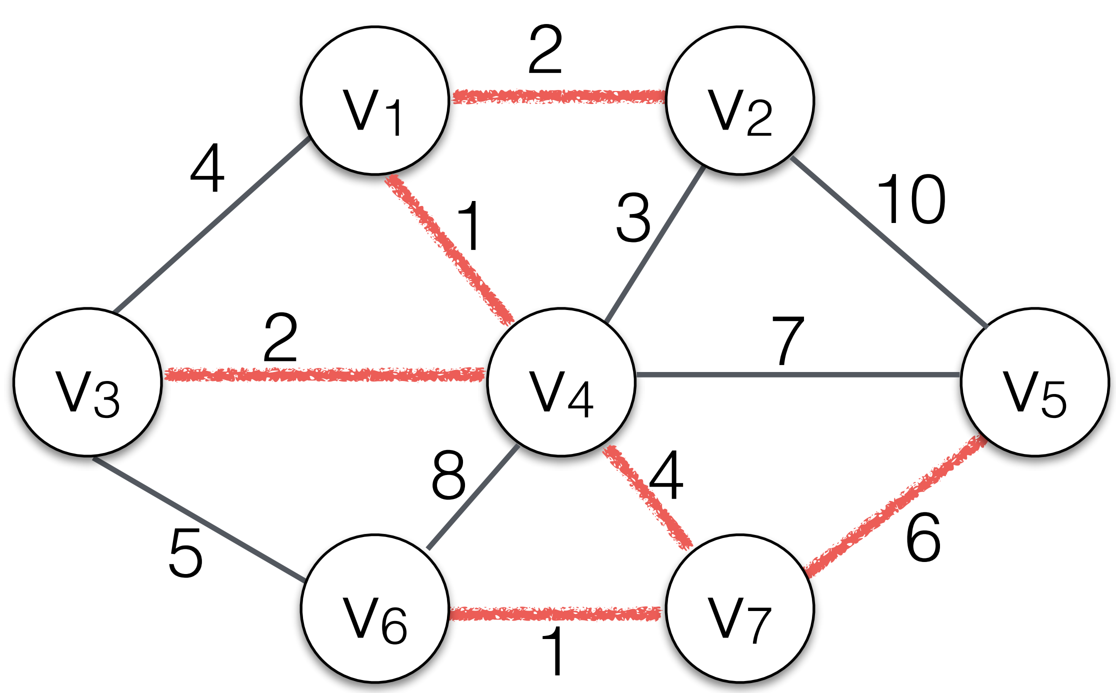
- Given a *weighted* undirected graph G=(E,V).
- A ***minimum spanning tree*** is a spanning tree with the minimum sum of edge weights.



Minimum Spanning Trees

- Given a *weighted* undirected graph G=(E,V).
- A ***minimum spanning tree*** is a spanning tree with the minimum sum of edge weights.

Total cost = 16

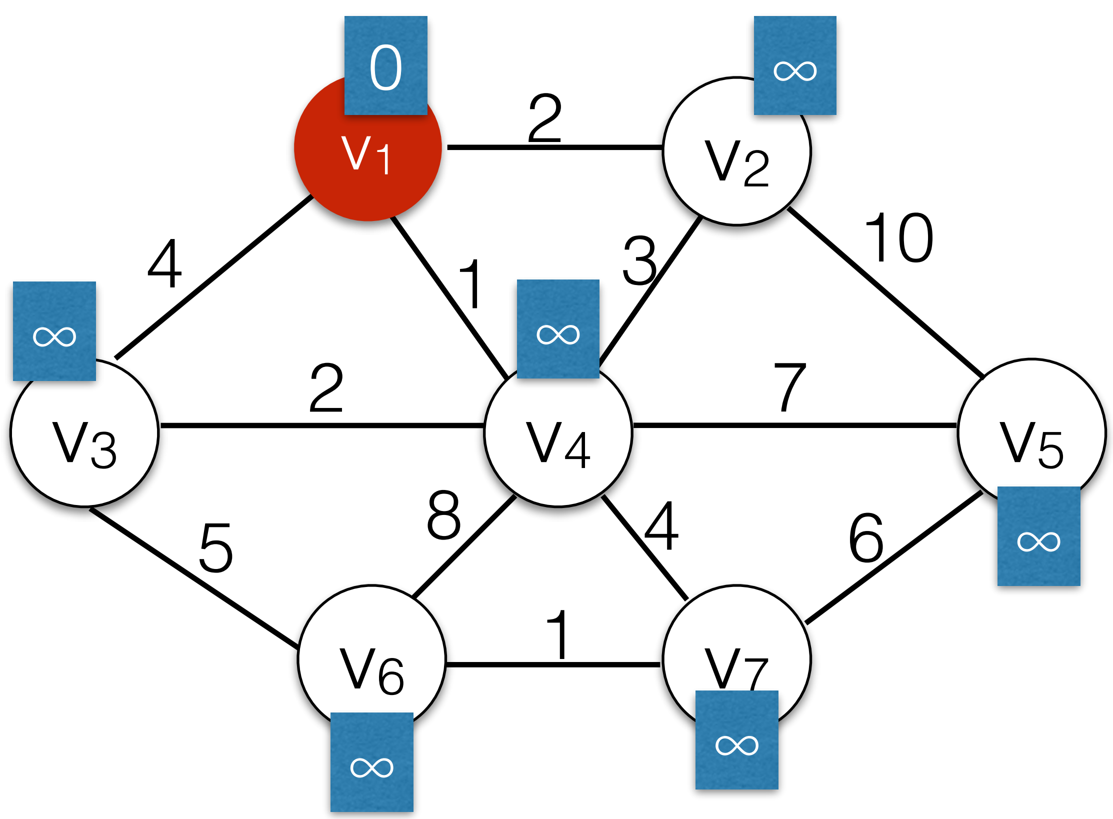


(often there are multiple minimum spanning trees)

Prim’s Algorithm for finding MSTs

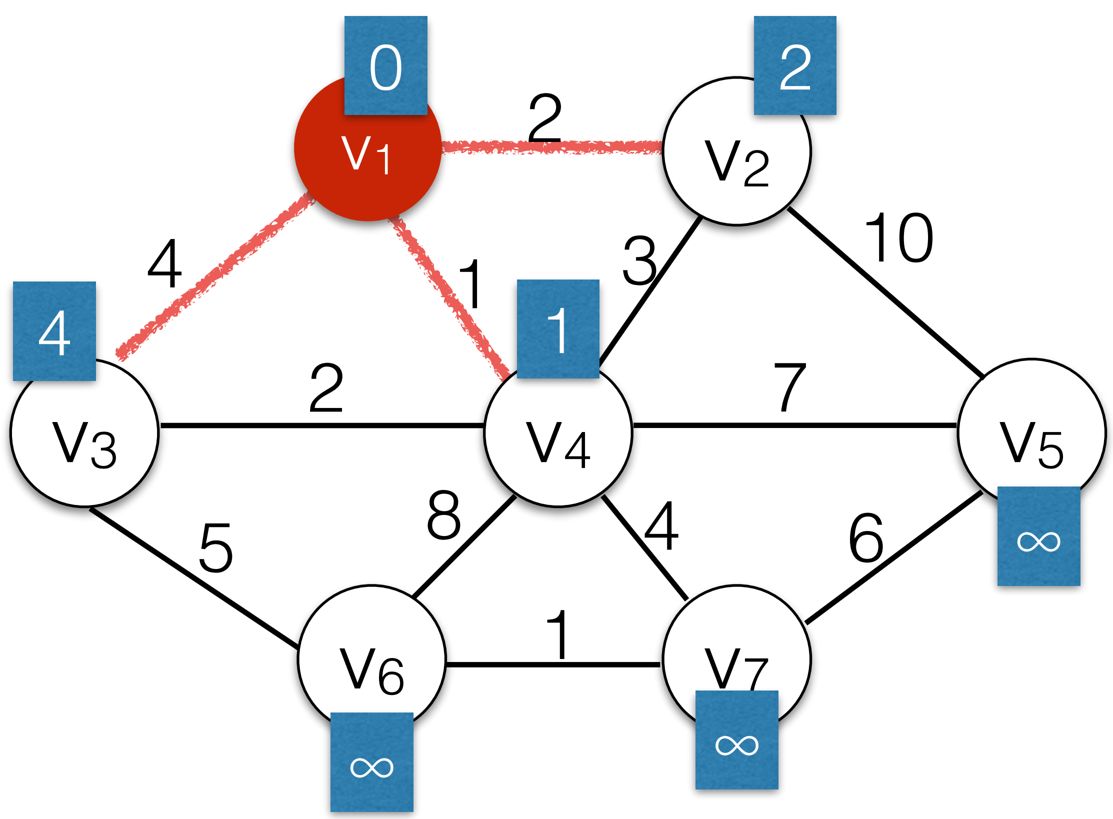
- Another greedy algorithm. A variant of Dijkstra’s algorithm.
- Cost annotations for each vertex v reflect the lowest weight of an edge connecting v to other *vertices already visited.*
    - That means there might be a lower-weight edge from another vertices that have not been seen yet.
- Keep vertices on a priority queue and always expand the vertex with the lowest cost annotation first.

Prim's Algorithm



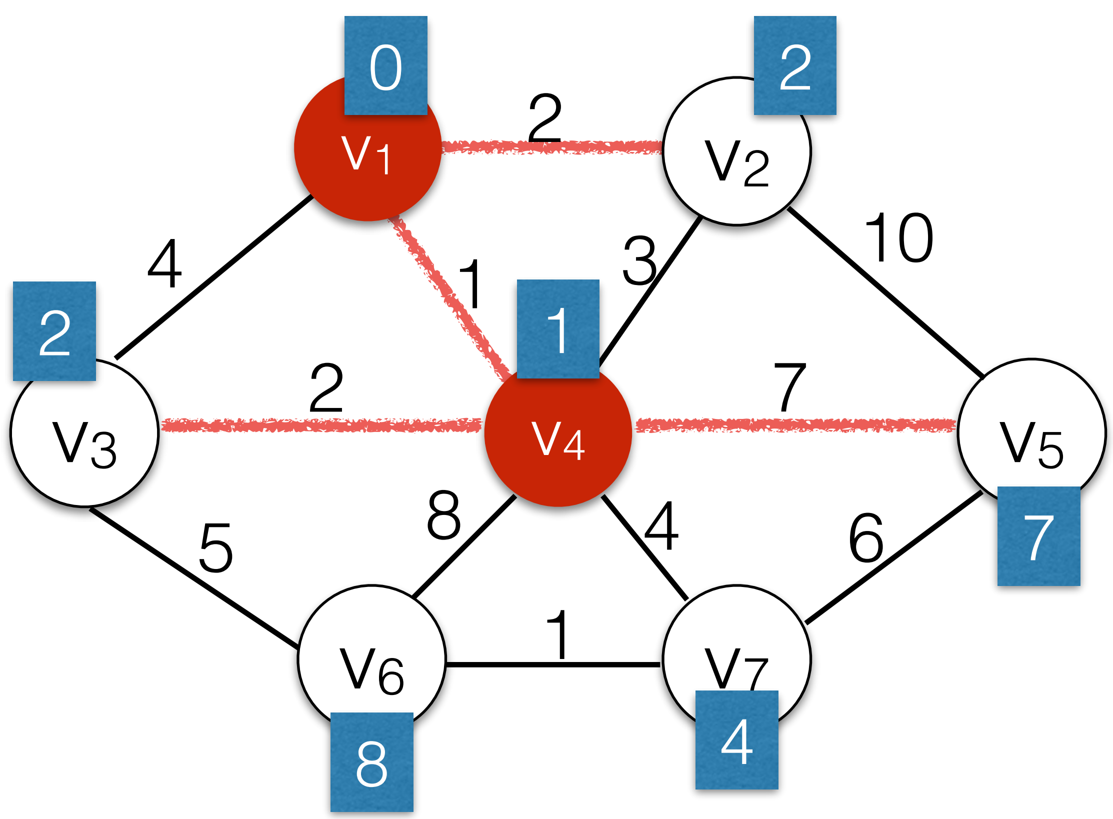

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim’s Algorithm



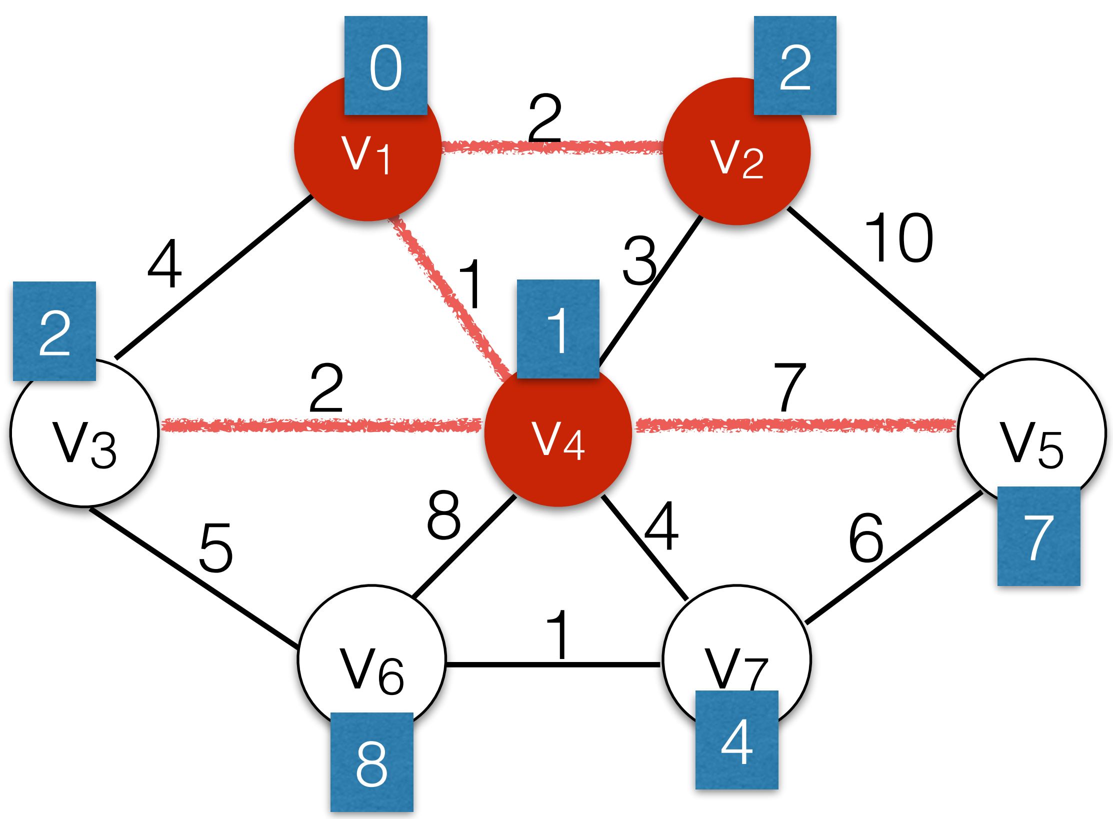

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm



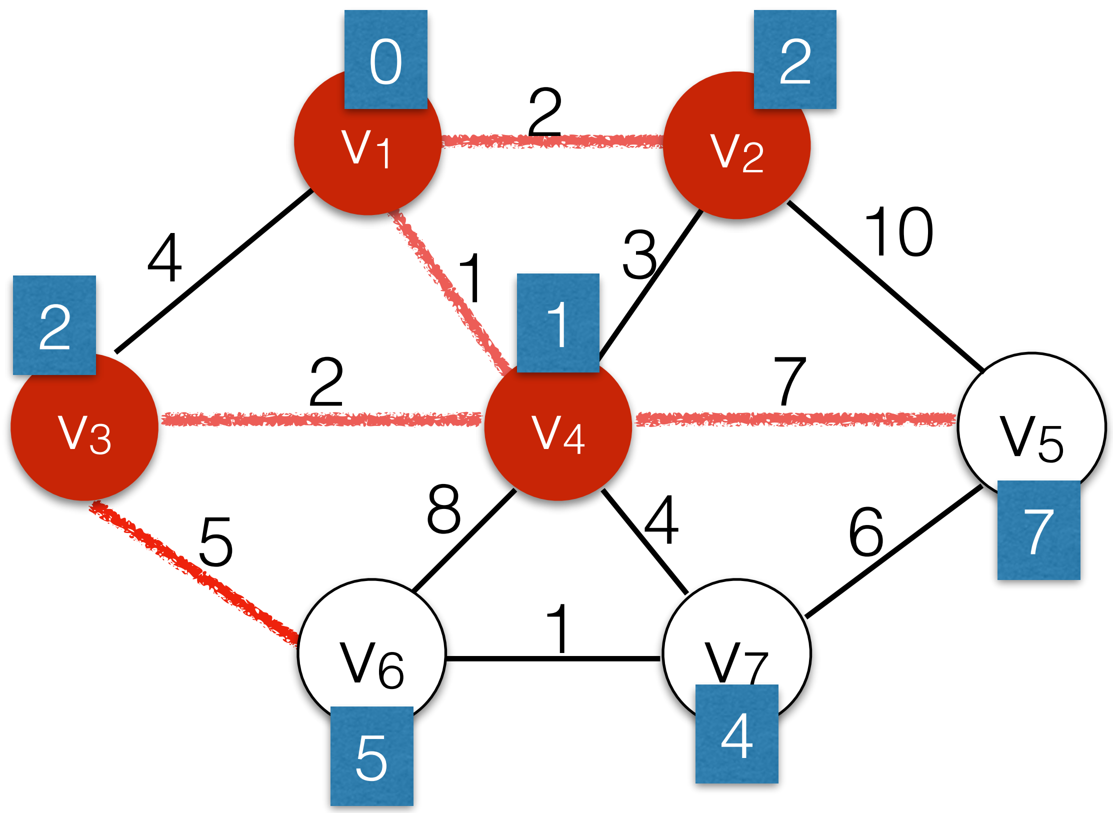

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm



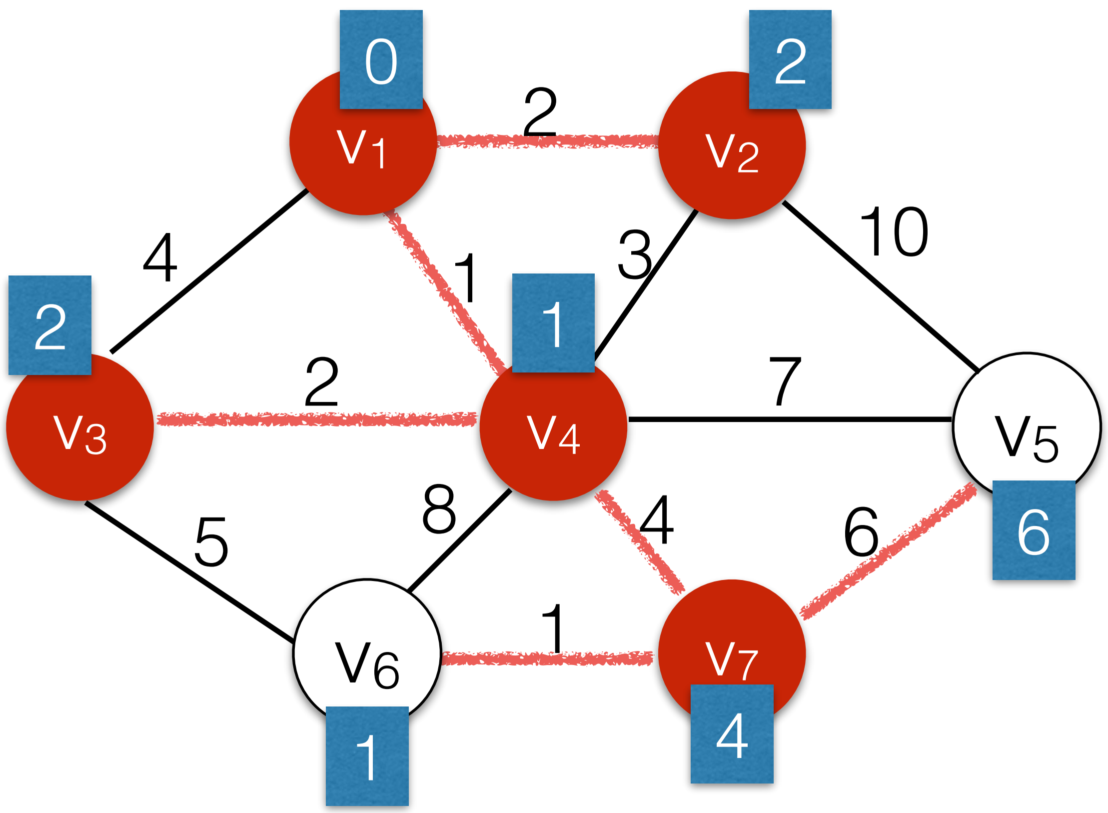

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm



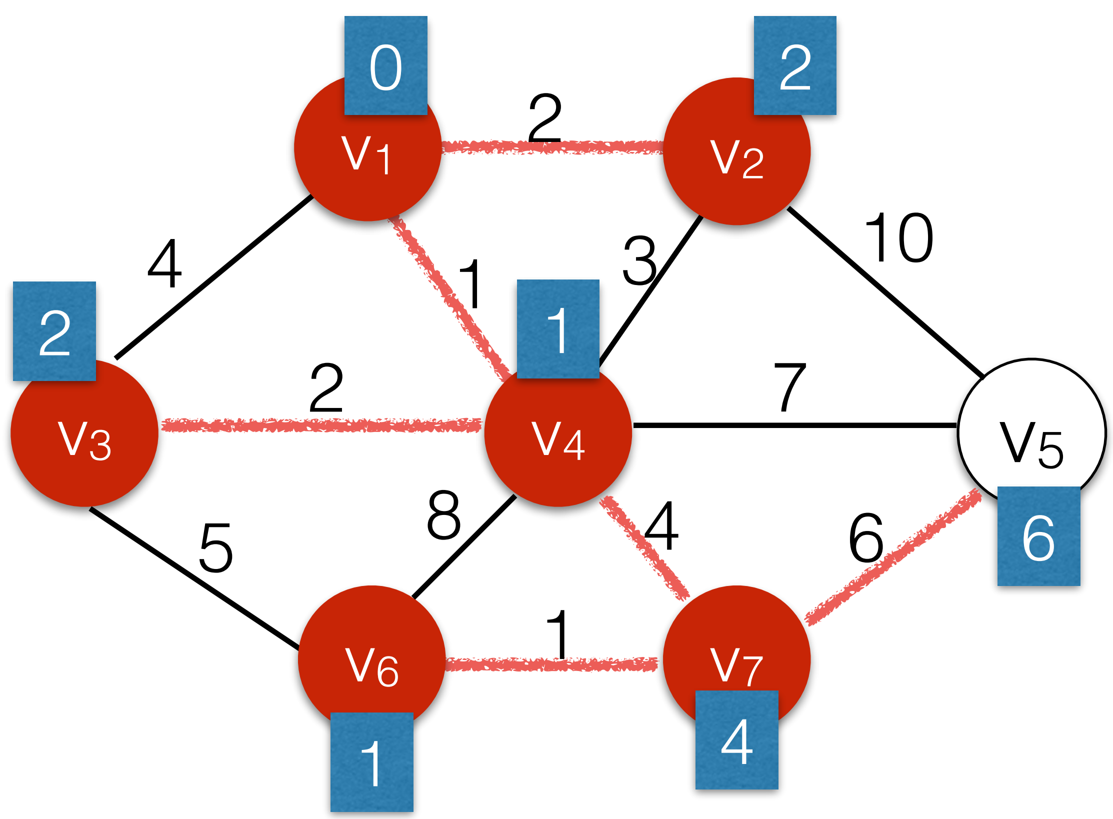

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm



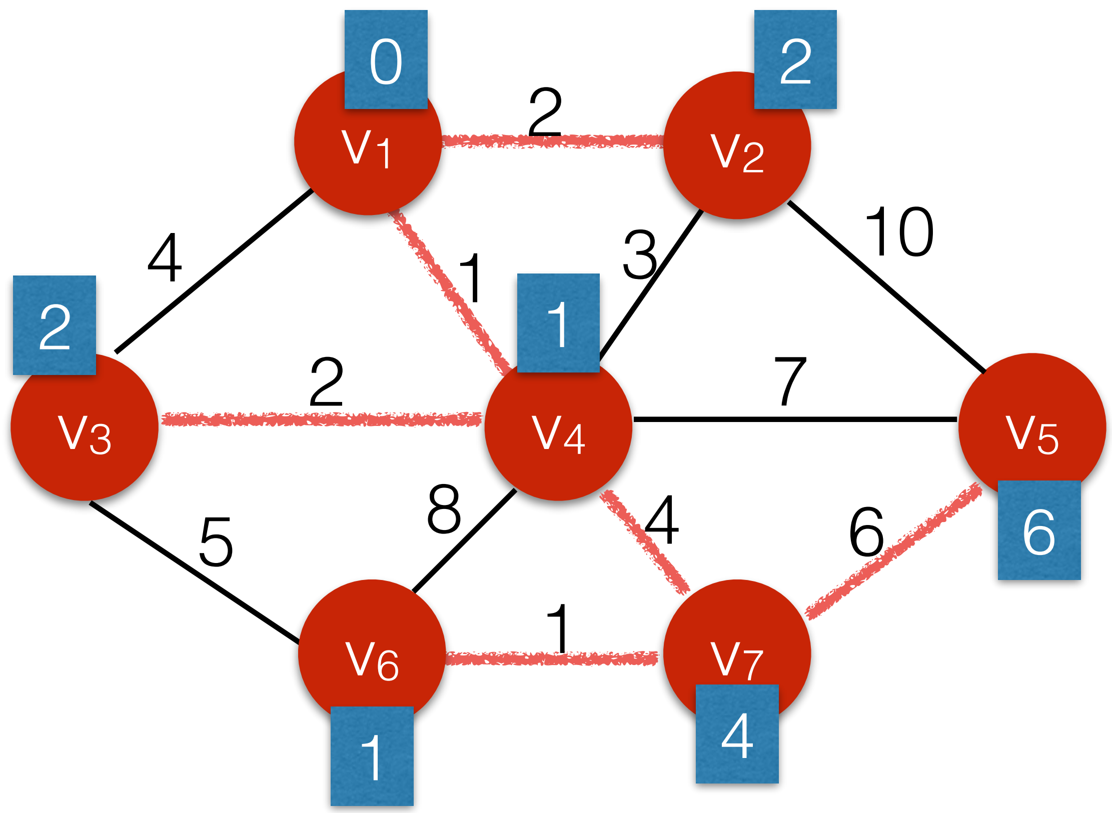

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm



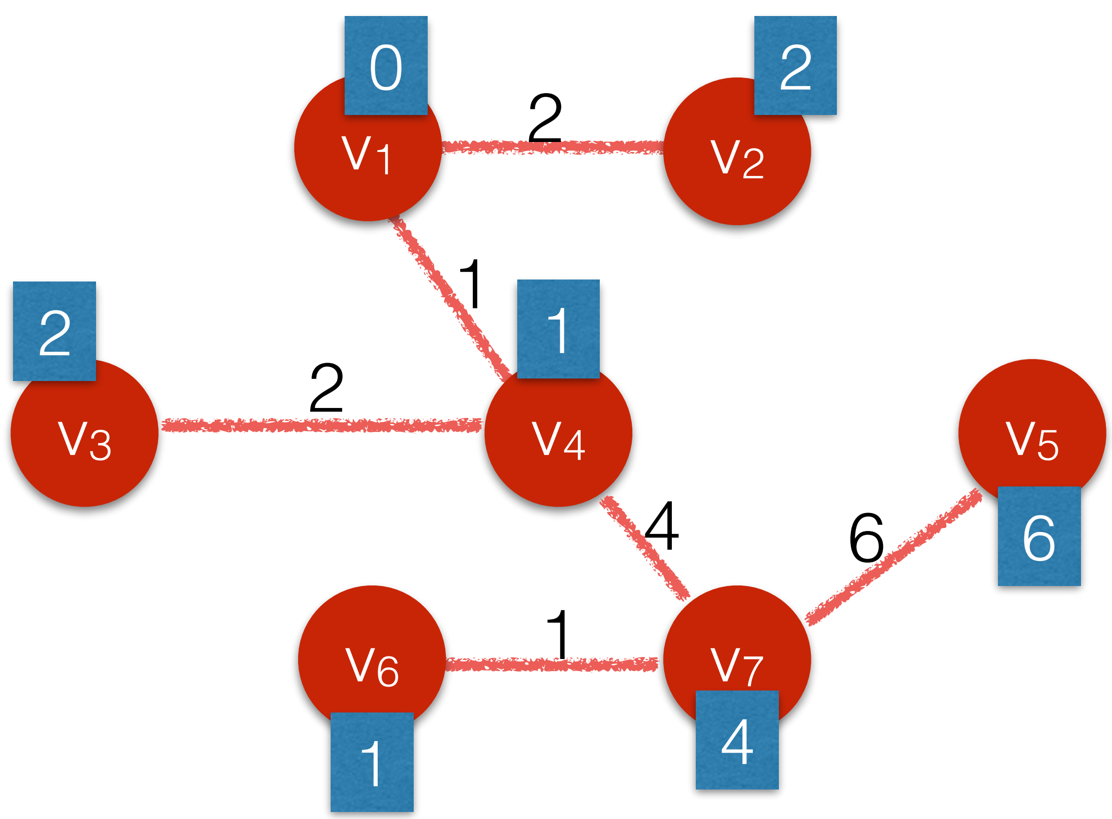

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm




```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

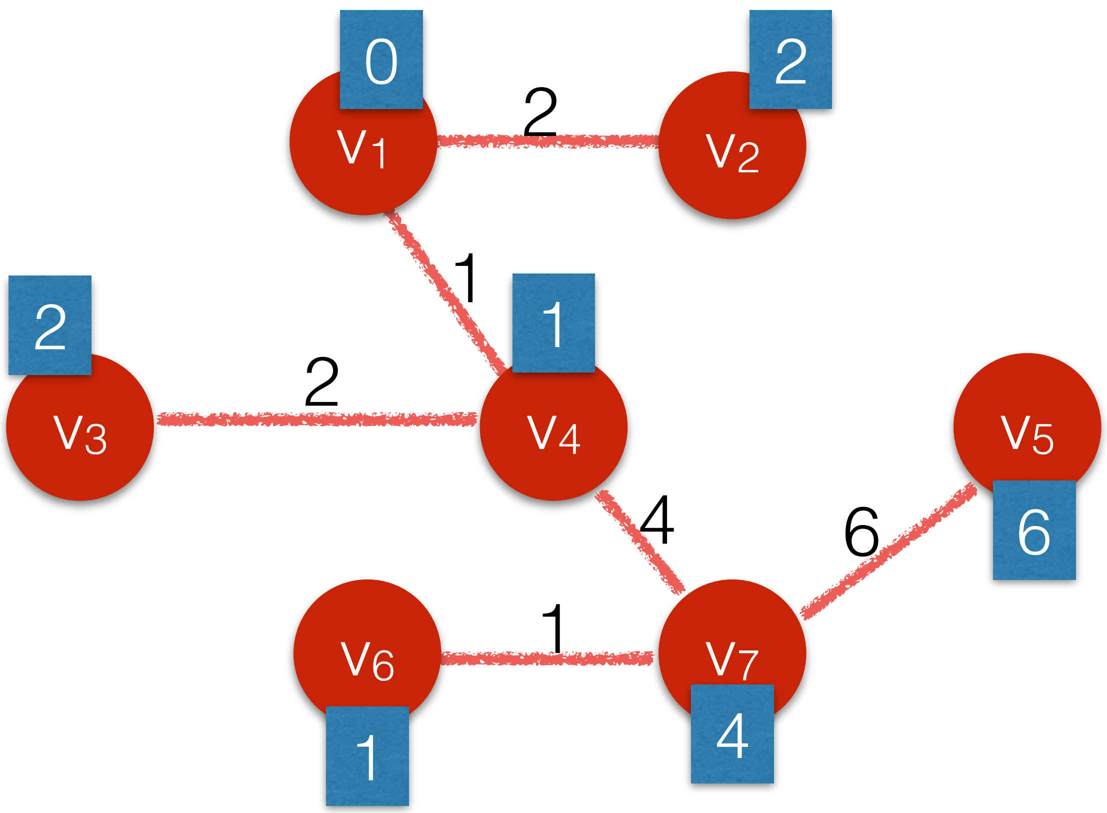
Prim's Algorithm




```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm

Running time: Same as Dijkstra's Algorithm
O(|E| log |V|)



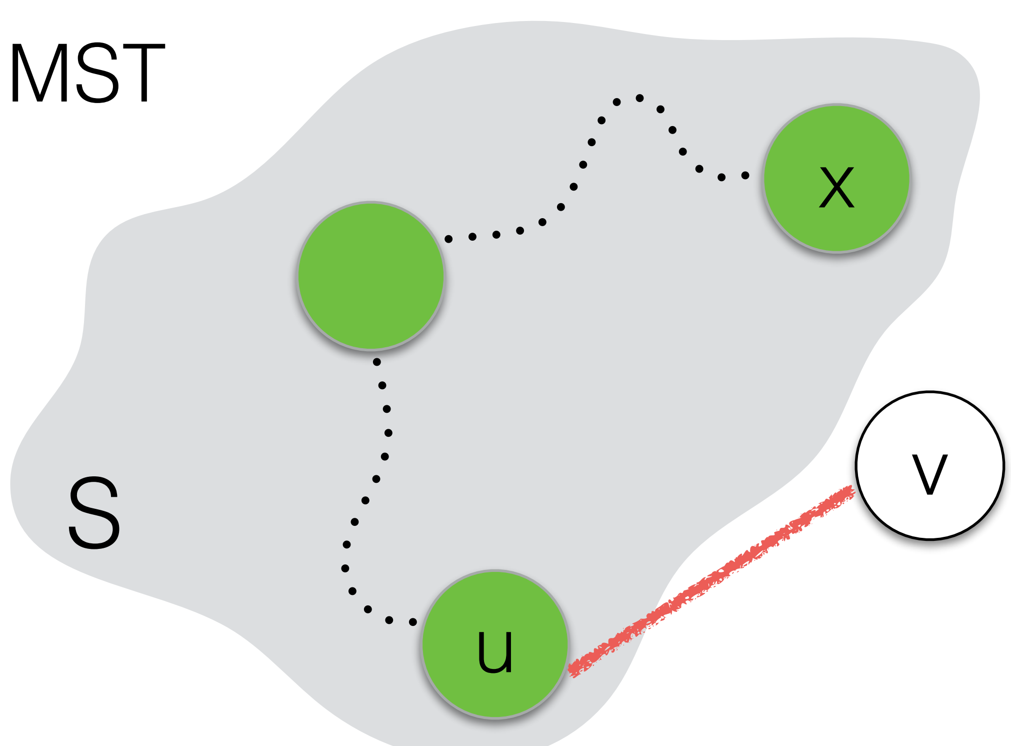

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
     if (cost(u,v) < v.cost):
        v.cost = cost(u,v)
        v.prev = u
        q.insert(v)
```

Prim's Algorithm - Correctness

- Observation:
  - A spanning tree has |V|-1 edges.
  - Choose any root. Then all vertices (except for the root) will have exactly one parent.
  - Adding ANY edge to an MST will create a cycle.

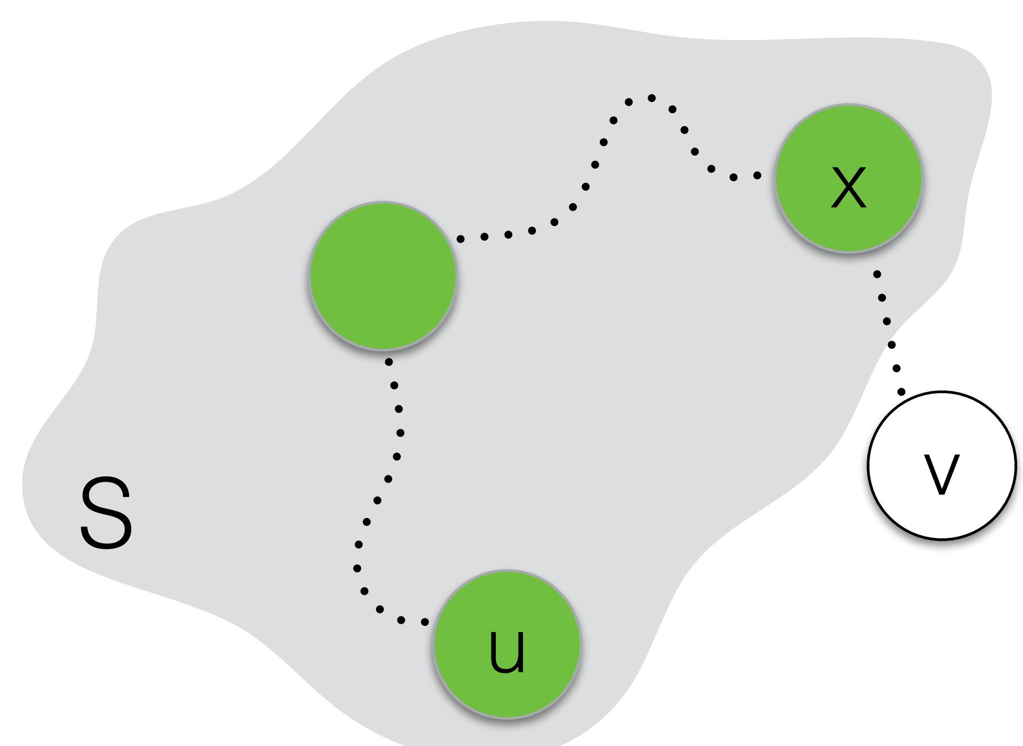
Prim's Algorithm - Correctness

- Show: Any spanning tree produces by Prim's algorithm is minimal.
- Proof by contradiction: Assume a tree produced by Prim's was not an MST.
- Then we must have chosen a first edge (u,v) that was not consistent with an MST at some point.
  - Let T be a tree on the subset S prior to adding (u,v).
  - Let M be an extension of T that is an MST of the graph.



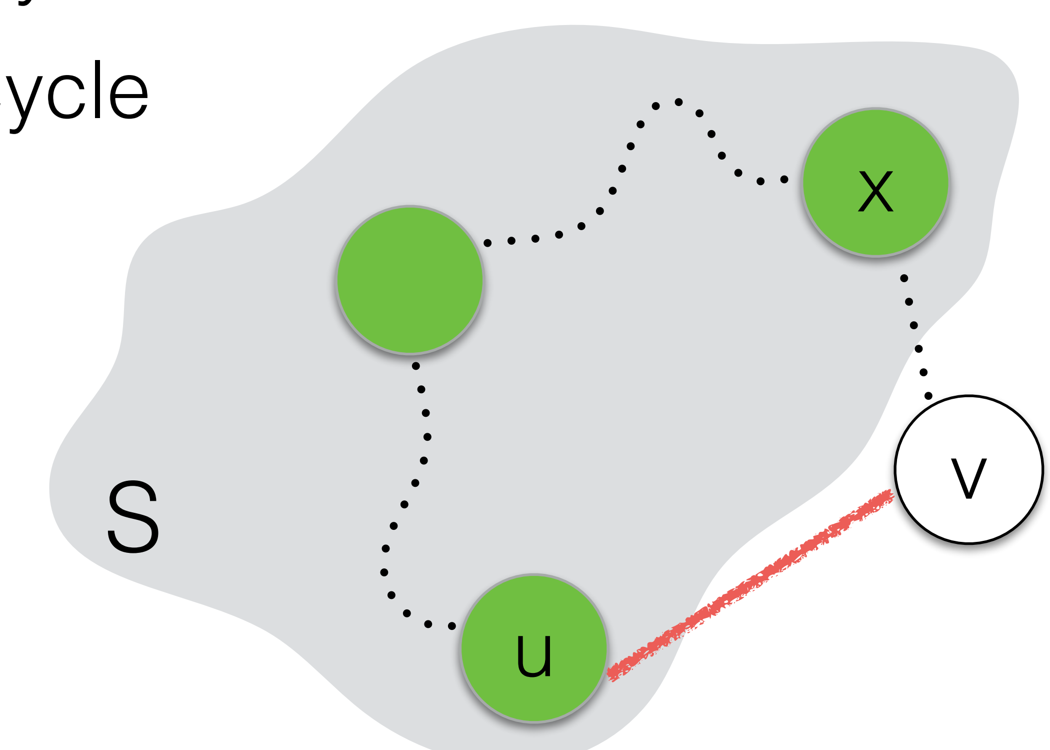
Prim’s Algorithm - Correctness

- Let T be a tree on the subset S prior to adding (u,v).
- Let M be an extension of T that is an MST of the graph.
- All other edges in M that we could have chosen instead of (u,v) (at the same time) must have higher cost than (u,v), or Prim’s algorithm would have chosen them.



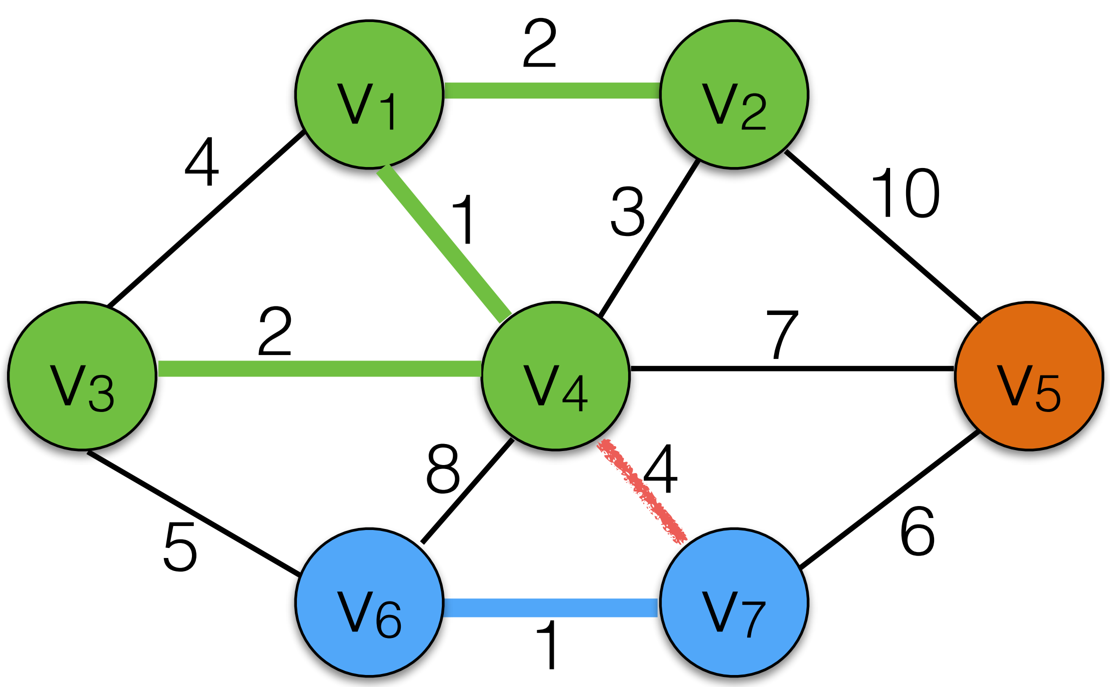
Prim’s Algorithm - Correctness

- Let T be a tree on the subset S prior to adding (u,v).
- Let M be an extension of T that is an MST of the graph.
- All other edges in M that we could have chosen instead of (u,v) (at the same time) must have higher cost than (u,v), or Prim’s algorithm would have chosen them.
- Adding (u,v) to M would produce a cycle.
- Removing any other edge from the cycle would restore the spanning tree. Because (u,v,) has a lower cost, this would lower the total cost of M.
- Therefore M could not have been minimal.



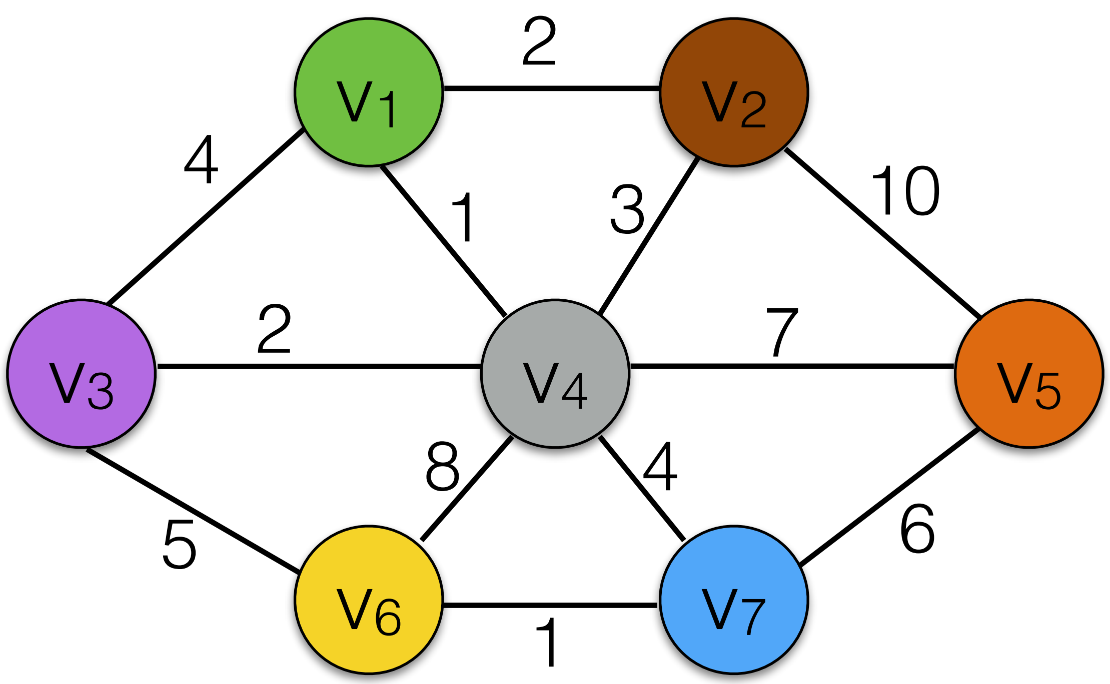
Kruskal's Algorithm for finding MSTs

- Kruskal's algorithm maintains a "forest" of trees.
- Initially each vertex is its own tree.
- Sort edges by weight. Then attempt to add them one-by one. Adding an edge merges two trees into a new tree.
- If an edge connects two nodes that are already in the same tree it would produce a cycle. Reject it.



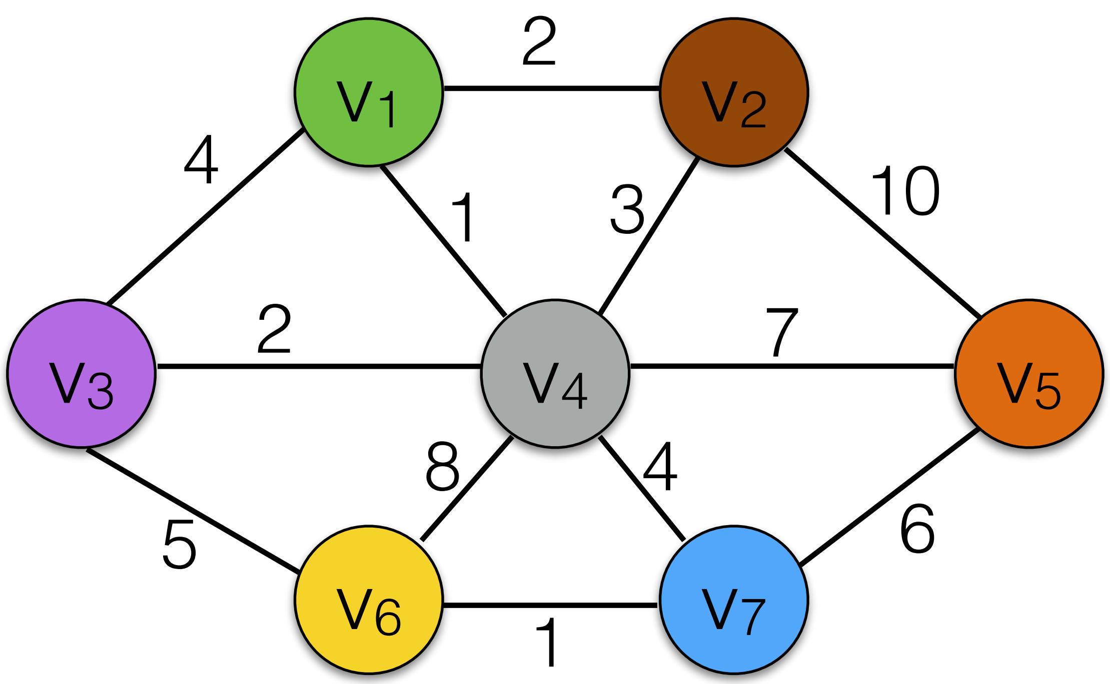
Kruskal's Algorithm

Sort edges (or keep them on a heap)



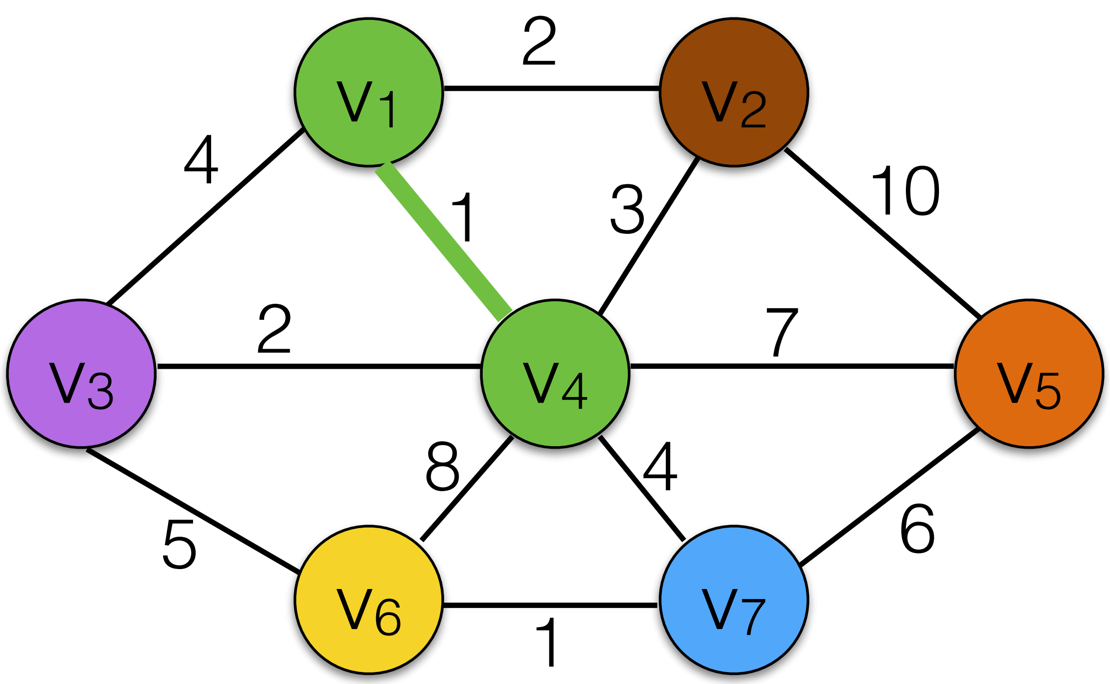
(v1,v2)	2
(v1,v3)	4
(v1,v4)	1
(v2,v4)	3
(v2,v5)	10
(v3,v4)	2
(v3,v6)	5
(v4,v5)	7
(v4,v6)	8
(v4,v7)	4
(v5,v7)	6
(v6,v7)	1

Kruskal's Algorithm



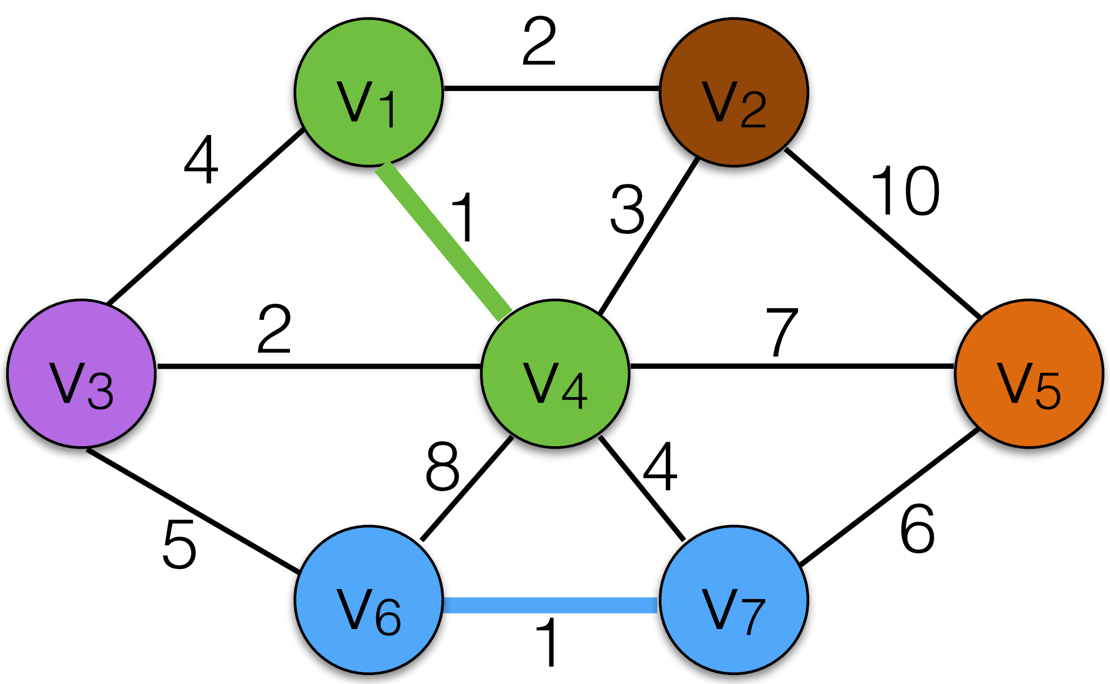
(v1,v4)	1
(v6,v7)	1
(v1,v2)	2
(v3,v4)	2
(v2,v4)	3
(v1,v3)	4
(v4,v7)	4
(v3,v6)	5
(v5,v7)	6
(v4,v5)	7
(v4,v6)	8
(v2,v5)	10

Kruskal's Algorithm



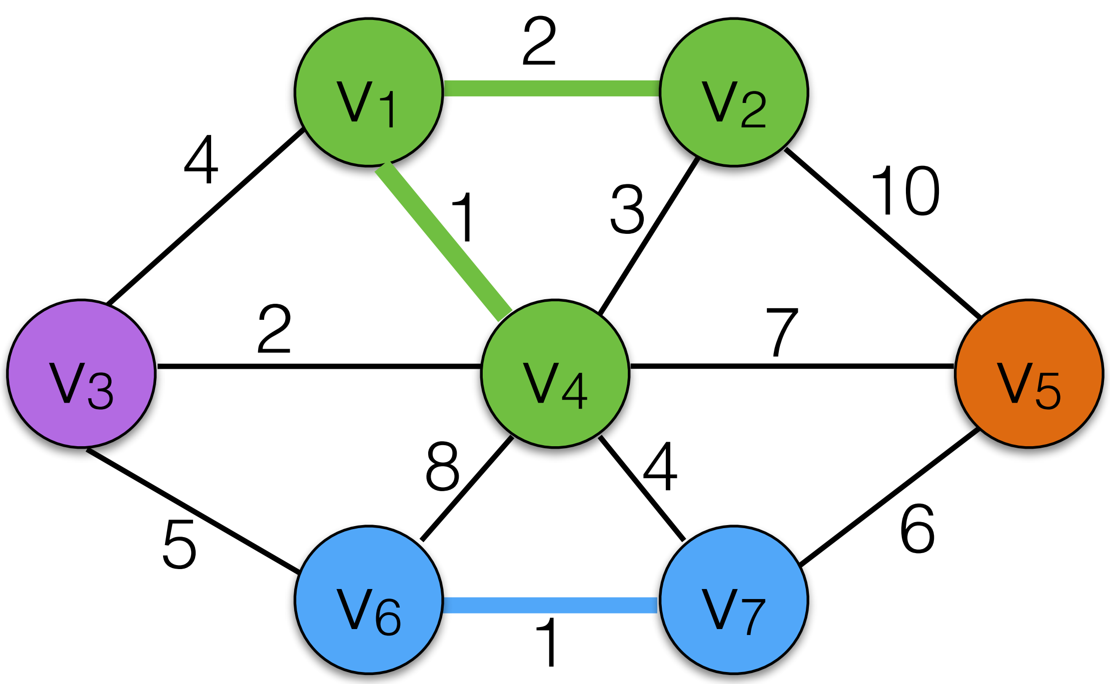
(v1,v4)	1
(v6,v7)	1
(v1,v2)	2
(v3,v4)	2
(v2,v4)	3
(v1,v3)	4
(v4,v7)	4
(v3,v6)	5
(v5,v7)	6
(v4,v5)	7
(v4,v6)	8
(v2,v5)	10

Kruskal's Algorithm



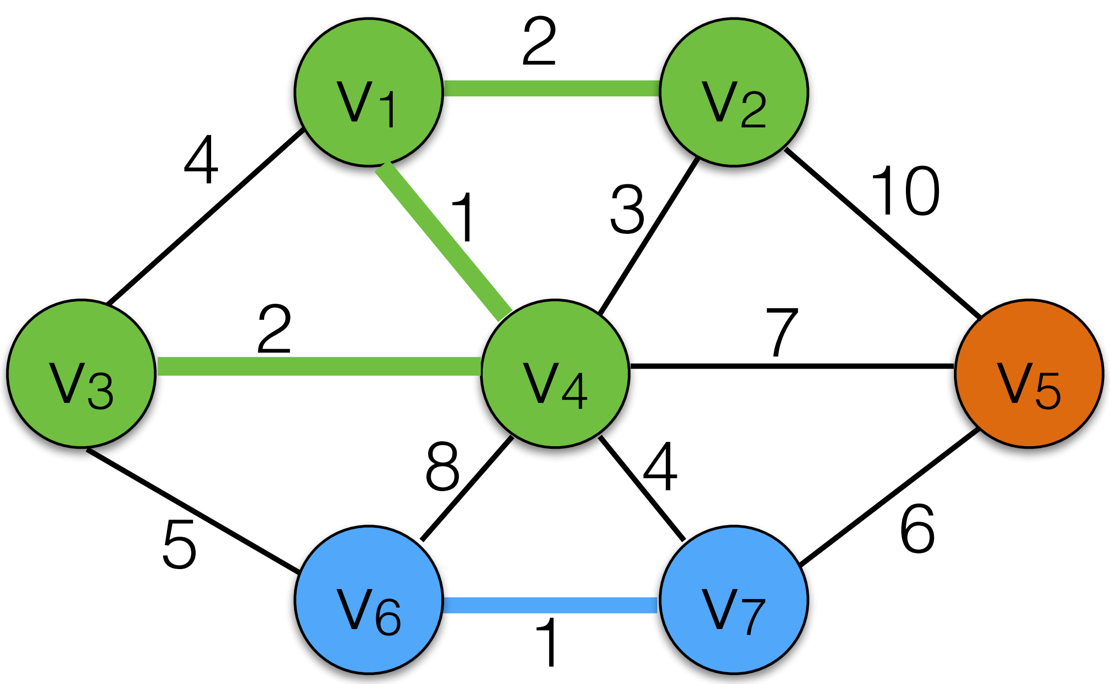
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	
(v3,v4)	2	
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



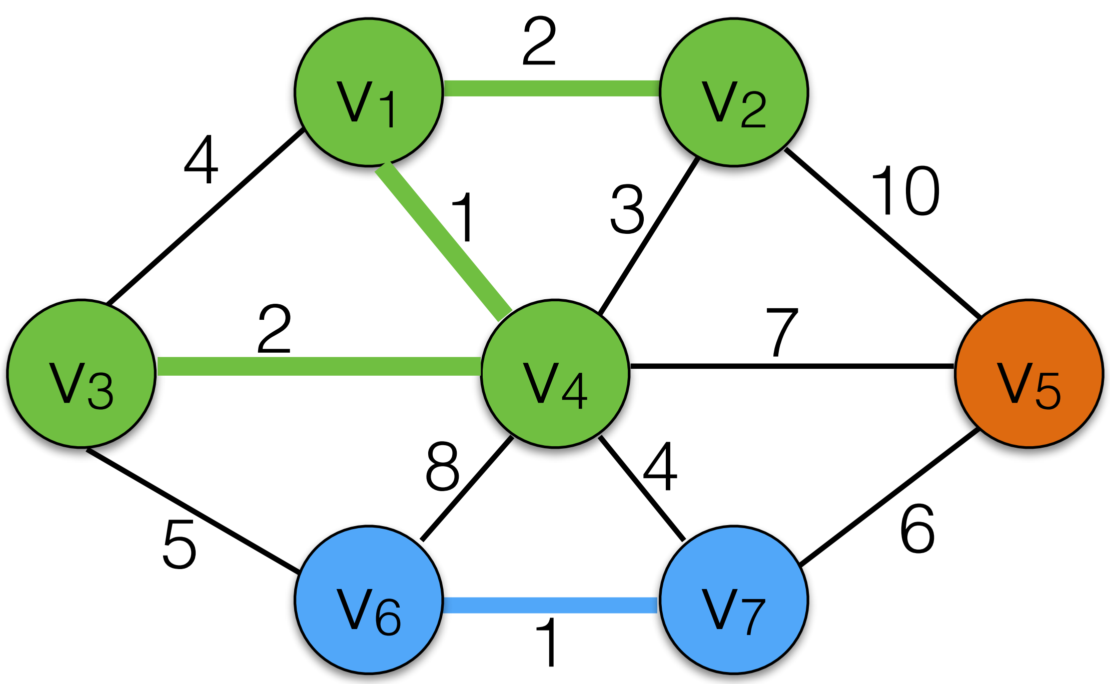
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



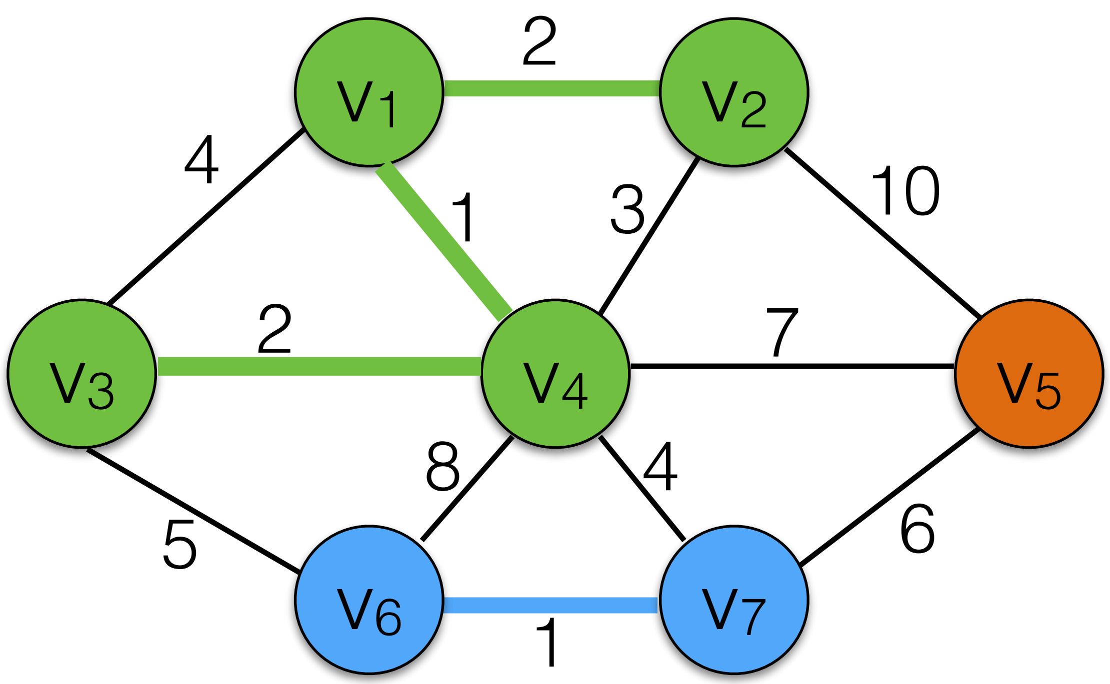
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



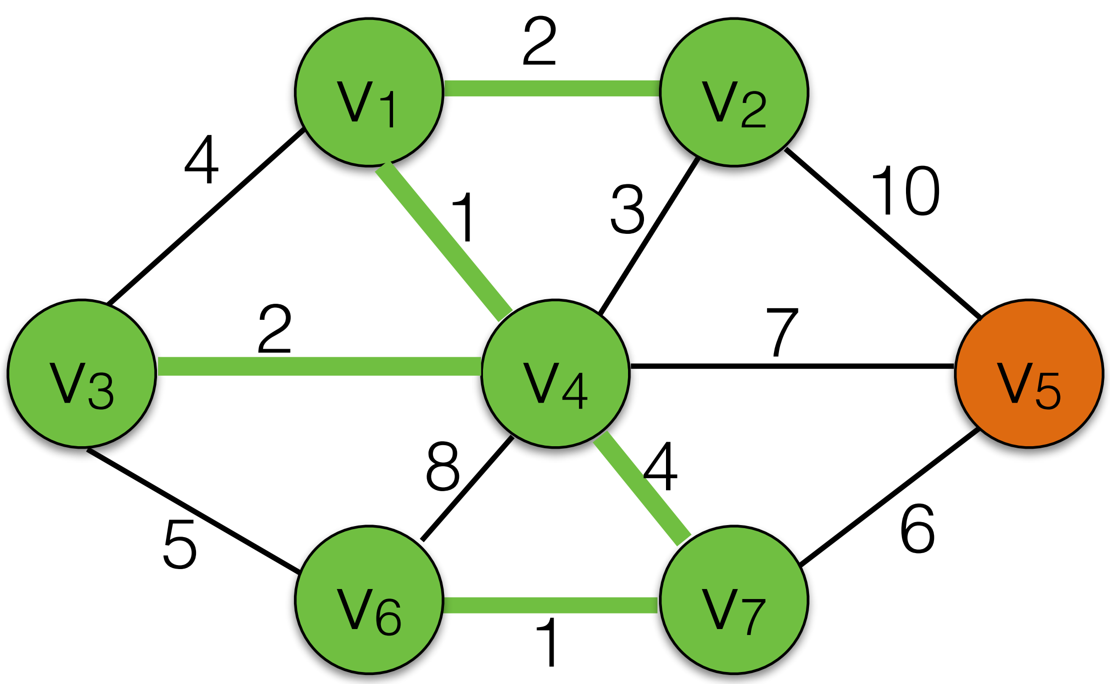
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



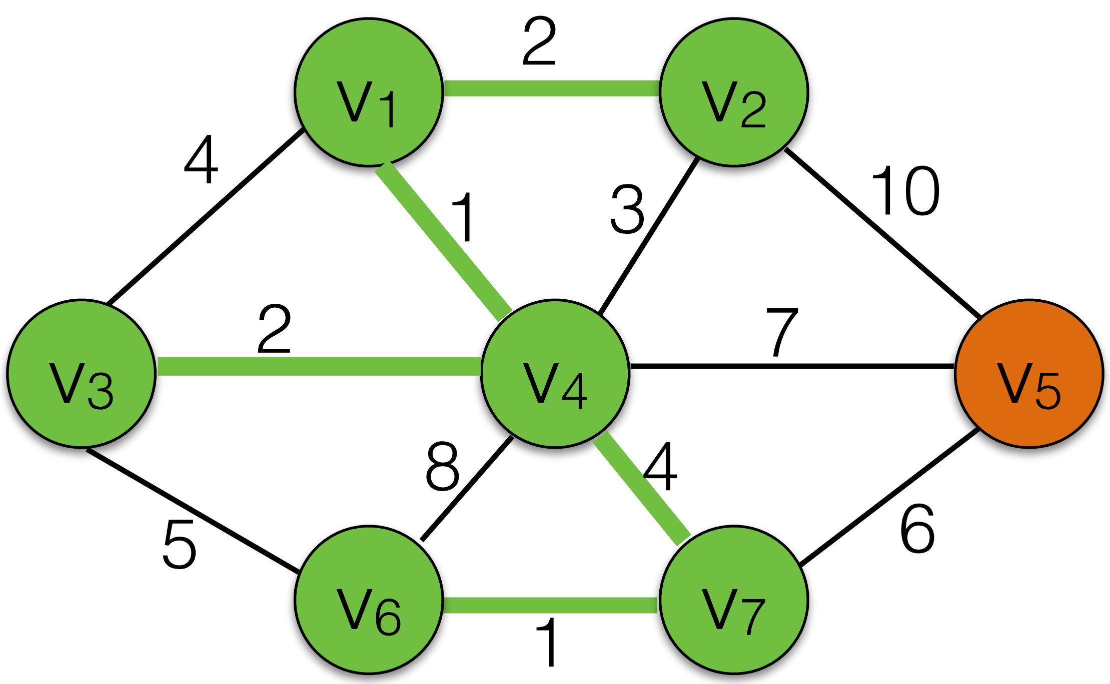
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



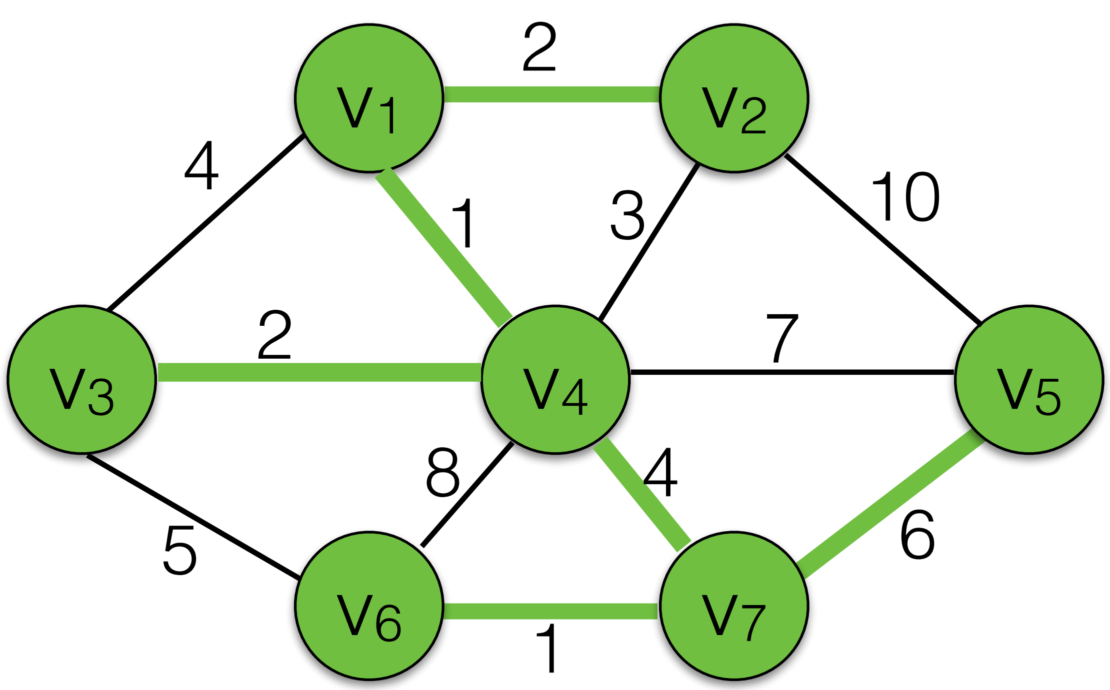
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	OK
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	OK
(v3,v6)	5	reject
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	OK
(v3,v6)	5	reject
(v5,v7)	6	OK
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Implementing Kruskal's Algorithm

- Try to add edges one-by-one in increasing order. Build a heap in O(|E|). Each deleteMin takes O(log |E|)
- How to maintain the forest?
  - Represent each tree in the forest as a set of vertices in the tree.
  - When adding an edge, check if both vertices are in the same set (*find*). If not, take the *union* of the two sets.
  - This can be done efficiently using a *disjoint set* data structure.

Implementing Kruskal's Algorithm

- Try to add edges one-by-one in increasing order. Build a heap in O(|E|). Each deleteMin takes O(log |E|)

- How to maintain the forest?

  - Represent each tree in the forest as a set of vertices in the tree.

  - When adding an edge, check if both vertices are in the same set (*find*). If not, take the *union* of the two sets.

  - This can be done efficiently using a *disjoint set* data structure.

Total turns out to be: O(|E| log |V|)

Application: Hierarchical Clustering

- This is a very common data analysis problem.
- Group together data items based on similarity (defined over some feature set).
- Discover classes and class relationships.

Zoo Data Set

101 animals

represent each data item as a vector of integers (15 attributes).

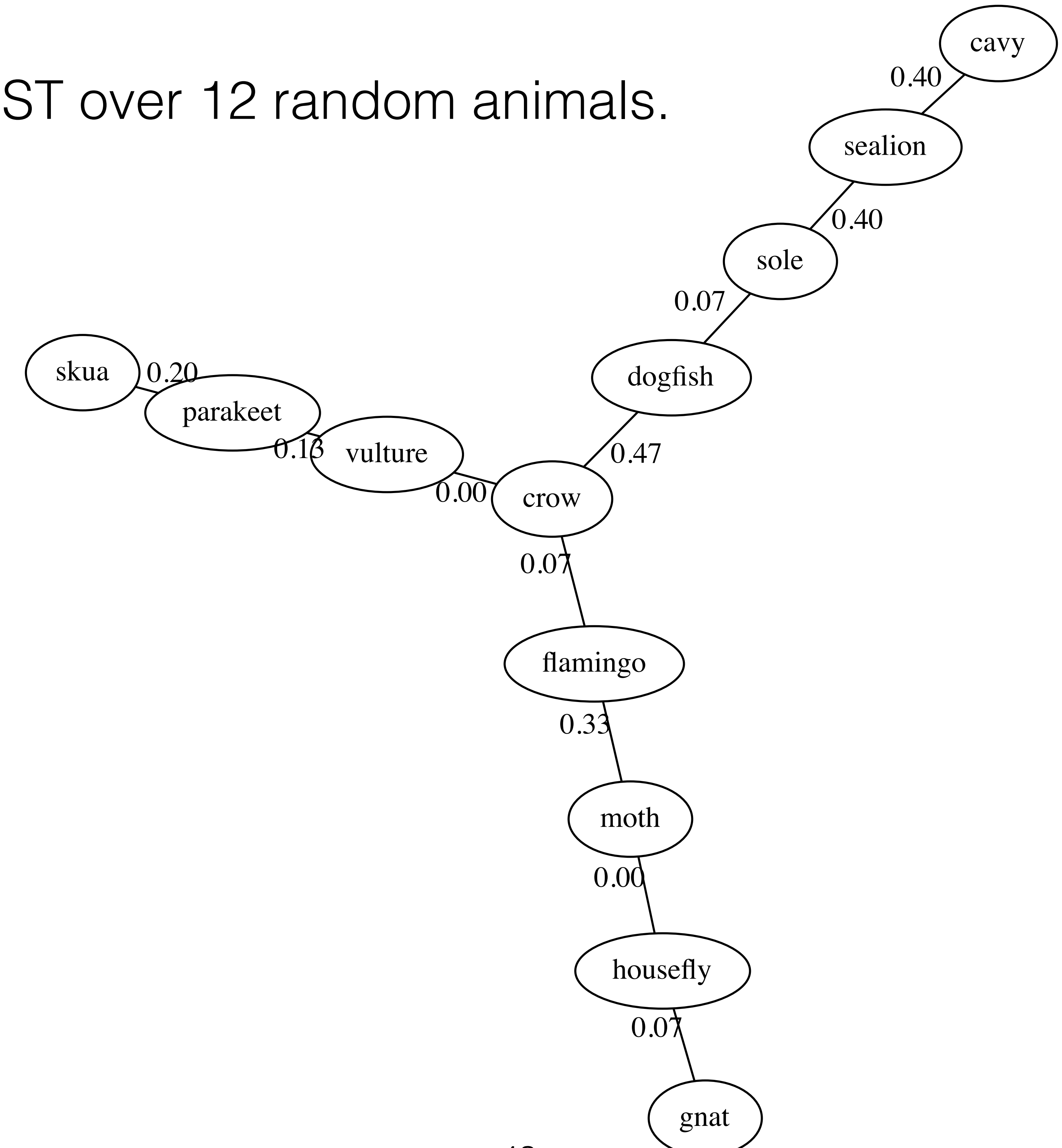
	bear	chicke	tortoise	flea
hair	1	0	0	0
feathers	0	1	0	0
eggs	0	1	1	1
milk	1	0	0	0
airborne	0	1	0	0
aquatic	0	0	0	0
predator	1	0	0	0
toothed	1	0	0	0
backbone	1	1	1	0
breathes	1	1	1	1
venomou	0	0	0	0
fins	0	0	0	0
legs	4	2	4	6
tail	1	1	1	0
domestic	0	1	0	0

. . .

https://archive.ics.uci.edu/ml/datasets/Zoo

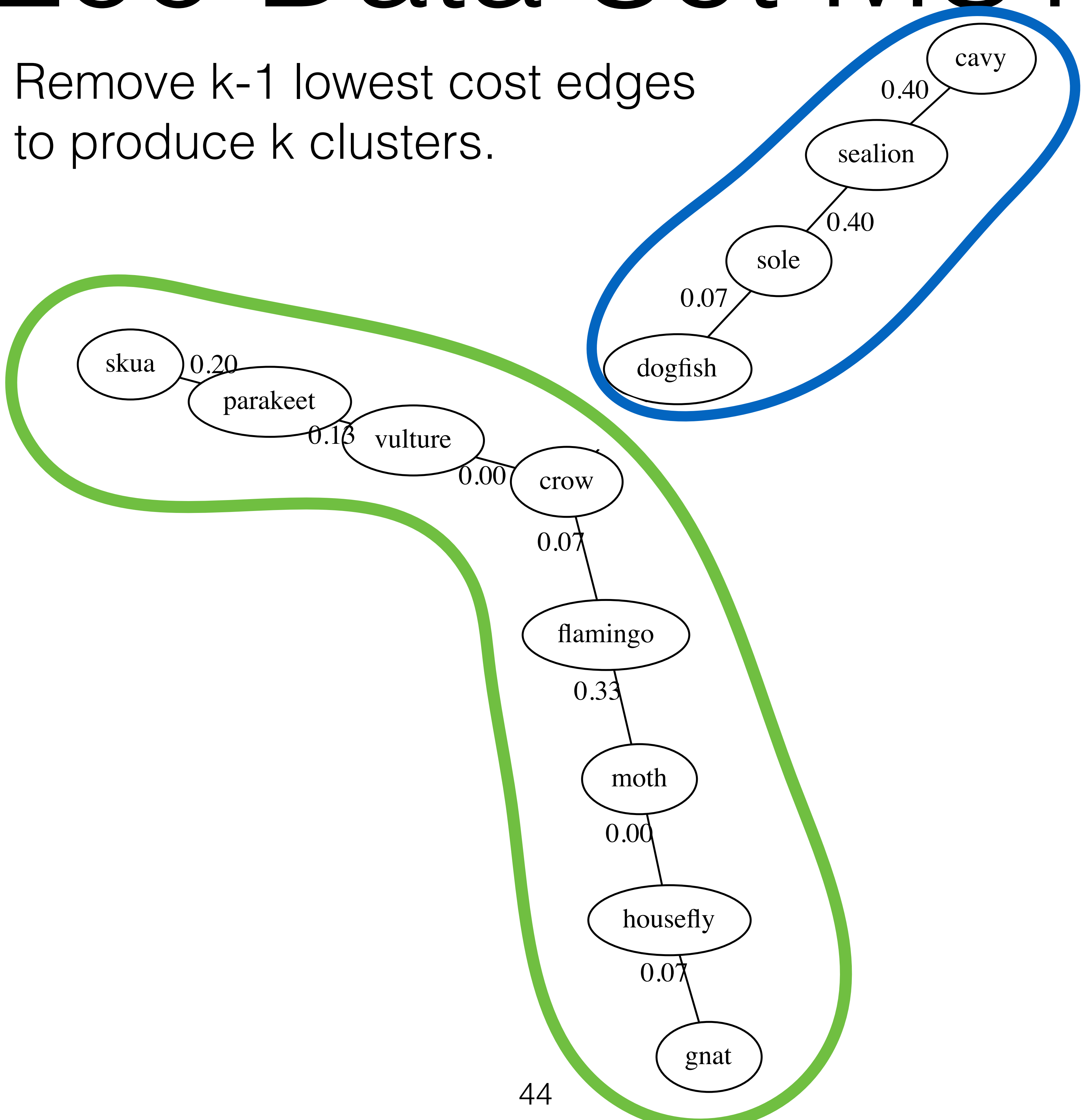
Zoo Data Set MST

- MST over 12 random animals.



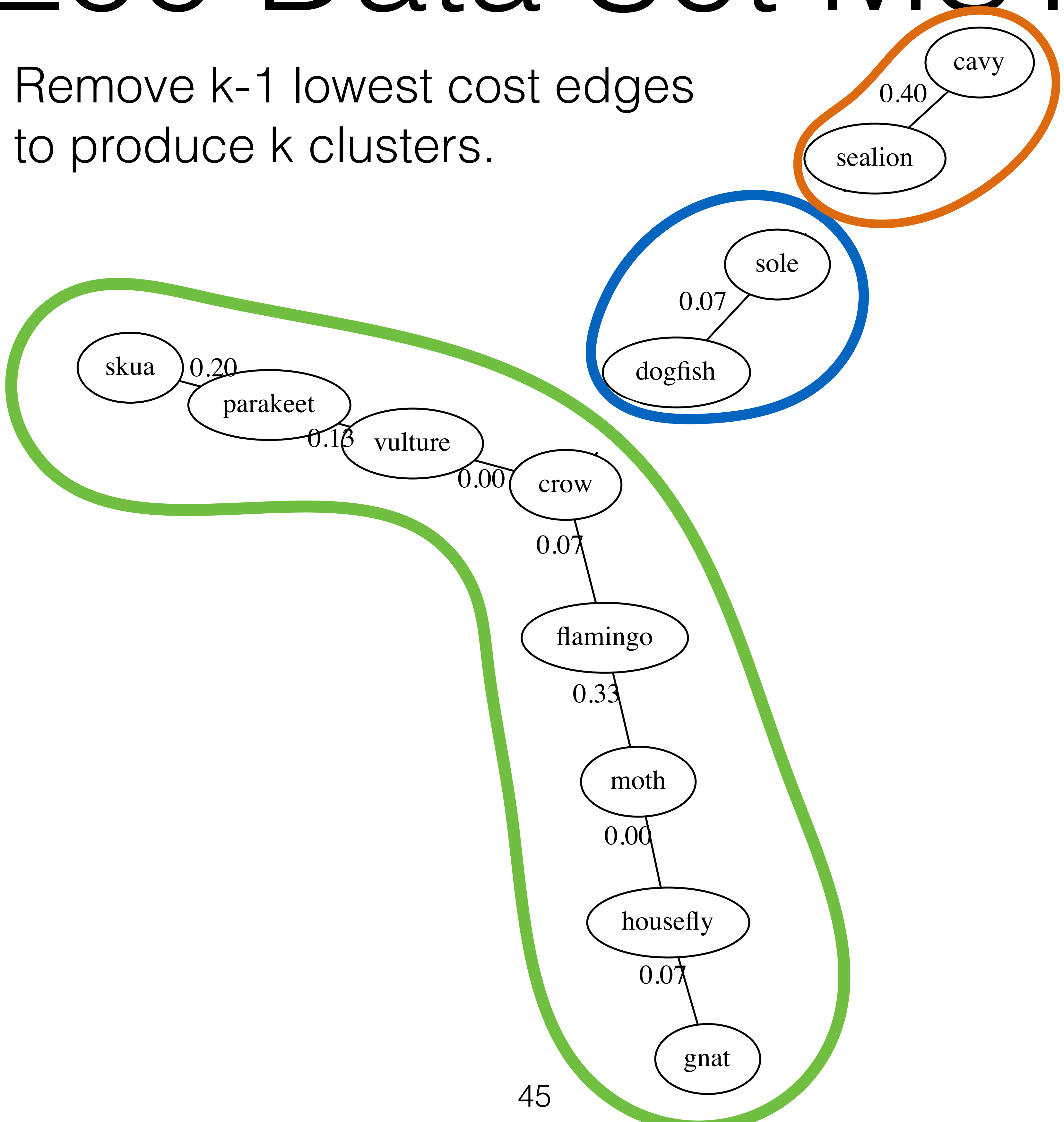
Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.



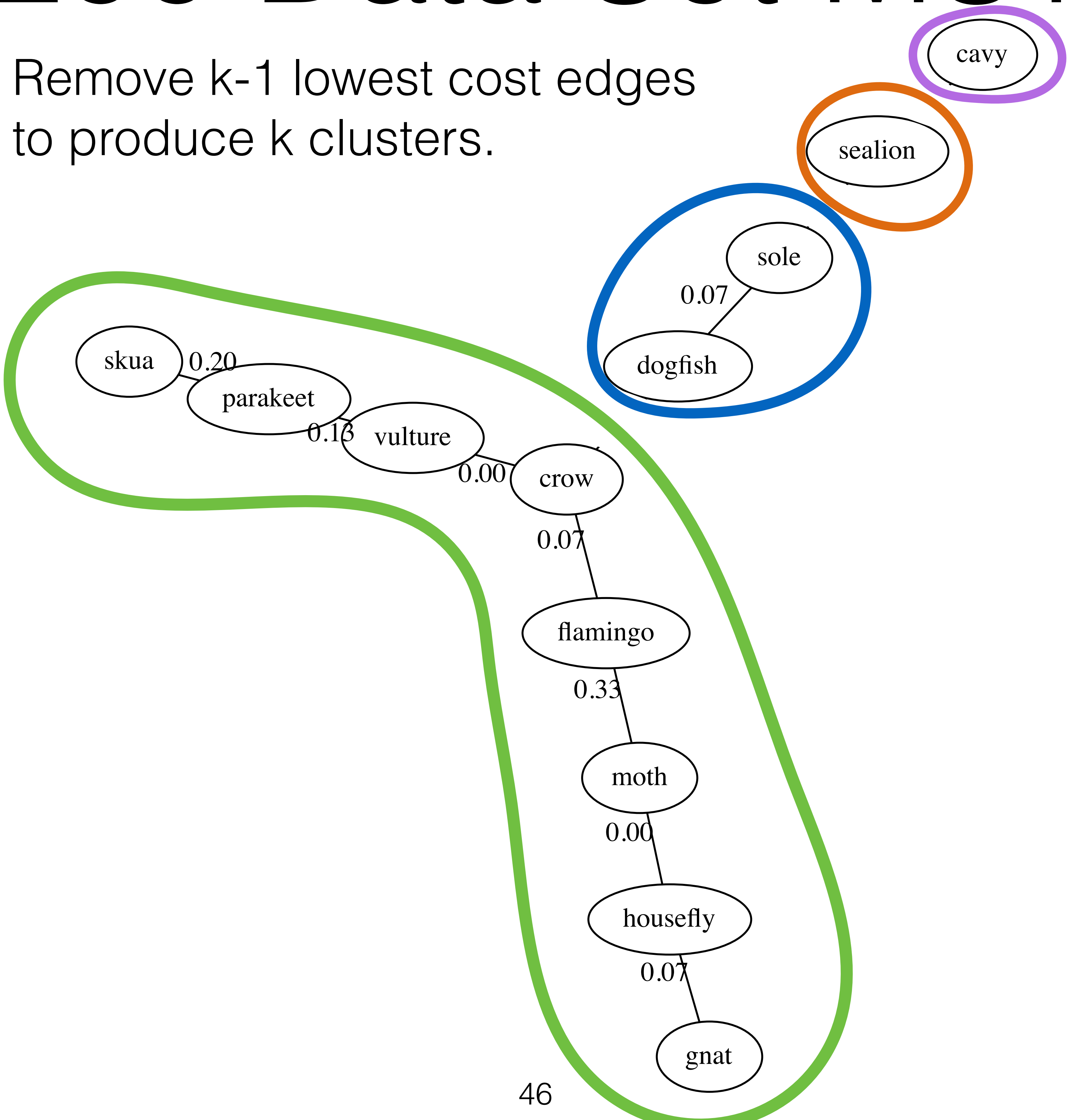
Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.



Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.



Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.