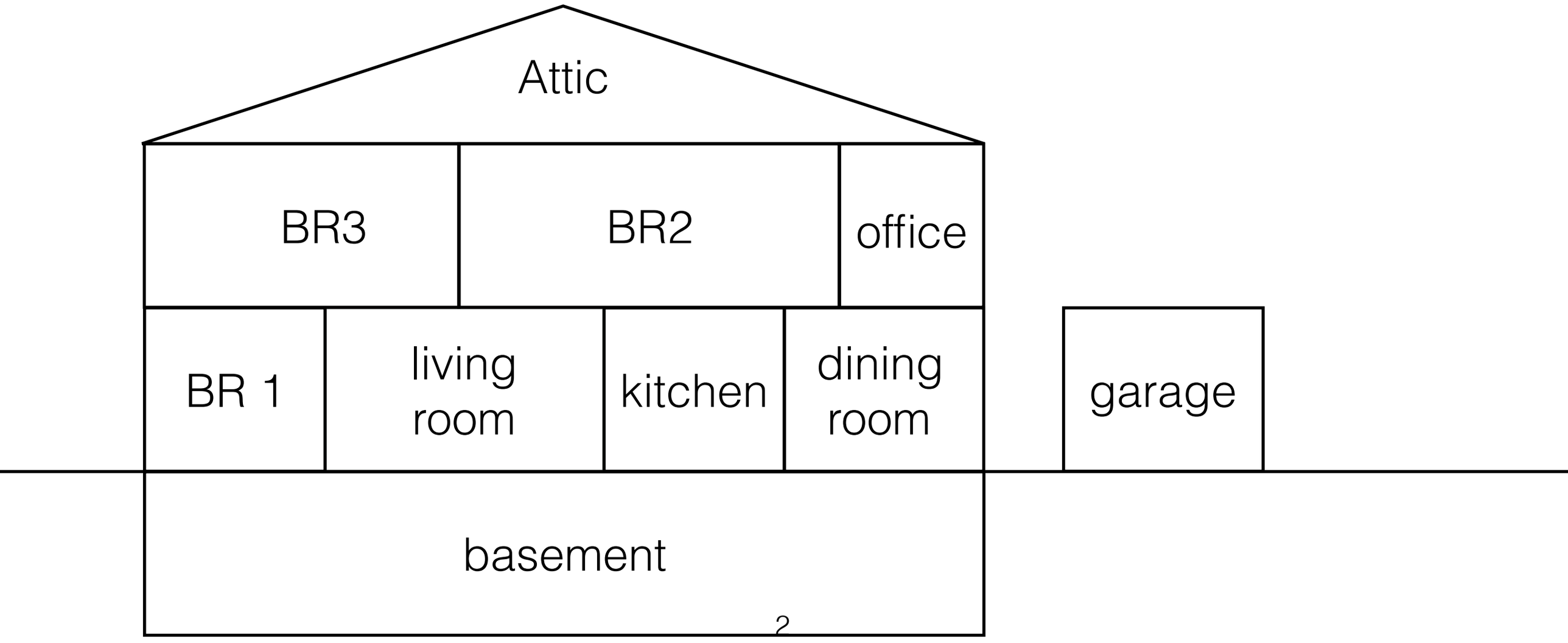
Data Structures in Java

Lecture 20: Minimum Spanning Trees.
Bi-connectivity. Euler circuits.

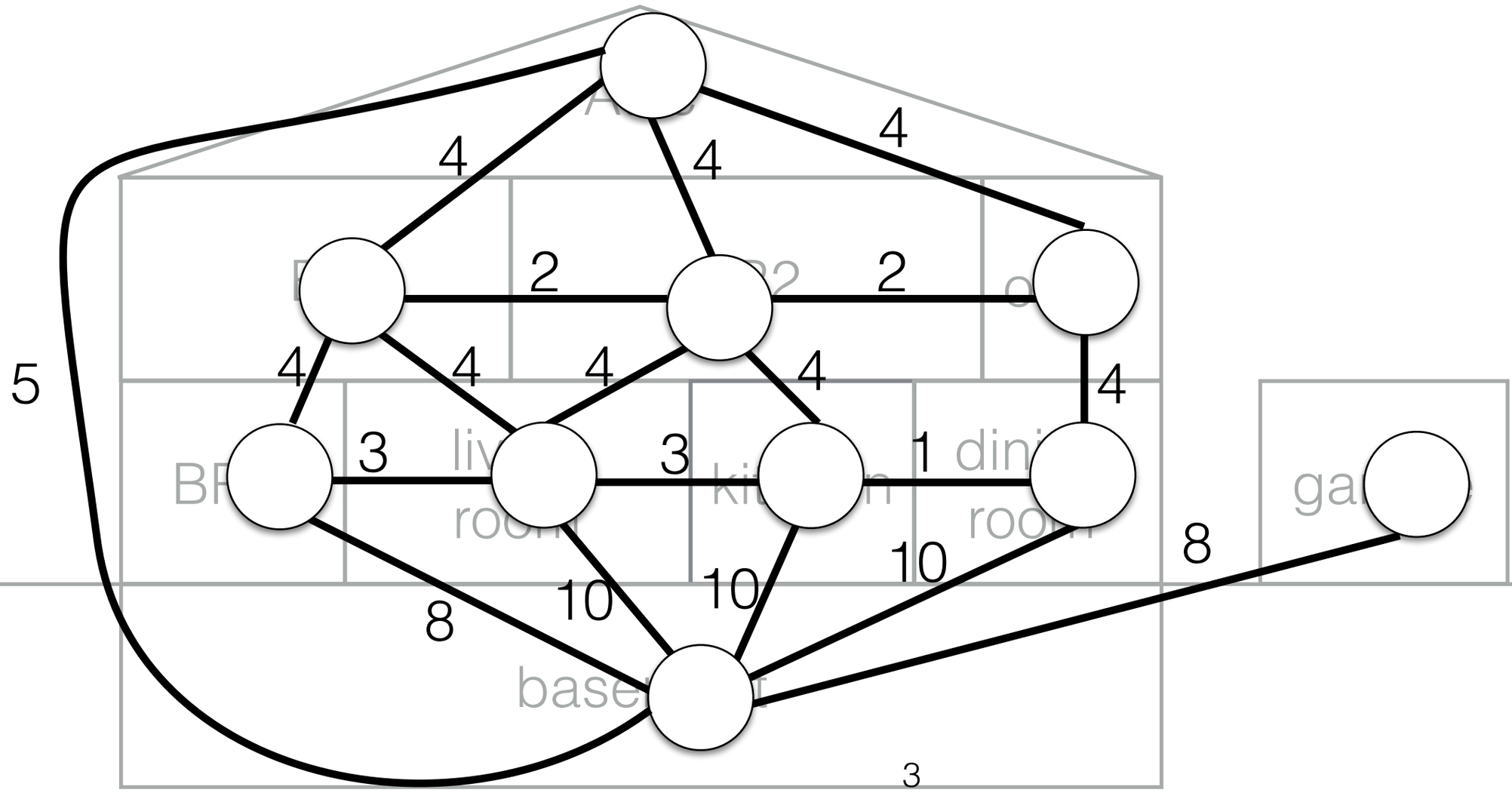
11/25/2019

Daniel Bauer

Designing a Home Network.

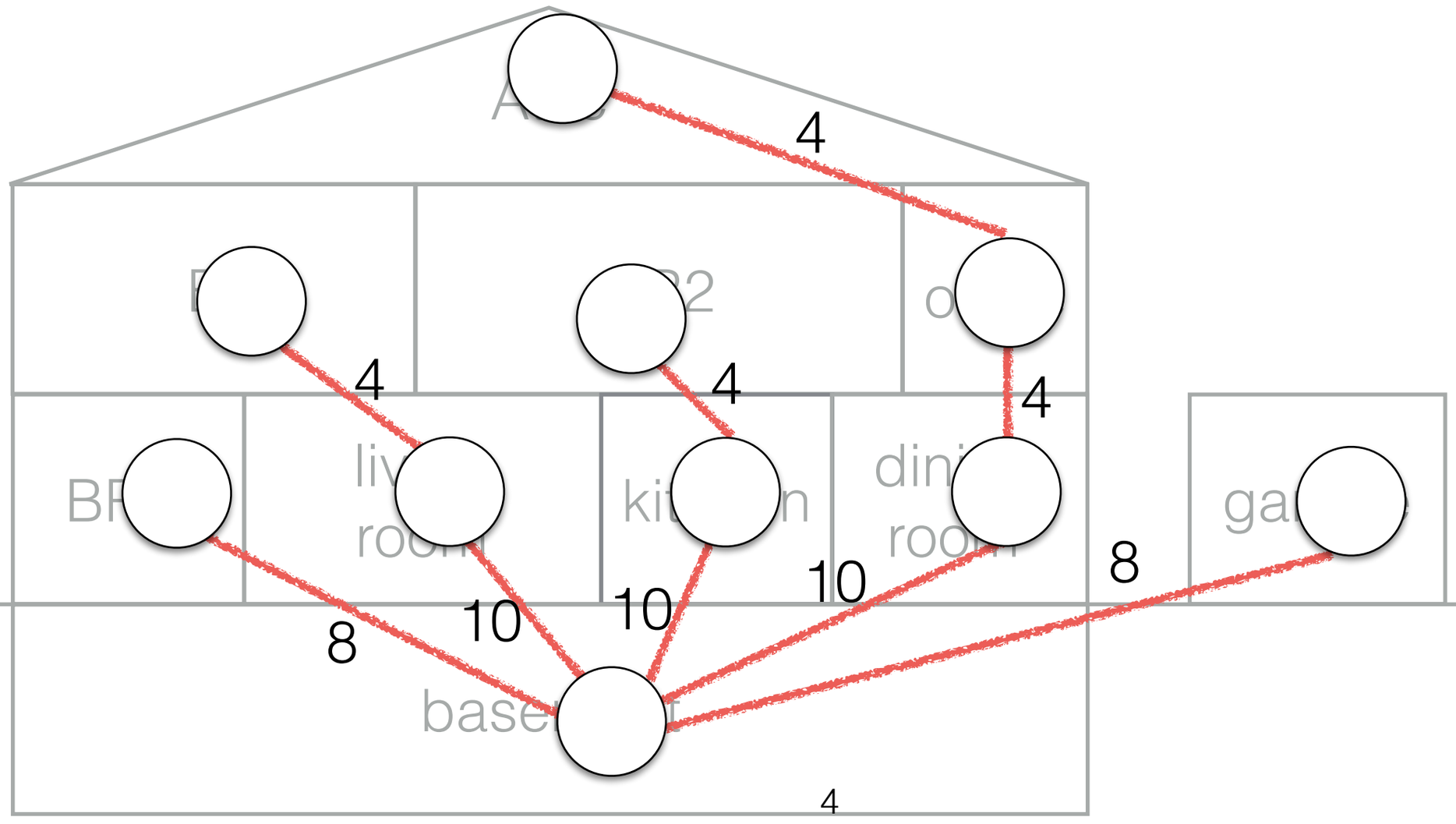


Designing a Home Network.



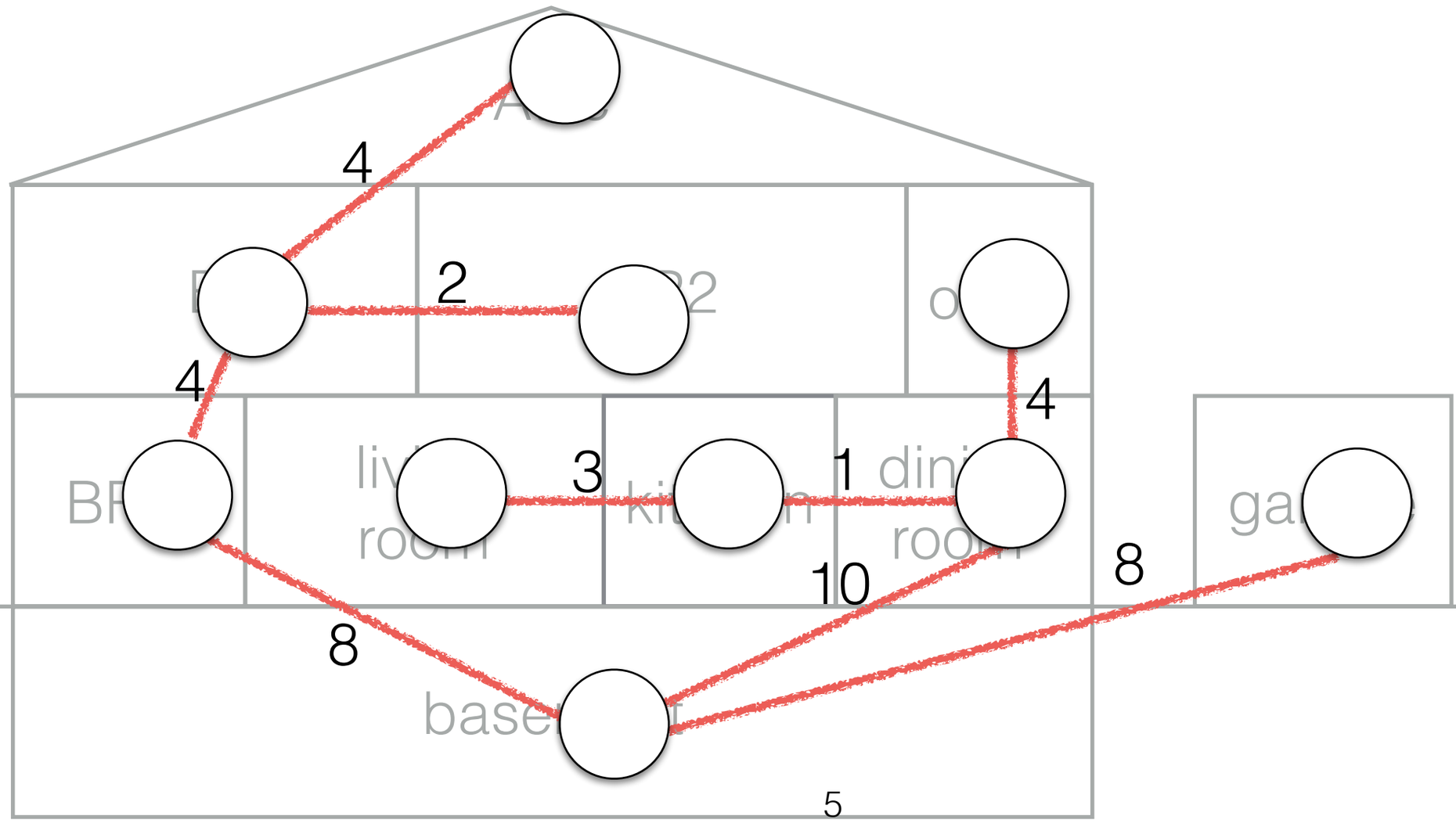
Designing a Home Network.

Total cost: 62



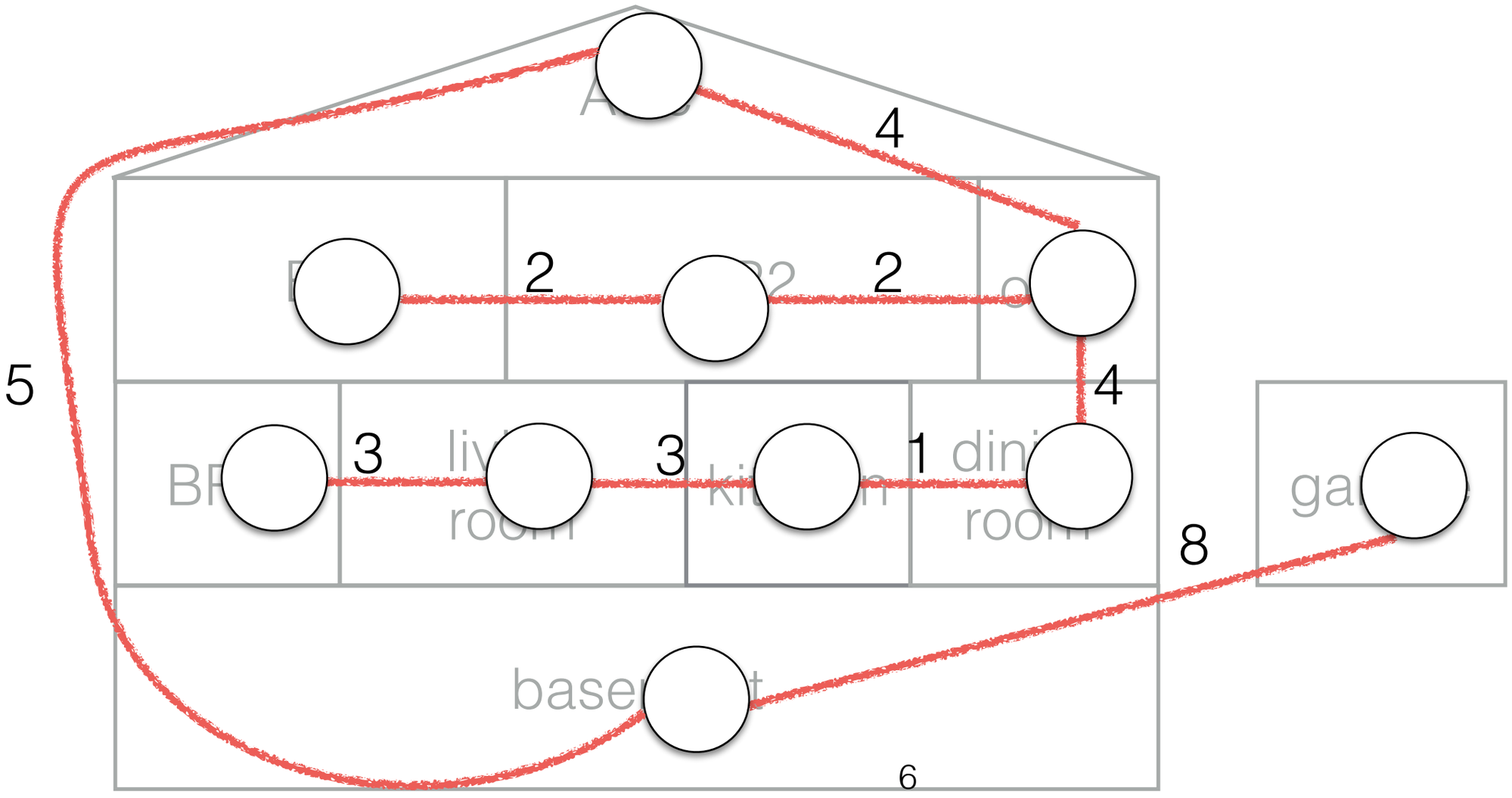
Designing a Home Network.

Total cost: 44



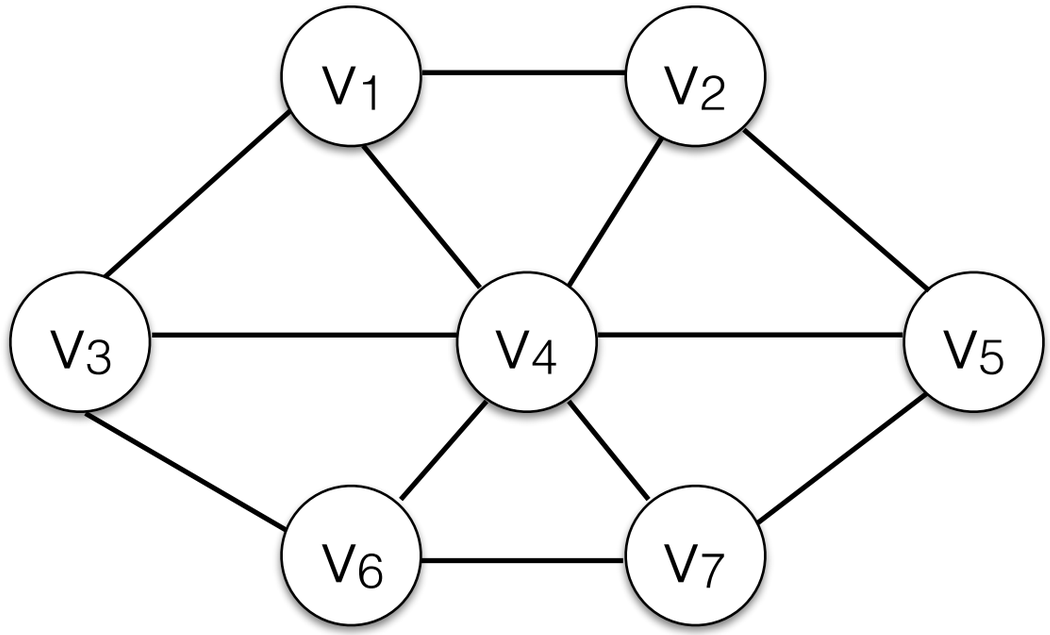
Designing a Home Network.

Total cost: 32



Spanning Trees

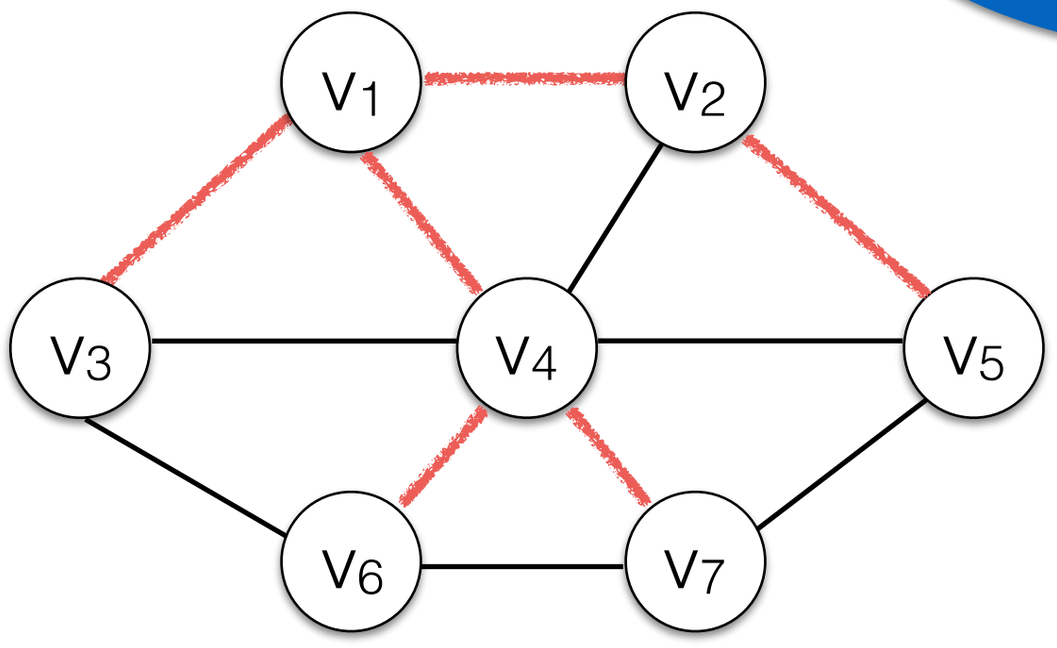
- Given an undirected, connected graph G=(V,E).
- A ***spanning tree*** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$



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, $E_T \subseteq E$)

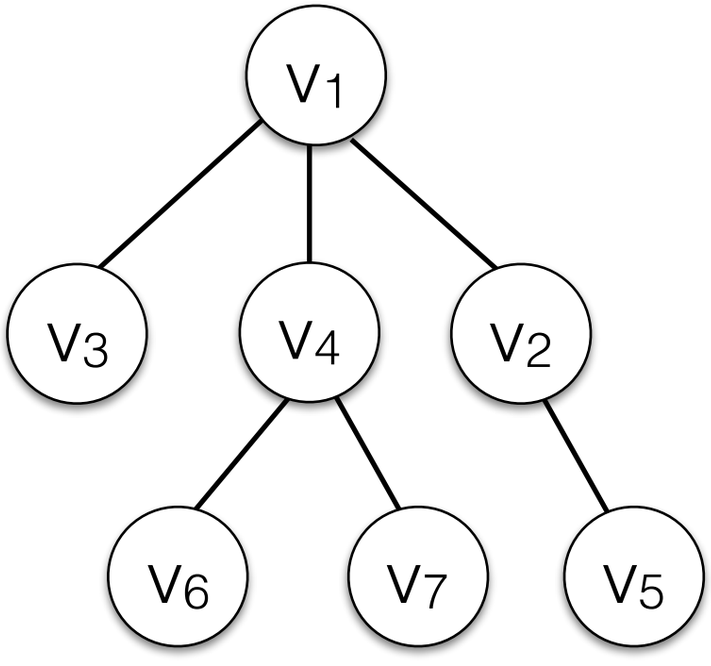
T is acyclic. There is a single path between any pair of vertices.

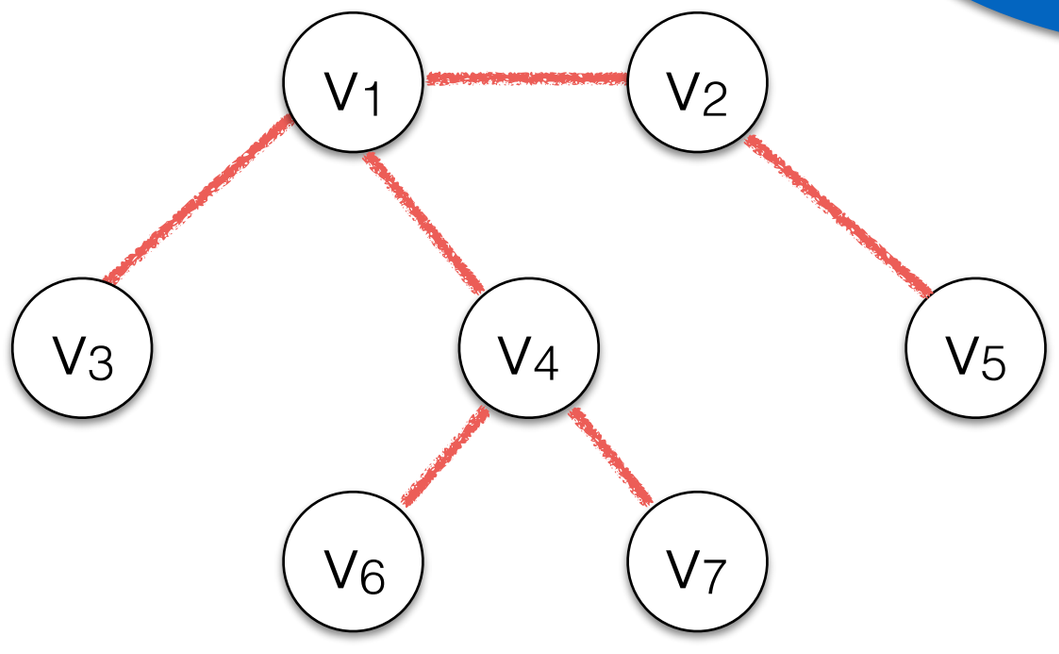


Spanning Trees

- 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.

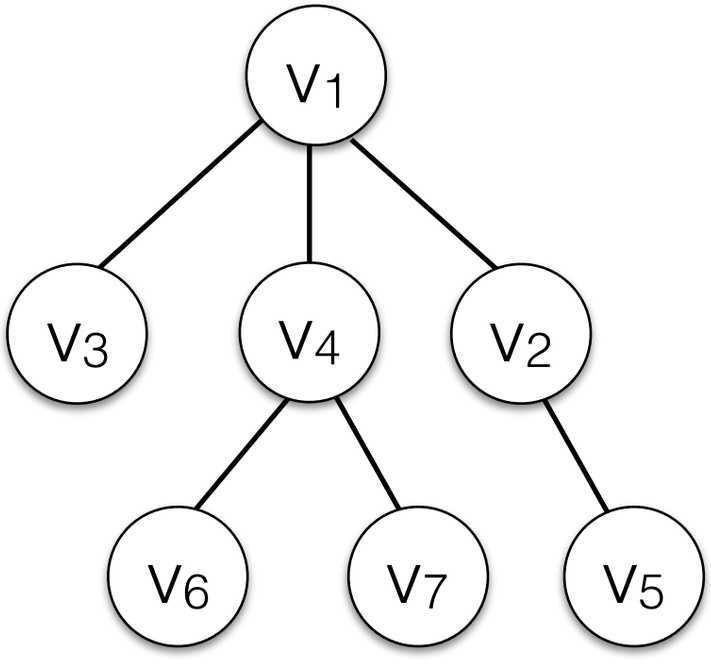


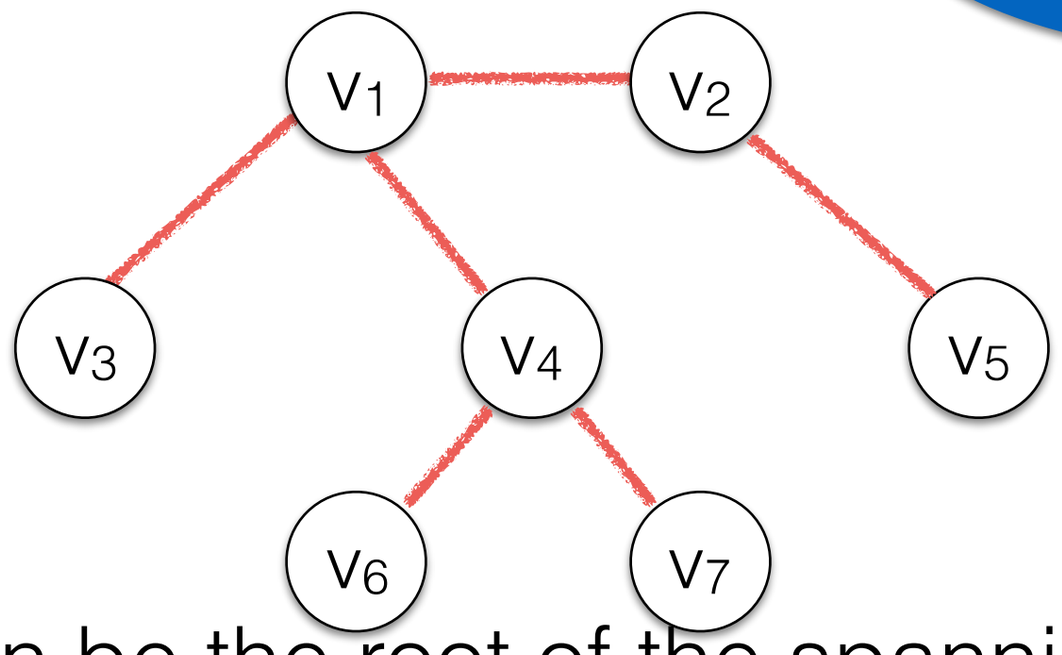


Spanning Trees

- 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.

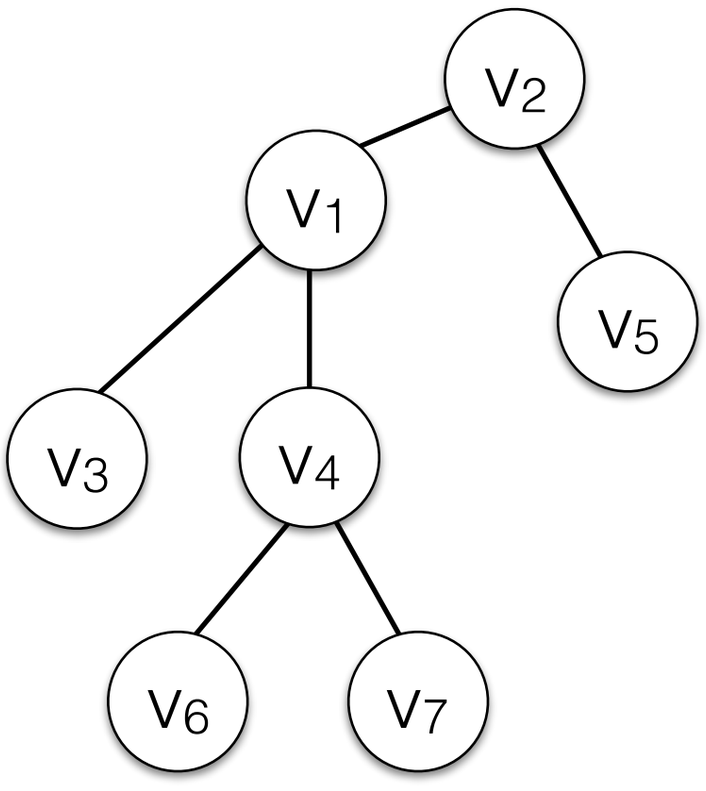


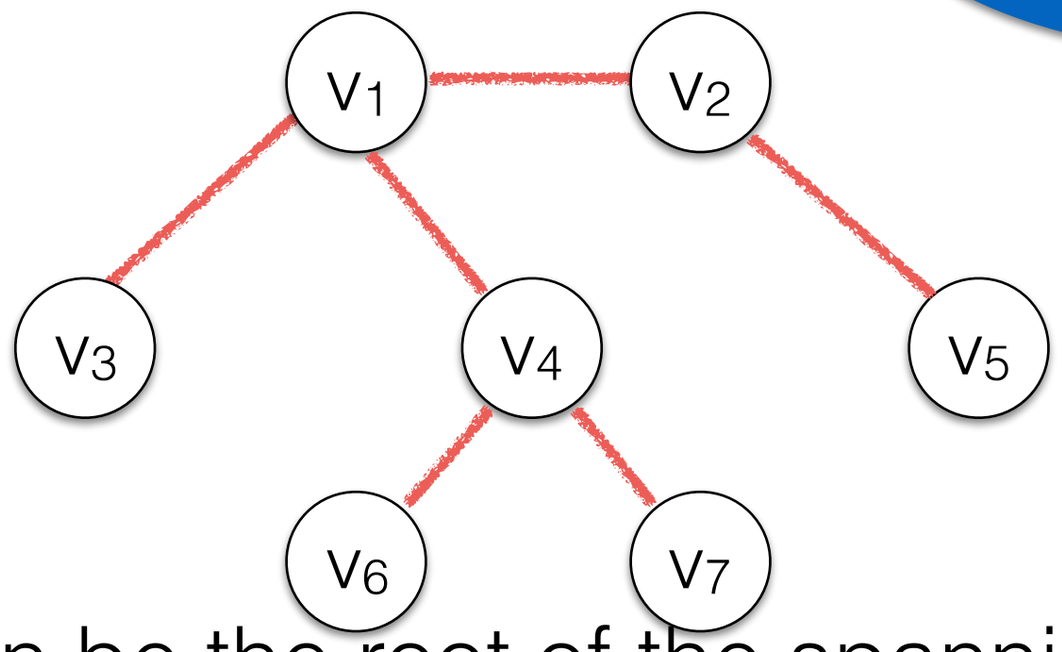


Any node can be the root of the spanning tree.

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, $E_T \subseteq E$)



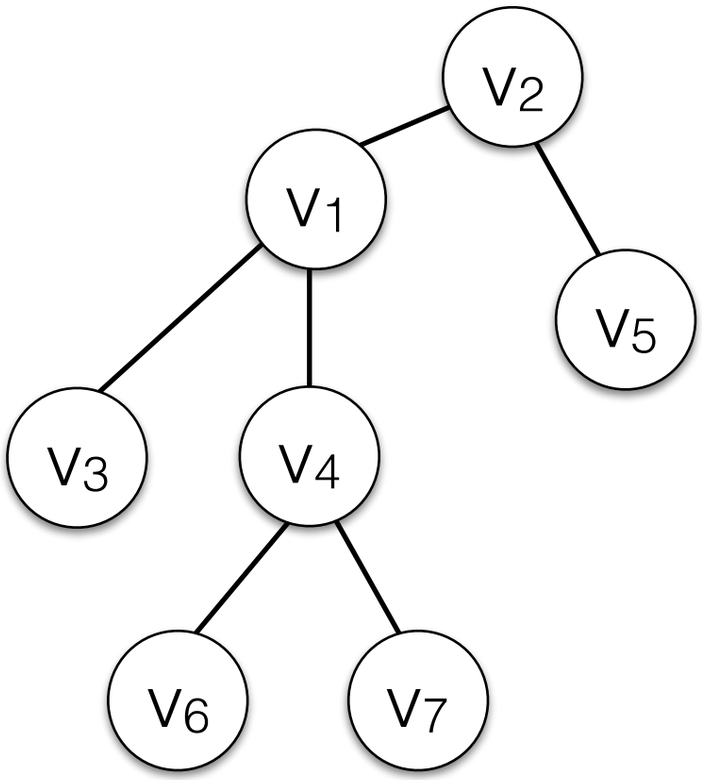


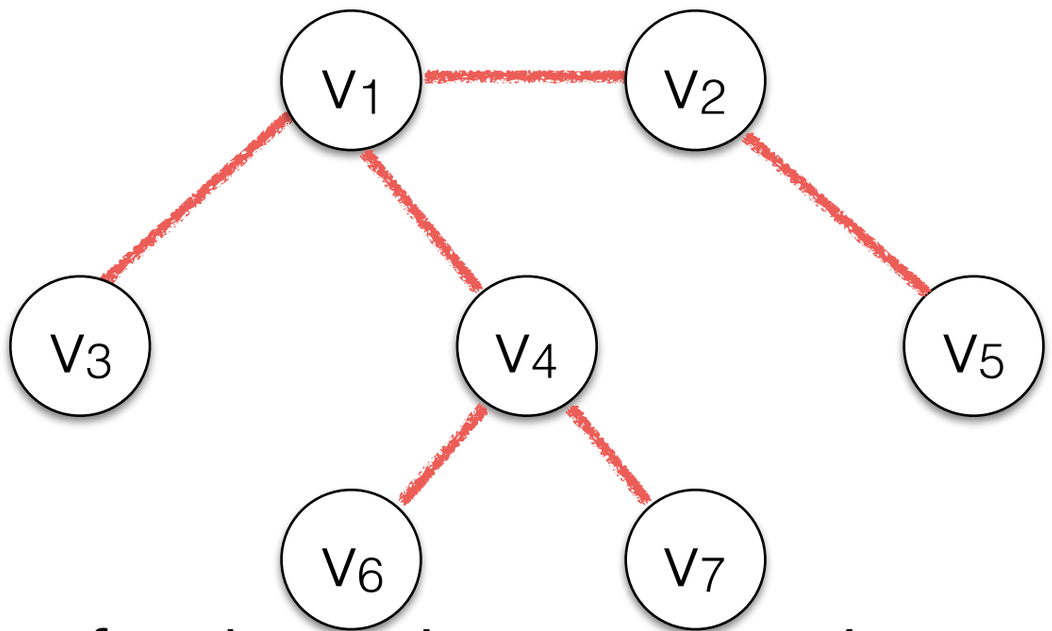
T is acyclic. There is a single path between any pair of vertices.

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, $E_T \subseteq E$)





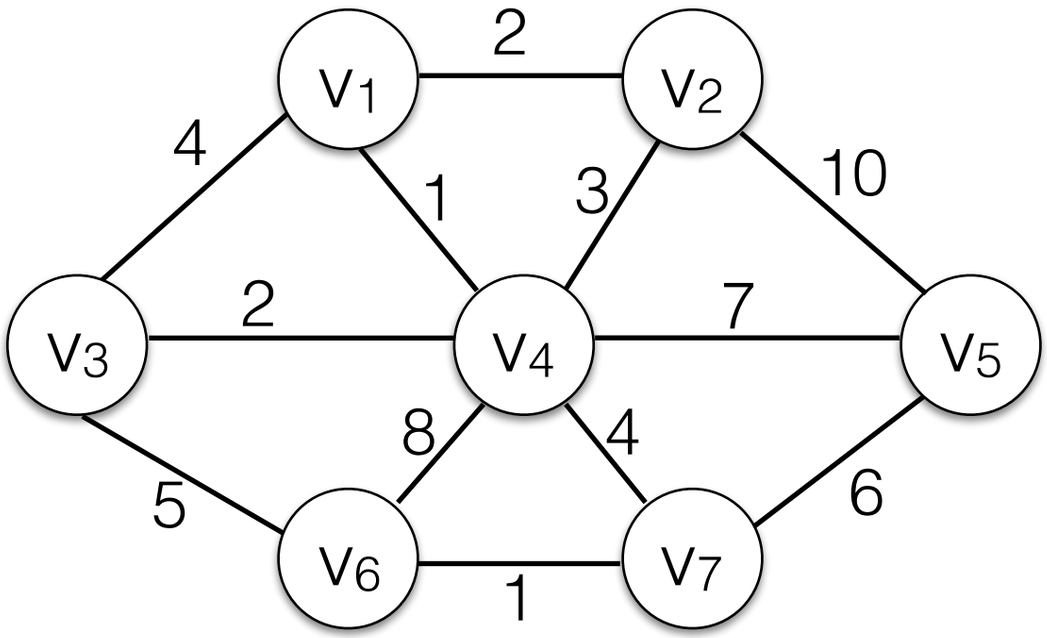
Number of edges in a spanning tree: |V|-1

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

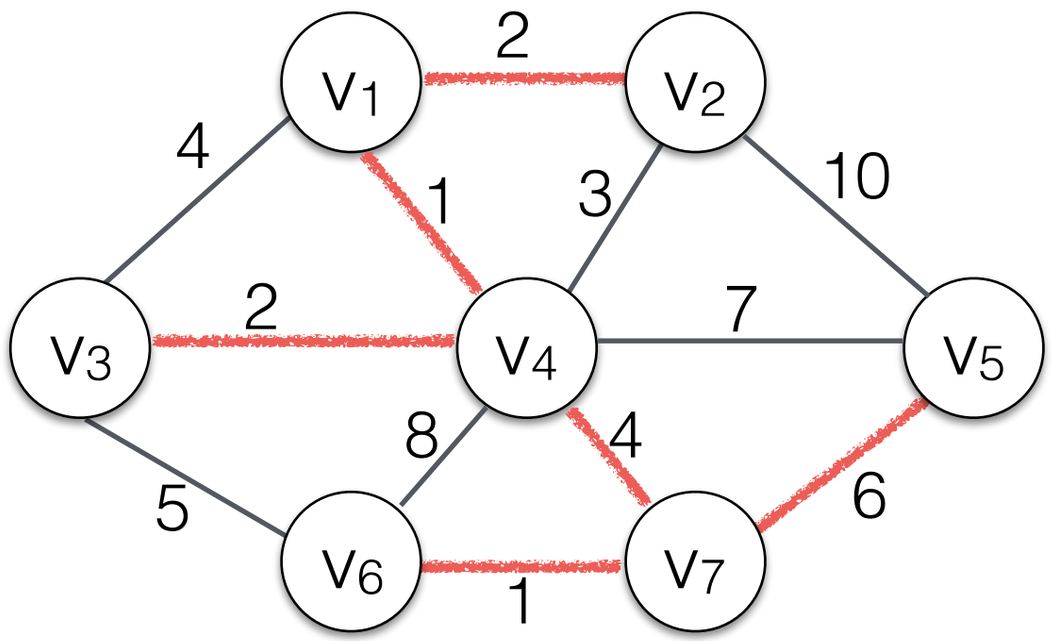
- Given a *weighted* undirected graph G=(E,V).
- A ***minimum spanning tree*** is a spanning tree with the minimum sum of edge weights.



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

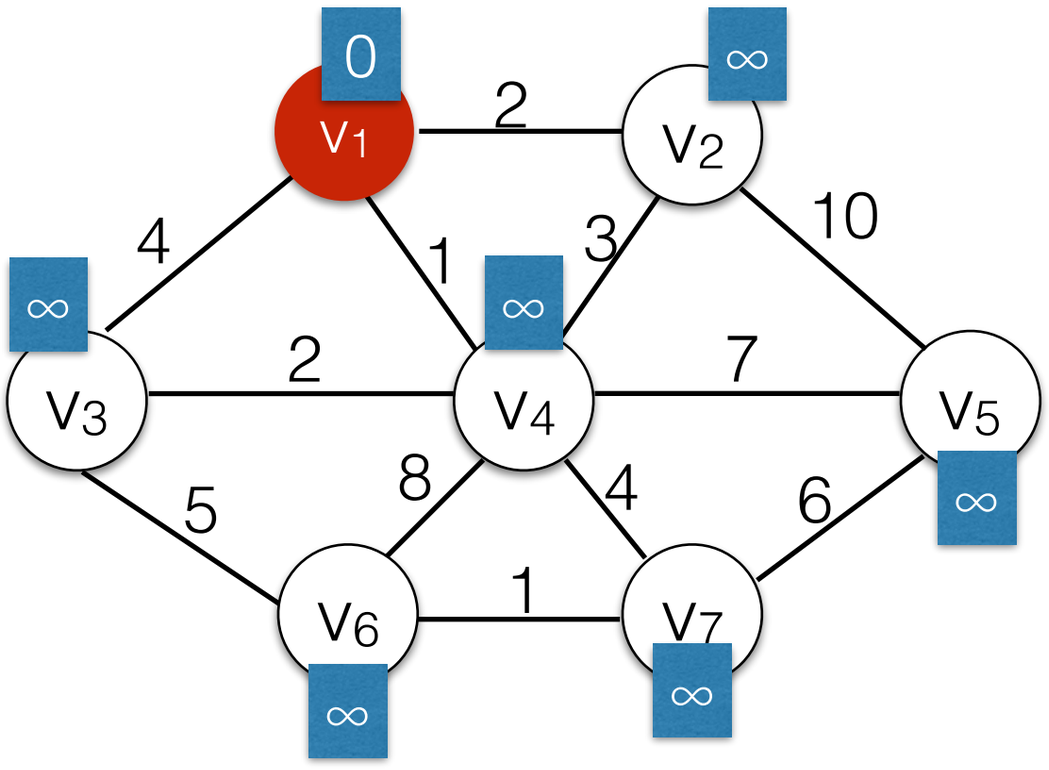


(often there are multiple minimum spanning trees)

Prim's Algorithm for finding MSTs

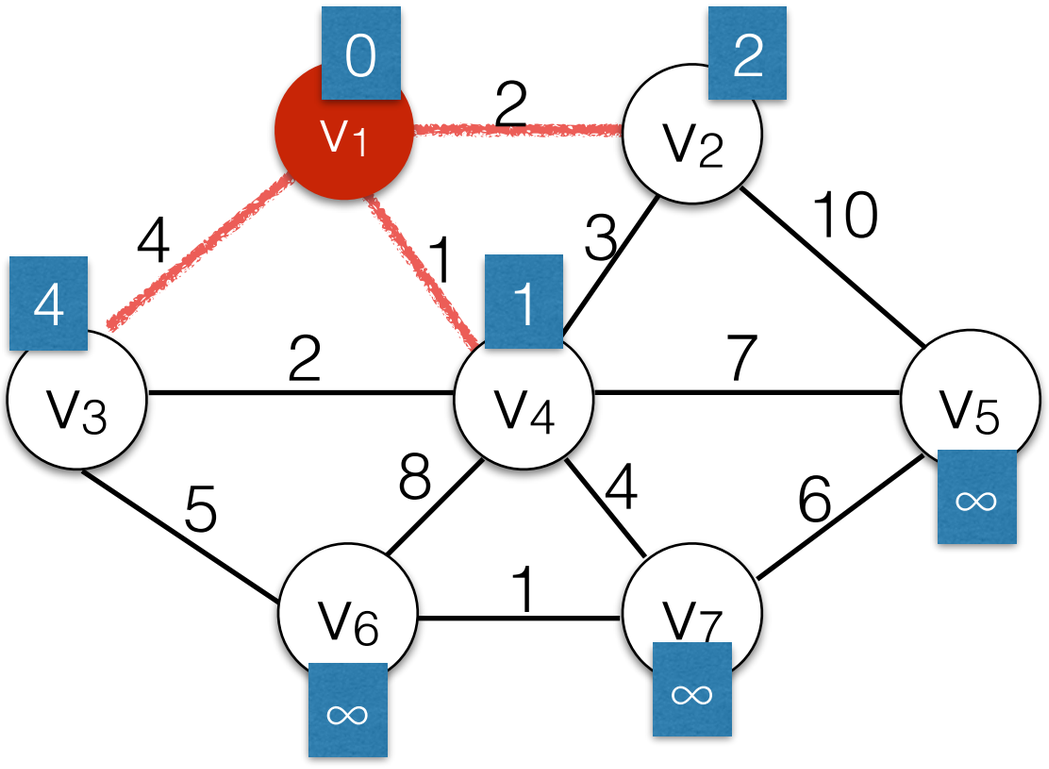
- Another greedy algorithm. A variant of Dijkstra's algorithm.
- Cost annotations for each vertex v reflect the lowest weight of an edge connecting v to other *vertices already visited.*
  - That means there might be a lower-weight edge from another vertices that have not been seen yet.
- Keep vertices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

Prim's Algorithm



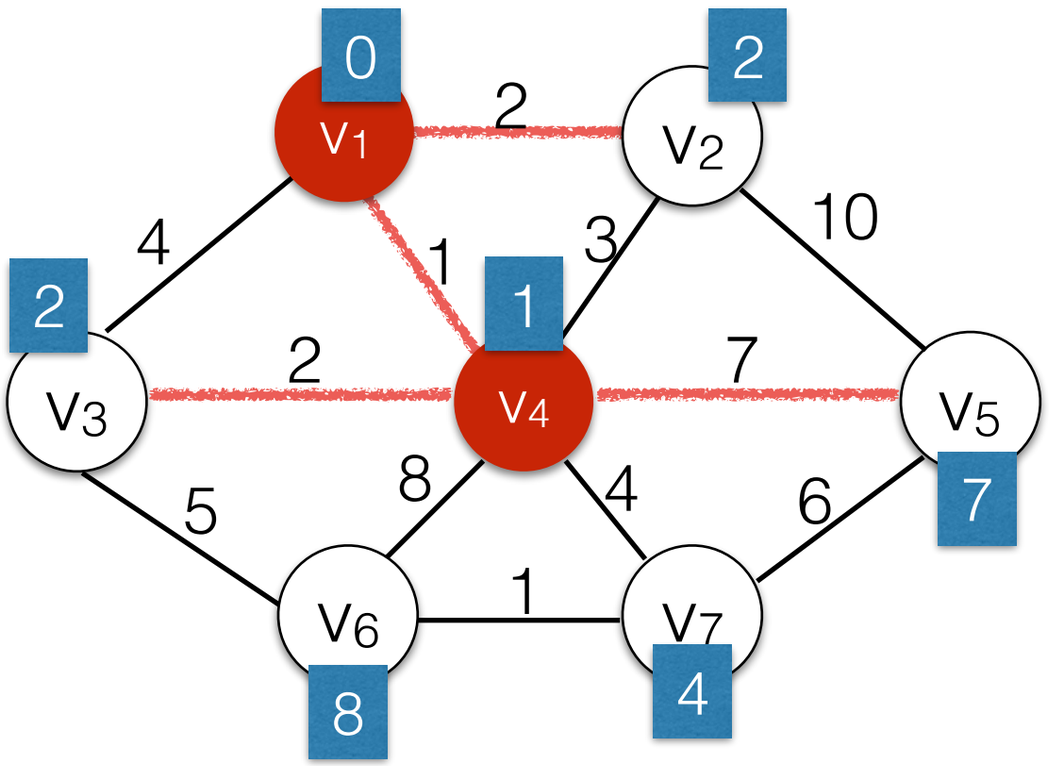

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



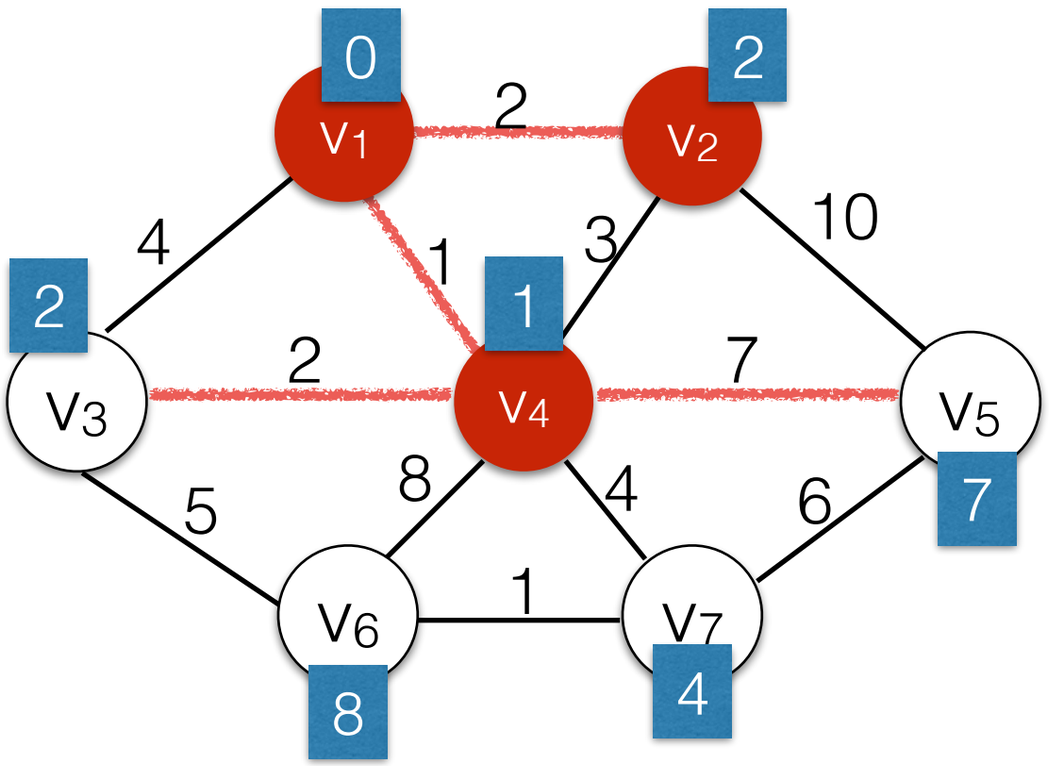

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



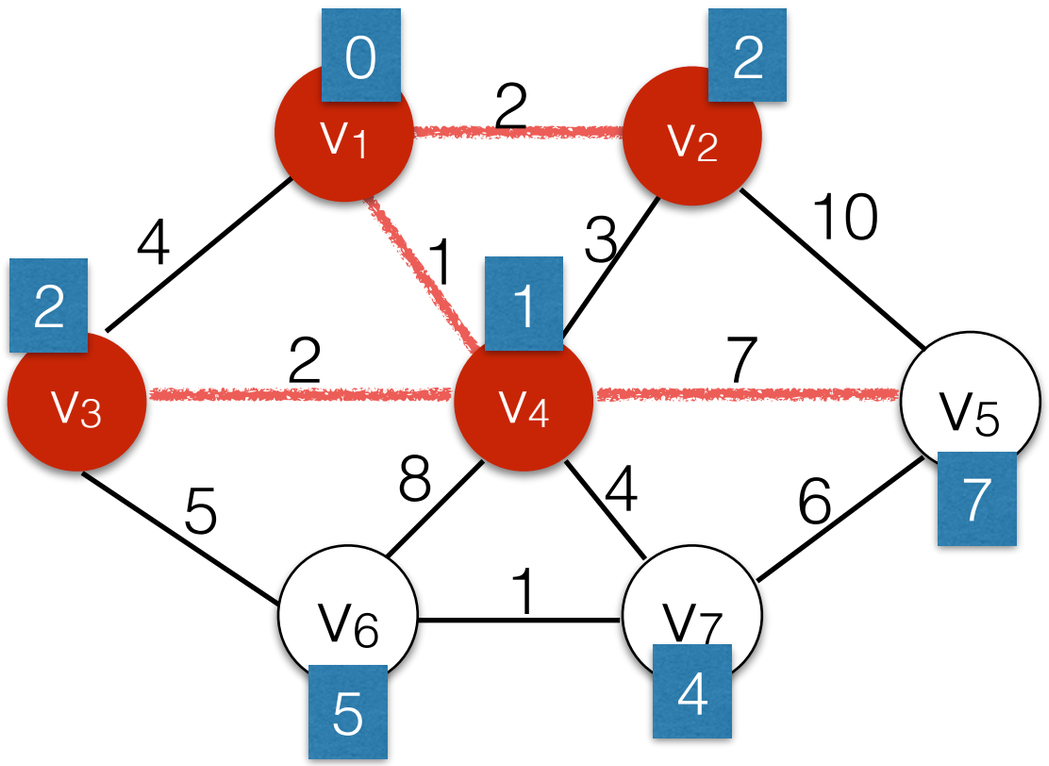

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



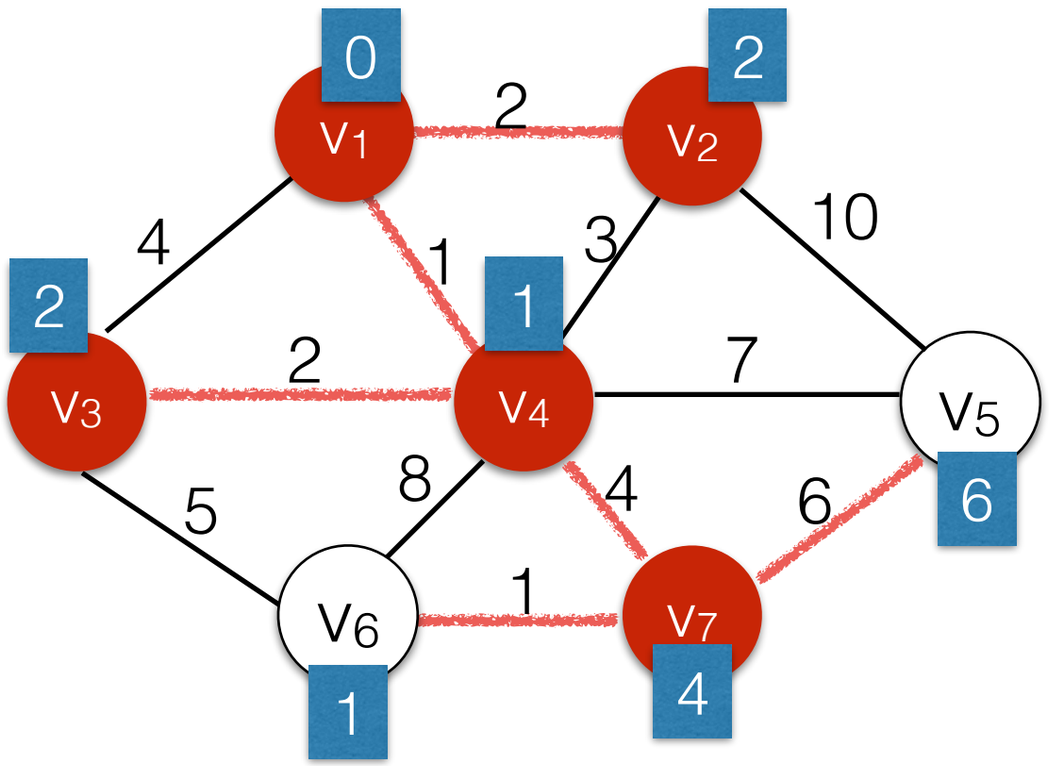

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



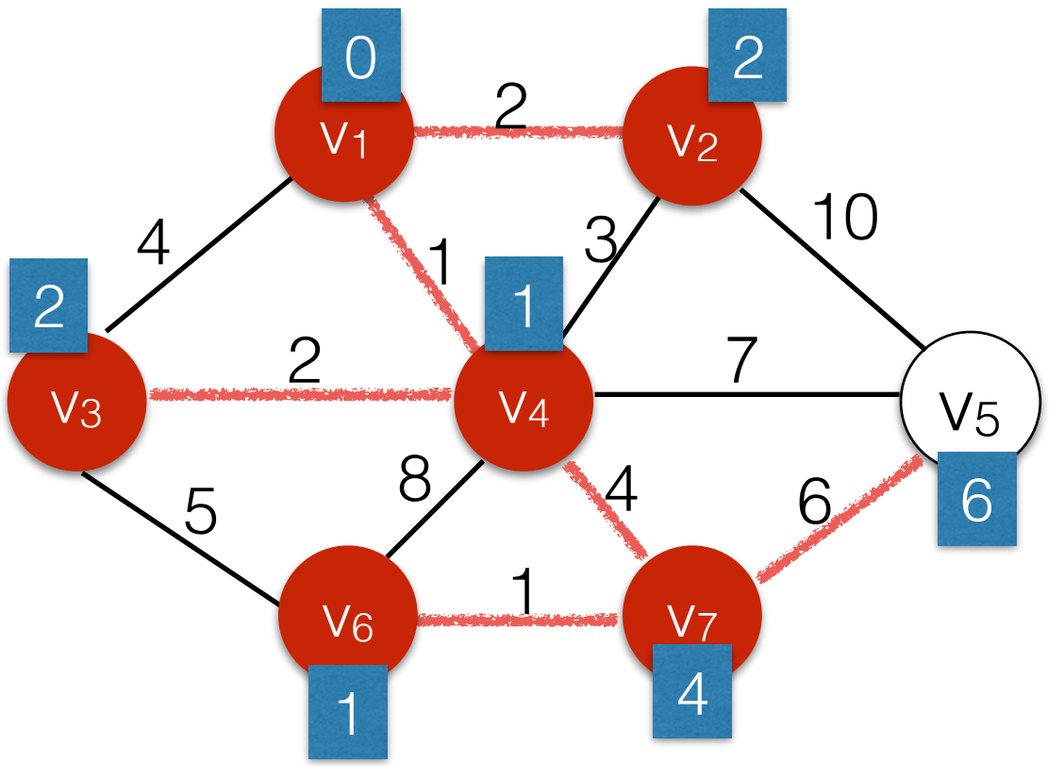

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



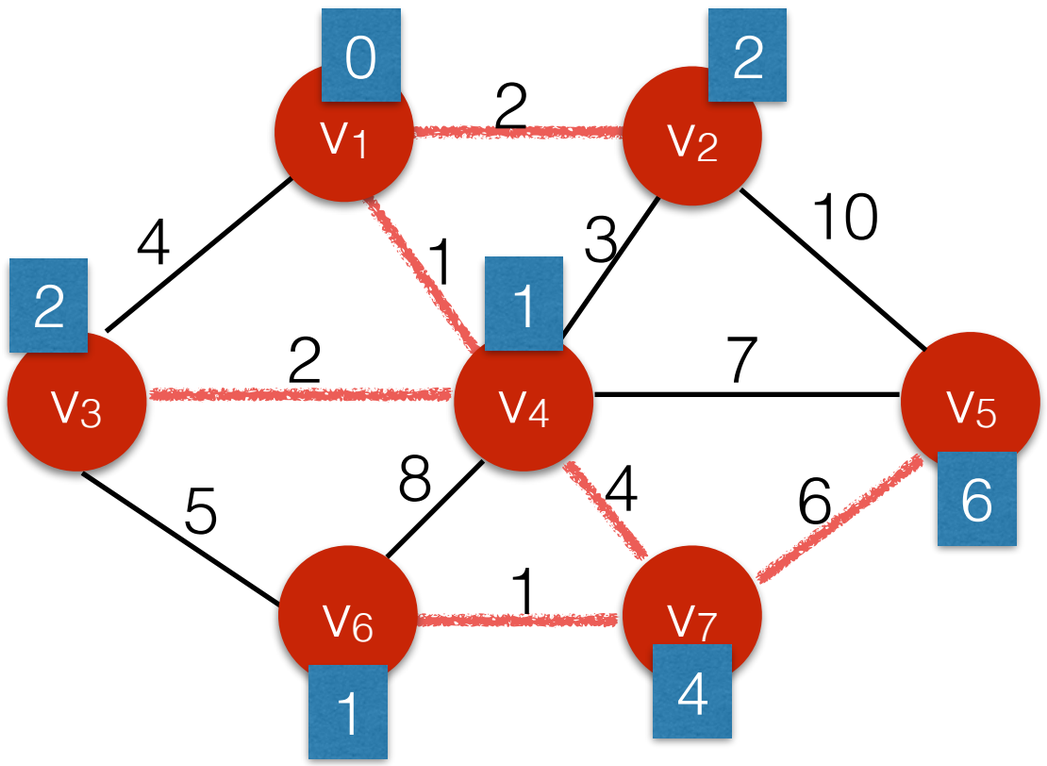

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



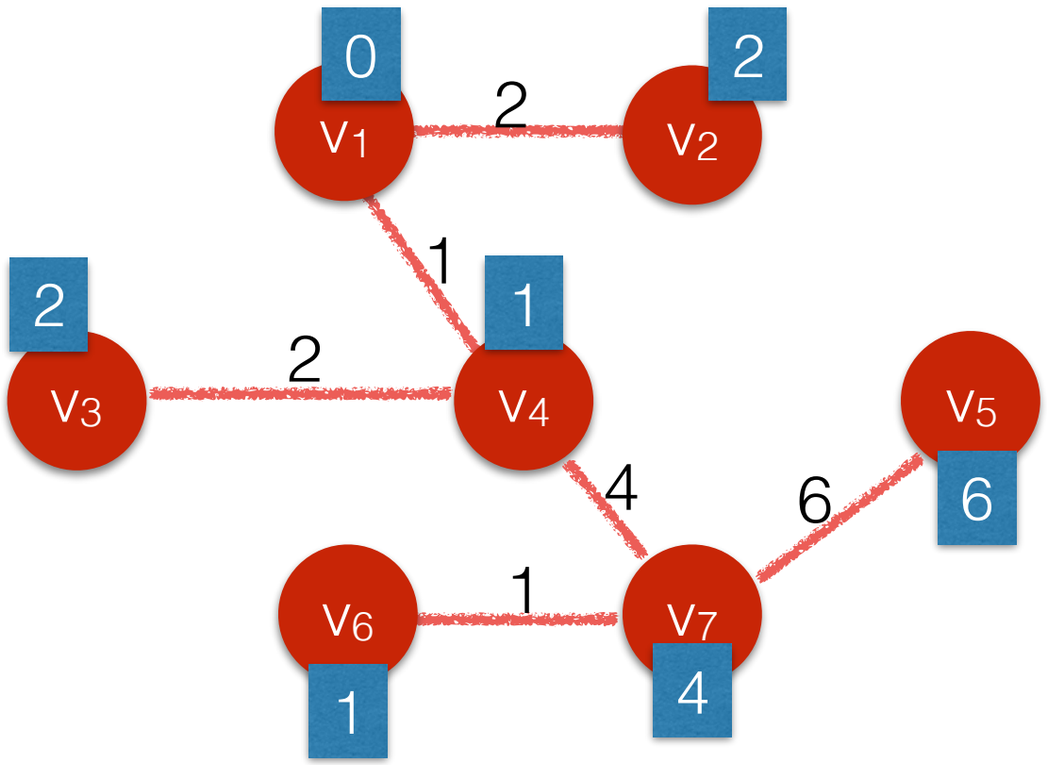

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



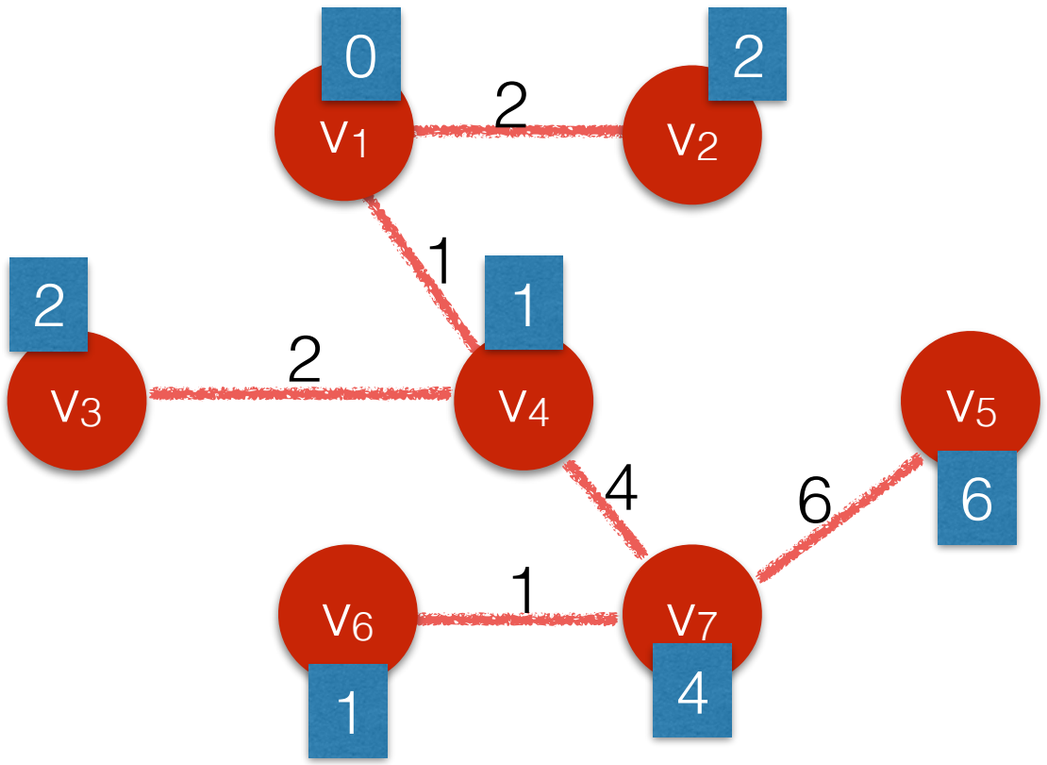

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

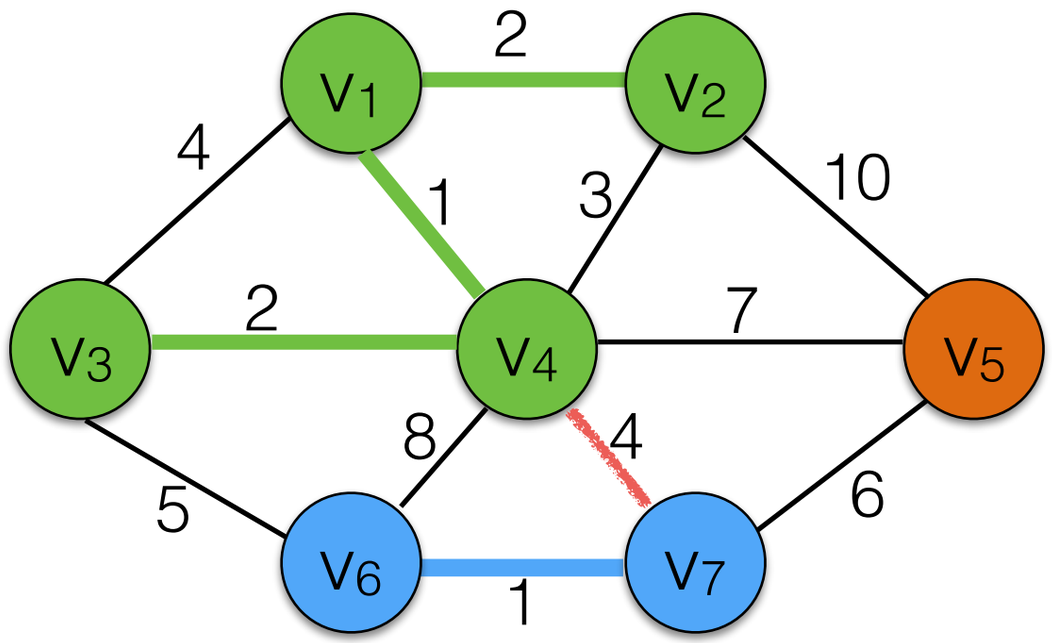



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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):
        .cost = cost(u,v)
        .prev = u
        .insert(v)
```

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

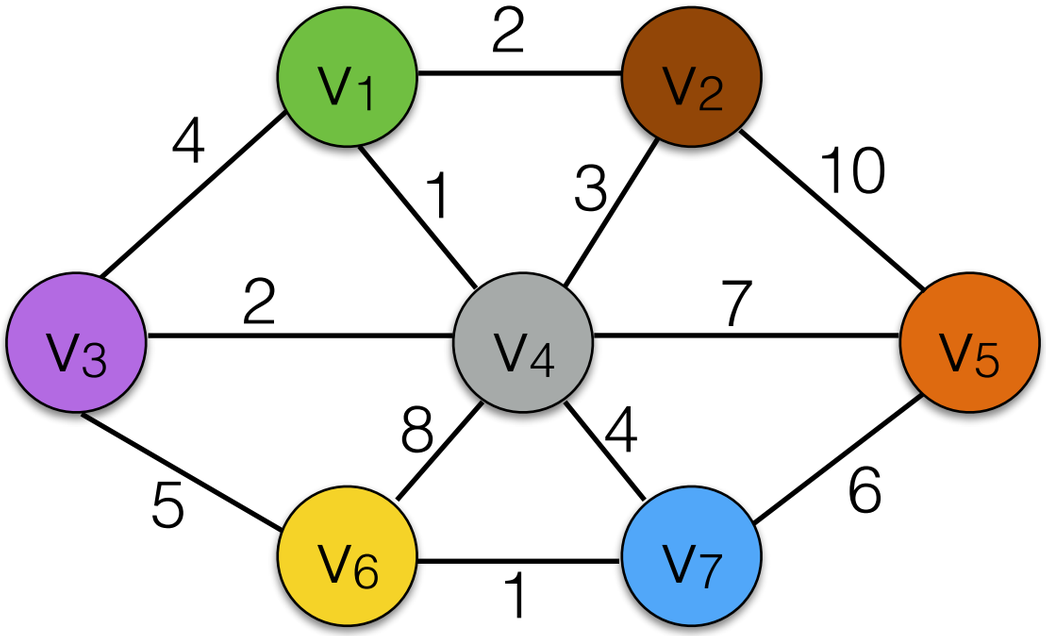
Kruskal's Algorithm for finding MSTs

- 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.



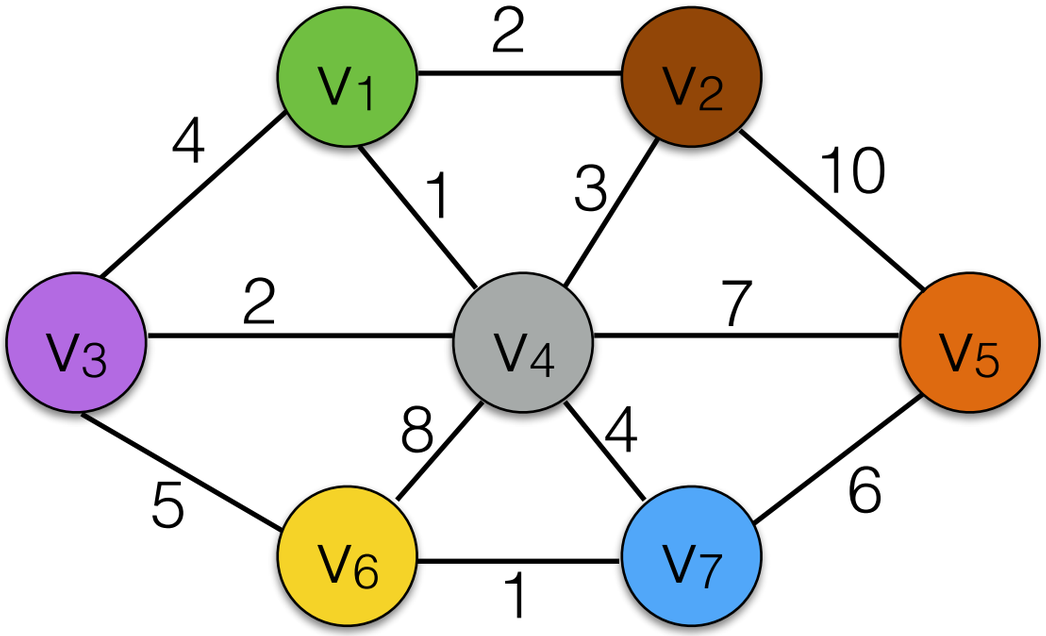
Kruskal's Algorithm

Sort edges (or keep them on a heap)



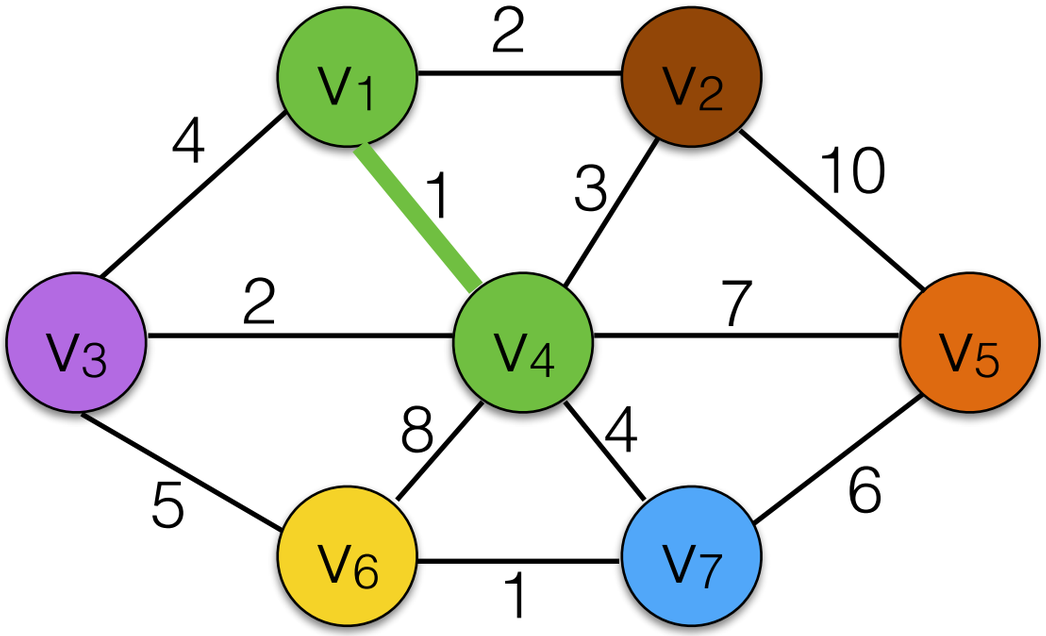
(v1,v2)	2
(v1,v3)	4
(v1,v4)	1
(v2,v4)	3
(v2,v5)	10
(v3,v4)	2
(v3,v6)	5
(v4,v5)	7
(v4,v6)	8
(v4,v7)	4
(v5,v7)	6
(v6,v7)	1

Kruskal's Algorithm



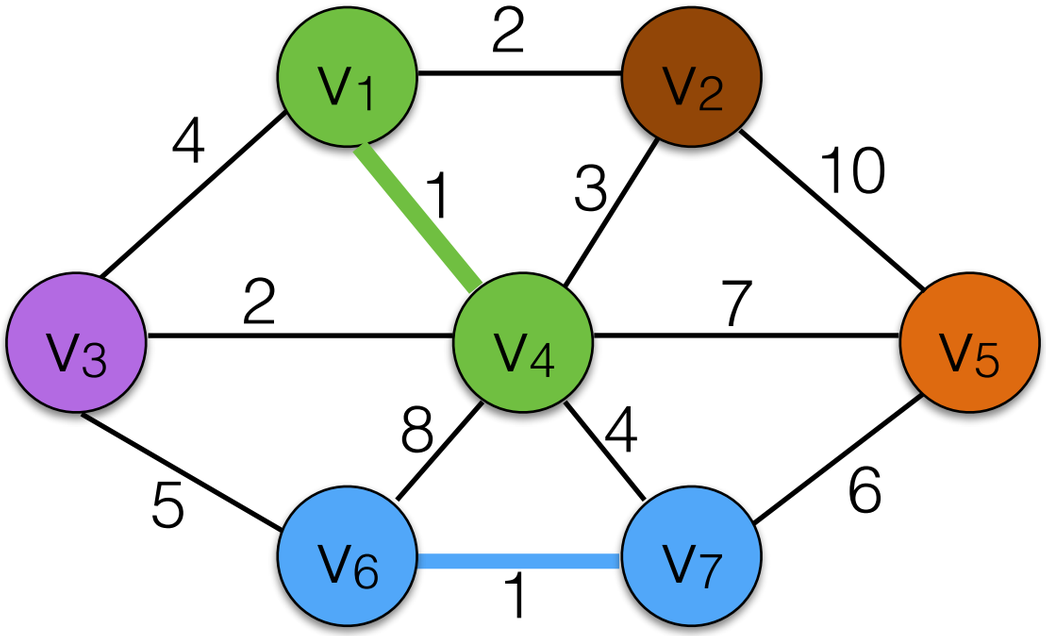
(v1,v4)	1
(v6,v7)	1
(v1,v2)	2
(v3,v4)	2
(v2,v4)	3
(v1,v3)	4
(v4,v7)	4
(v3,v6)	5
(v5,v7)	6
(v4,v5)	7
(v4,v6)	8
(v2,v5)	10

Kruskal's Algorithm



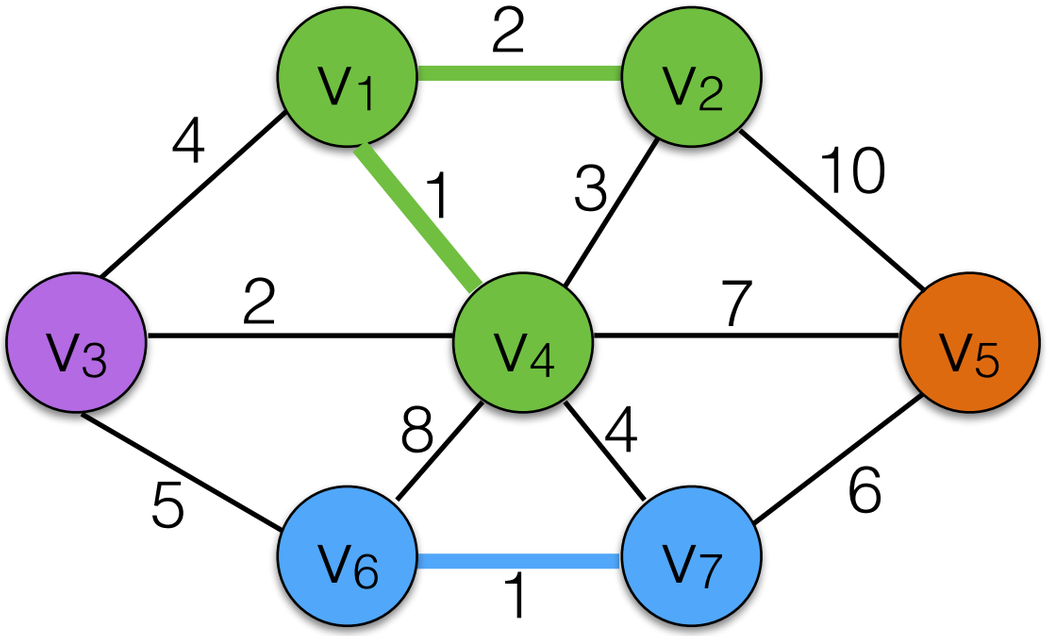
(v1,v4)	1
(v6,v7)	1
(v1,v2)	2
(v3,v4)	2
(v2,v4)	3
(v1,v3)	4
(v4,v7)	4
(v3,v6)	5
(v5,v7)	6
(v4,v5)	7
(v4,v6)	8
(v2,v5)	10

Kruskal's Algorithm



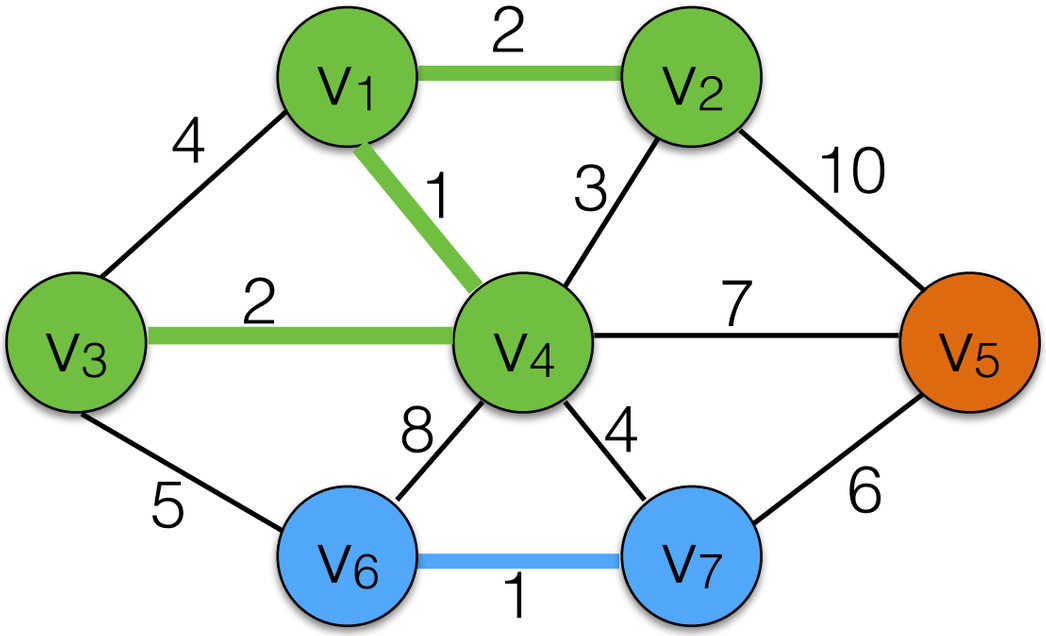
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	
(v3,v4)	2	
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



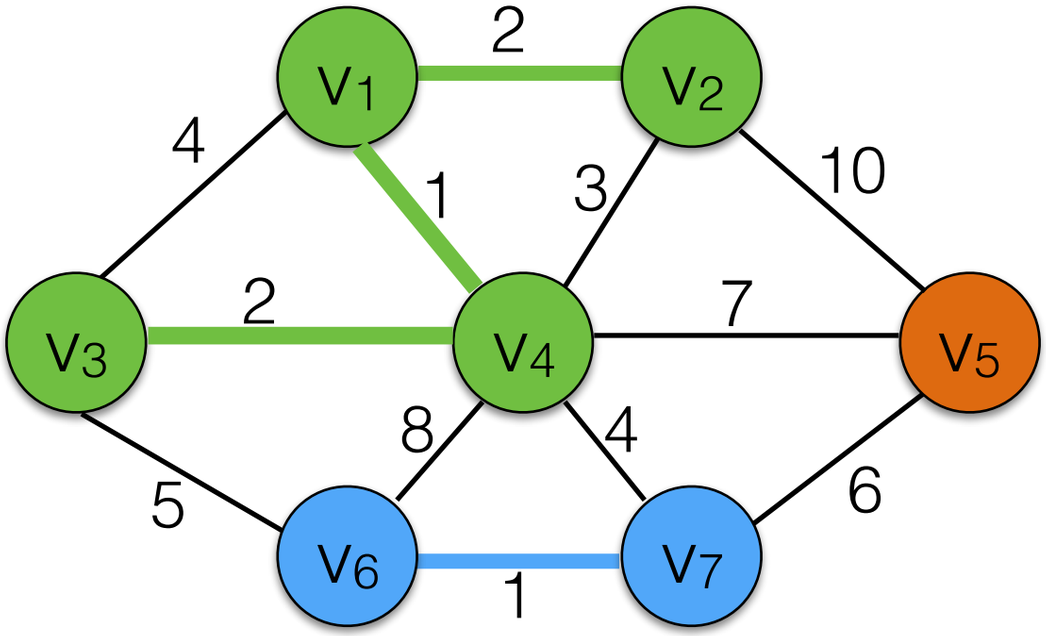
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



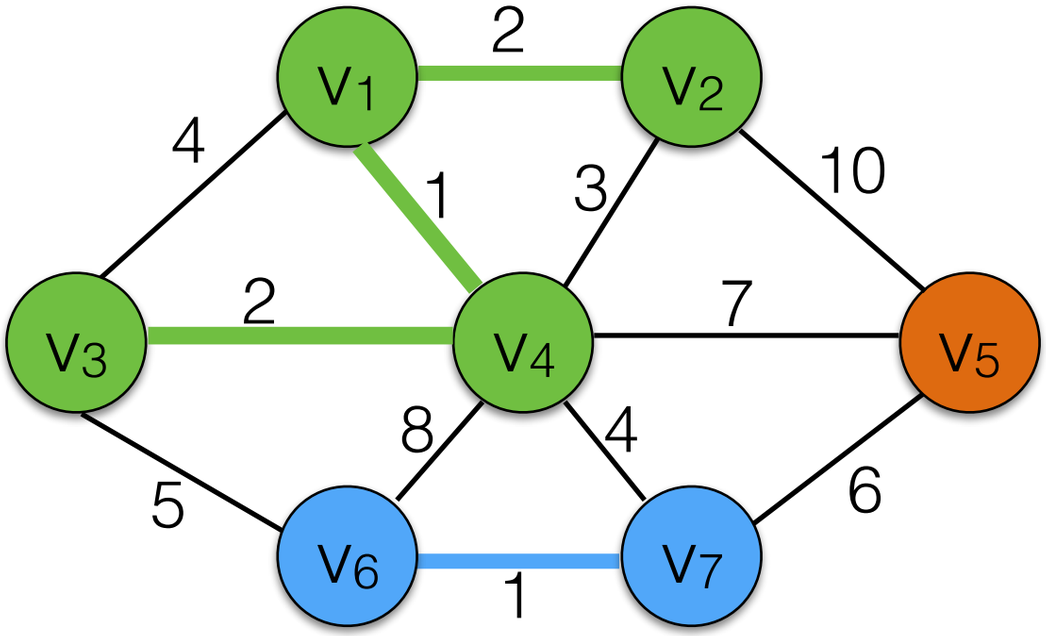
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



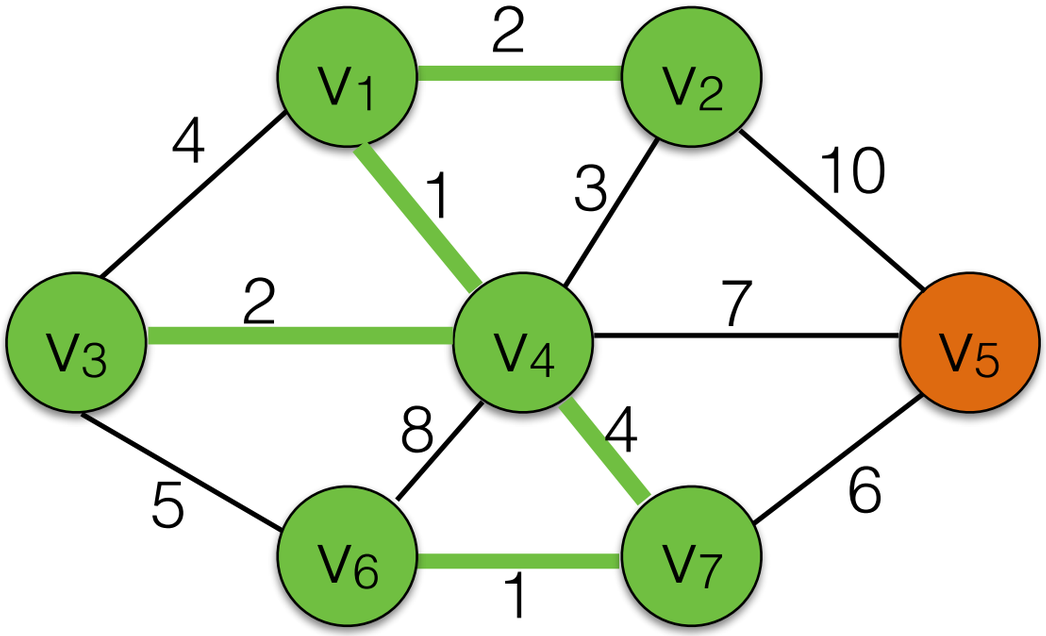
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



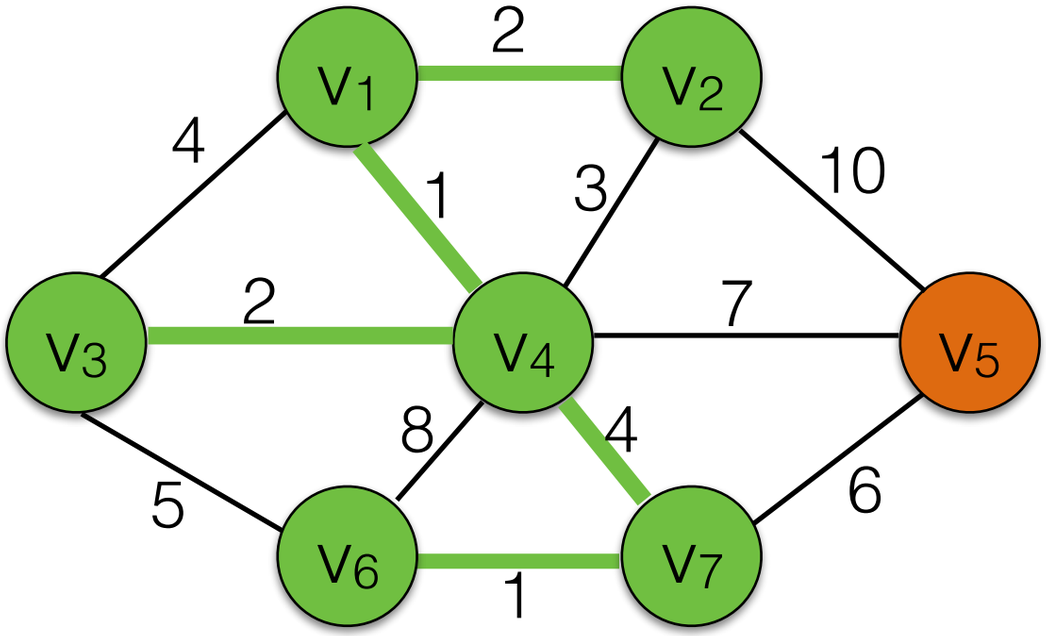
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

Kruskal's Algorithm



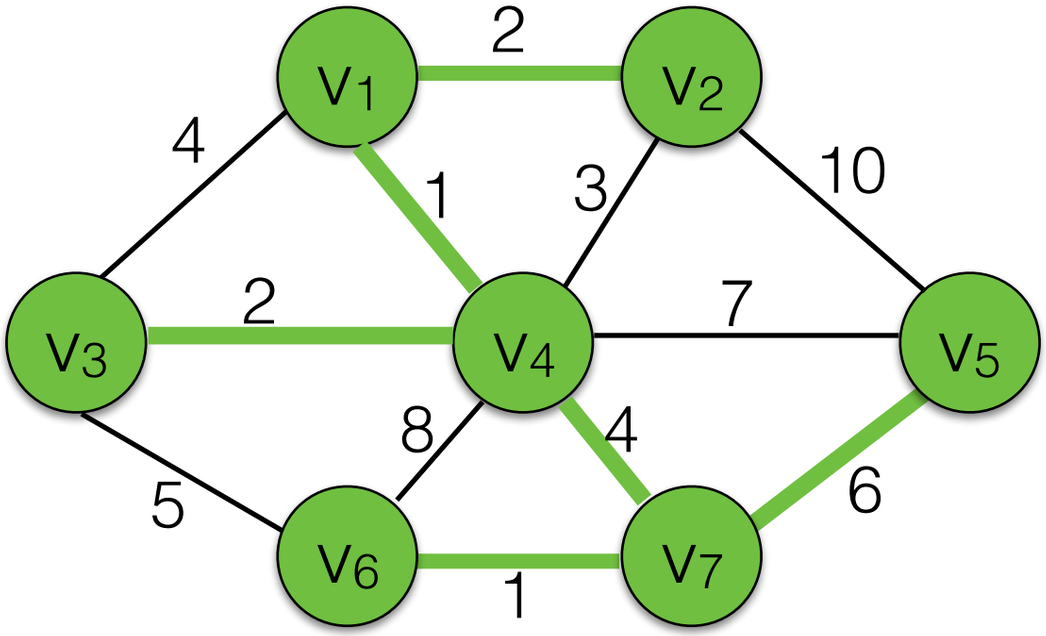
(v1,v4)	1	OK
(v6,v7)	1	OK
(v1,v2)	2	OK
(v3,v4)	2	OK
(v2,v4)	3	reject
(v1,v3)	4	reject
(v4,v7)	4	OK
(v3,v6)	5	
(v5,v7)	6	
(v4,v5)	7	
(v4,v6)	8	
(v2,v5)	10	

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

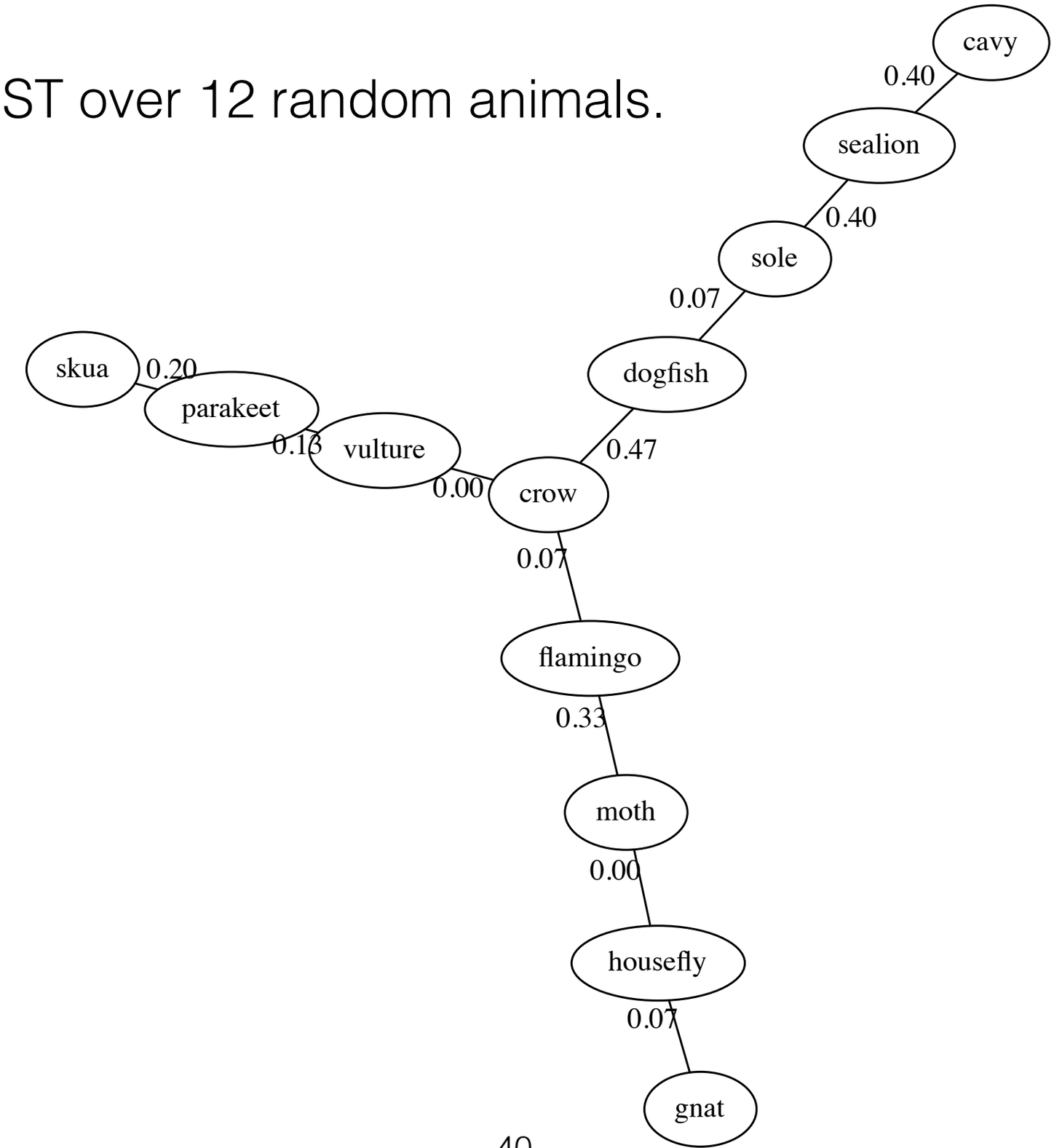
represent each data item as a vector of integers (15 attributes).

	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	0	1	1	0	
domestic	0	1	0	0	

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

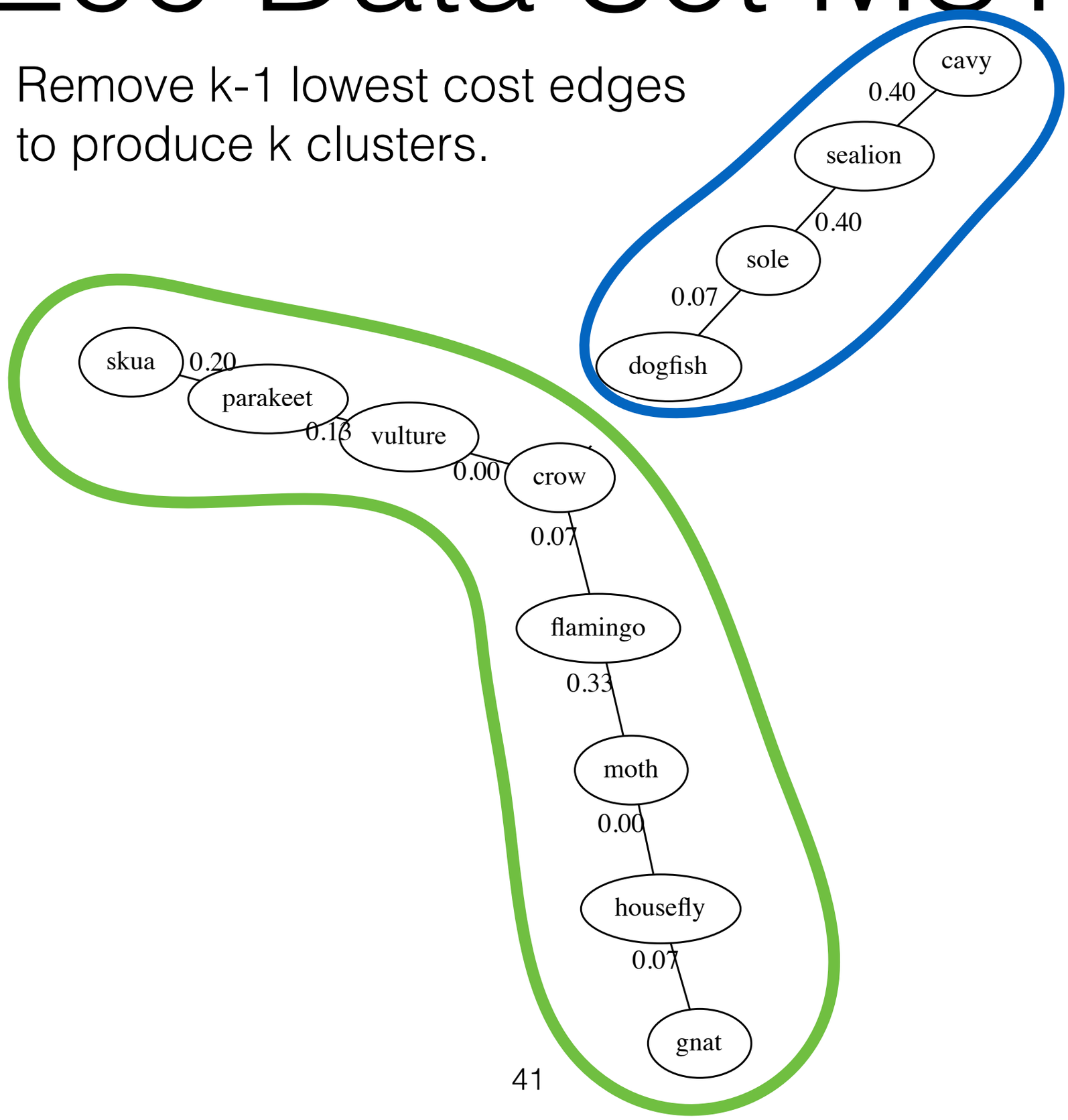
Zoo Data Set MST

- MST over 12 random animals.



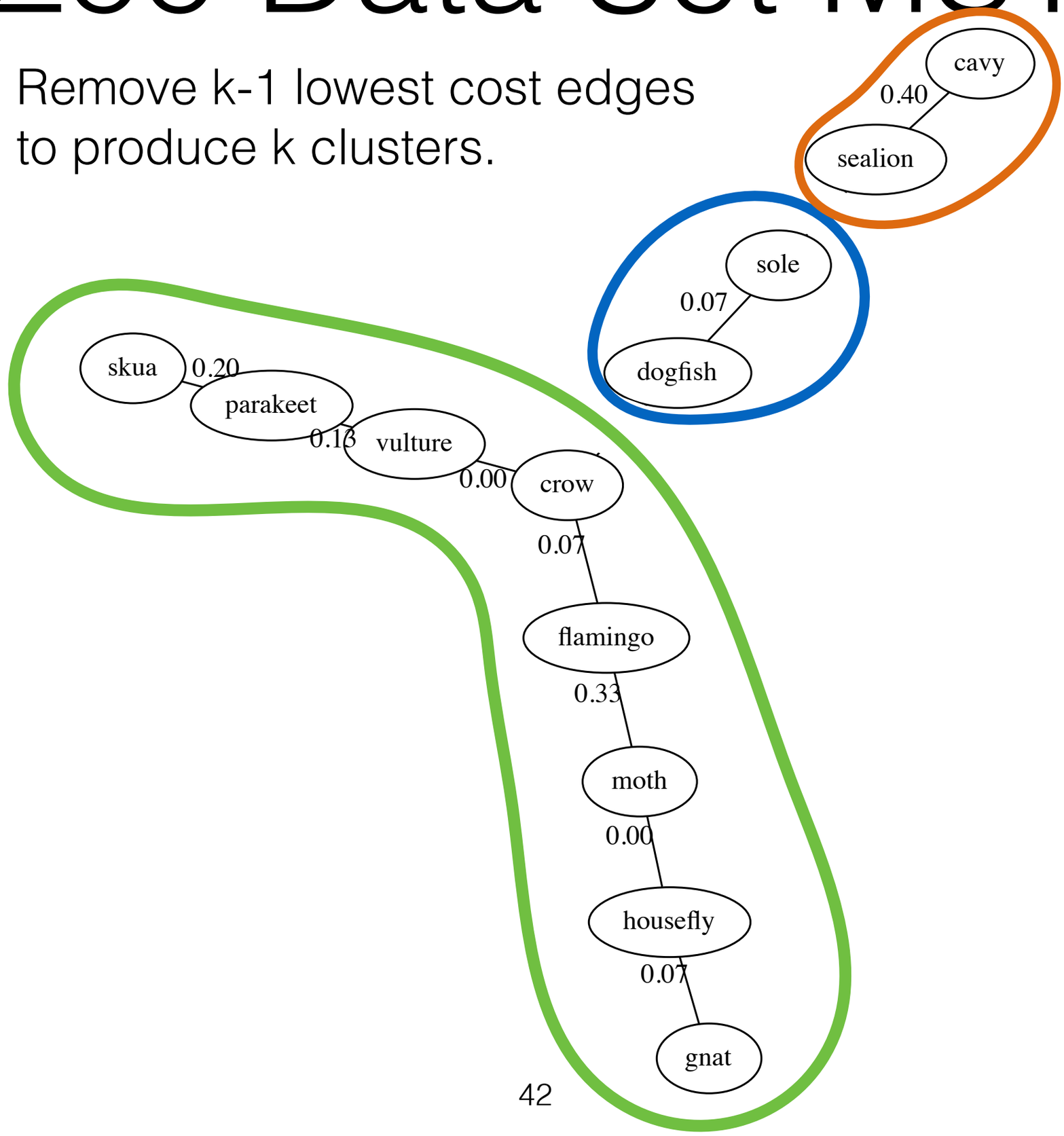
Zoo Data Set MST

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



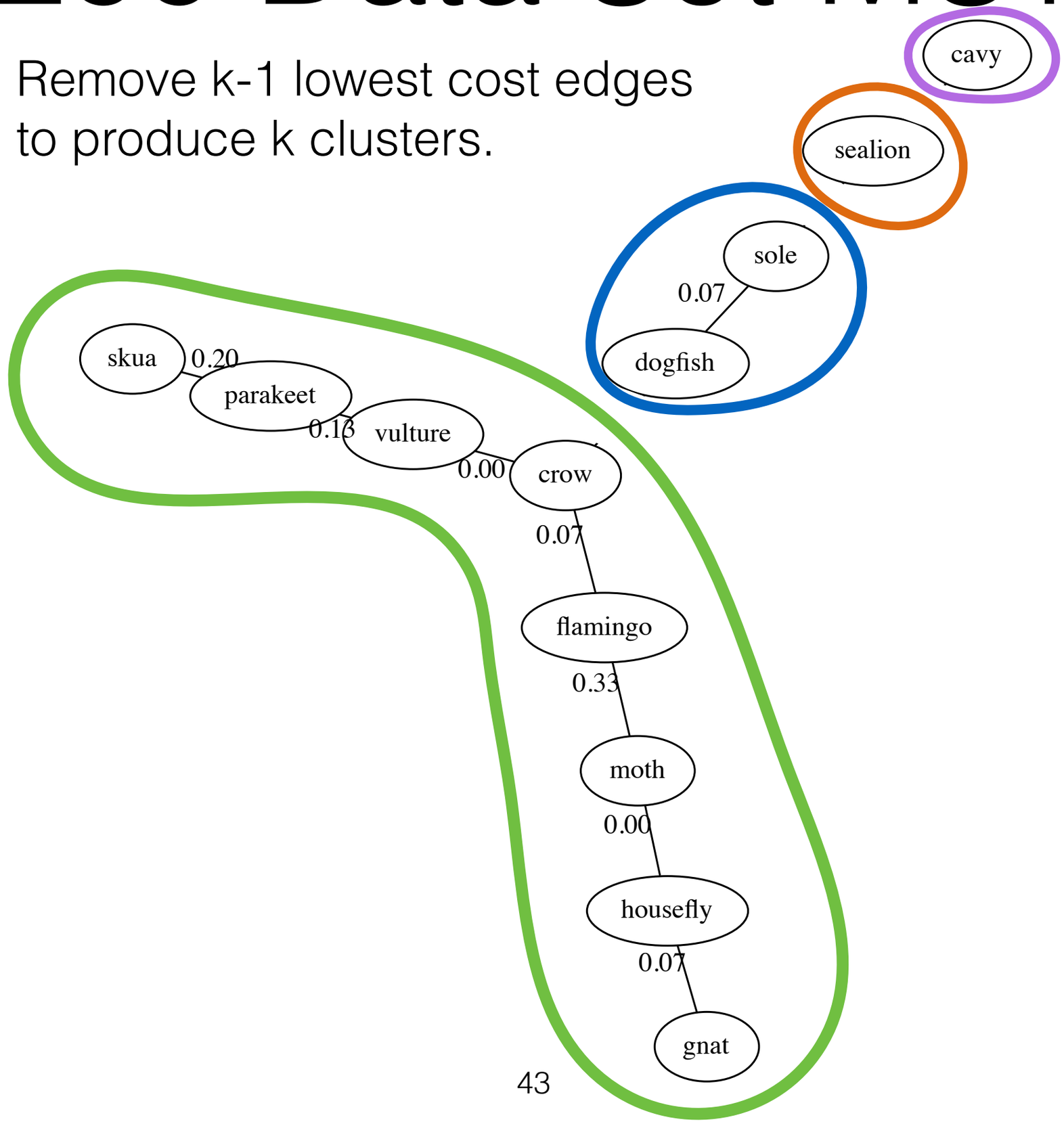
Zoo Data Set MST

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



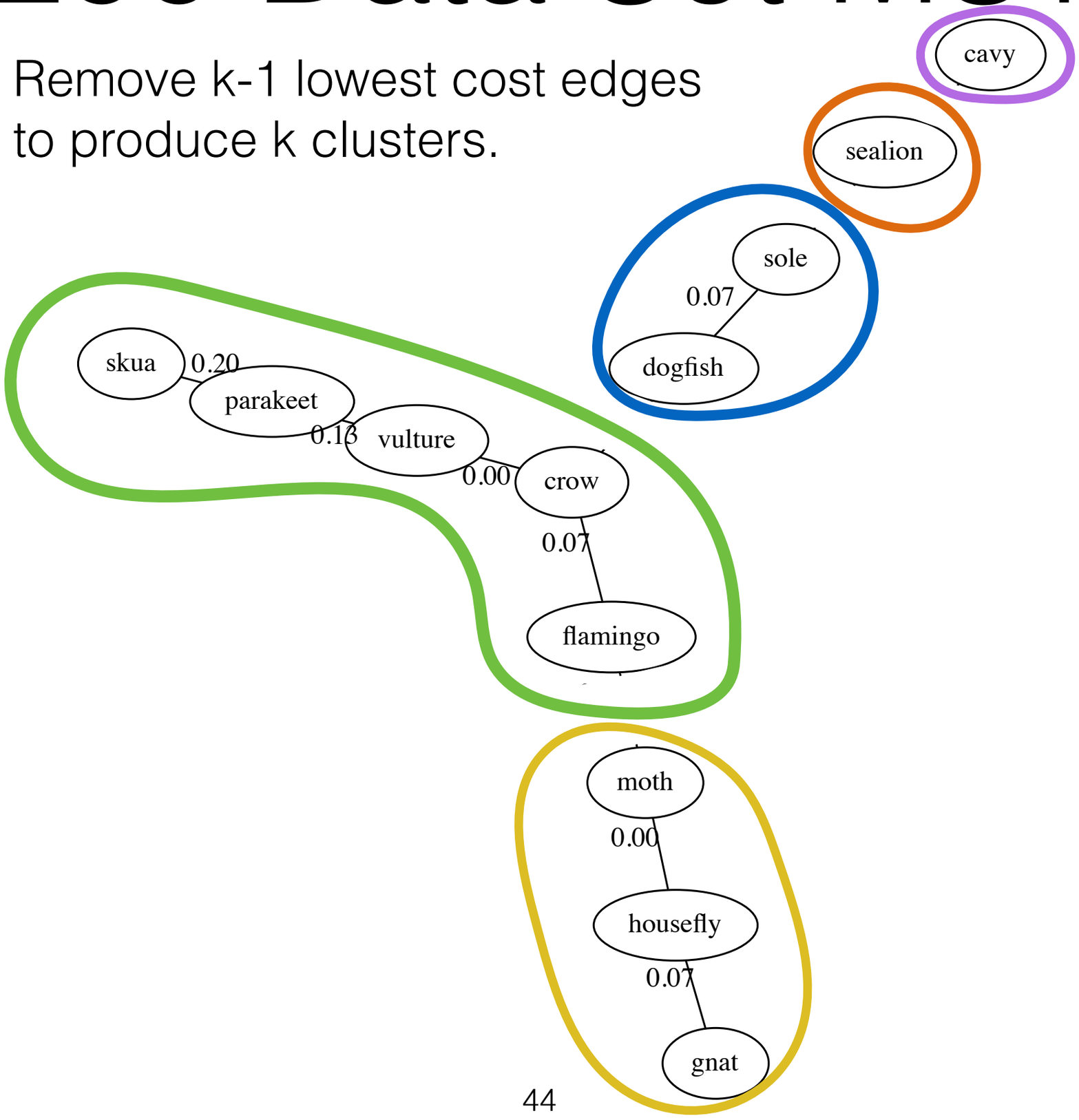
Zoo Data Set MST

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



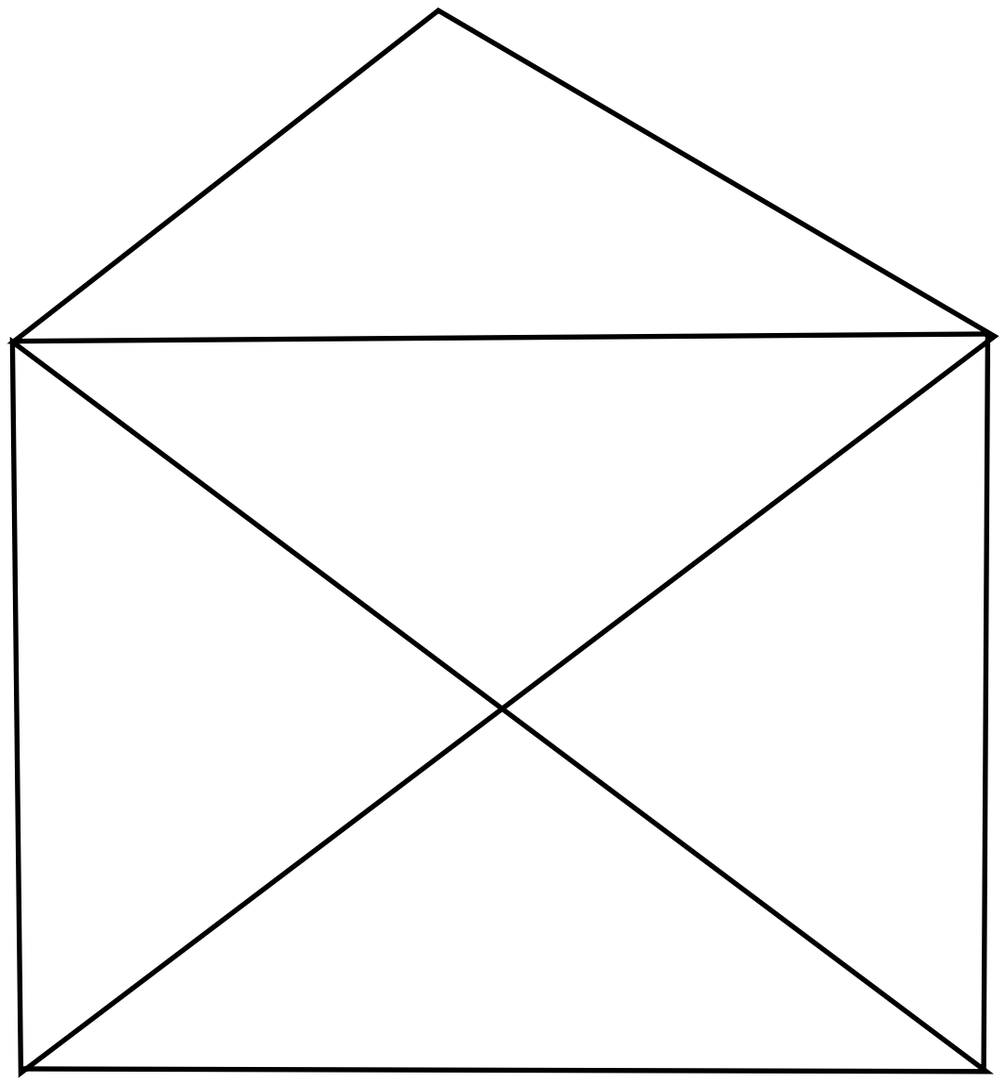
Zoo Data Set MST

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



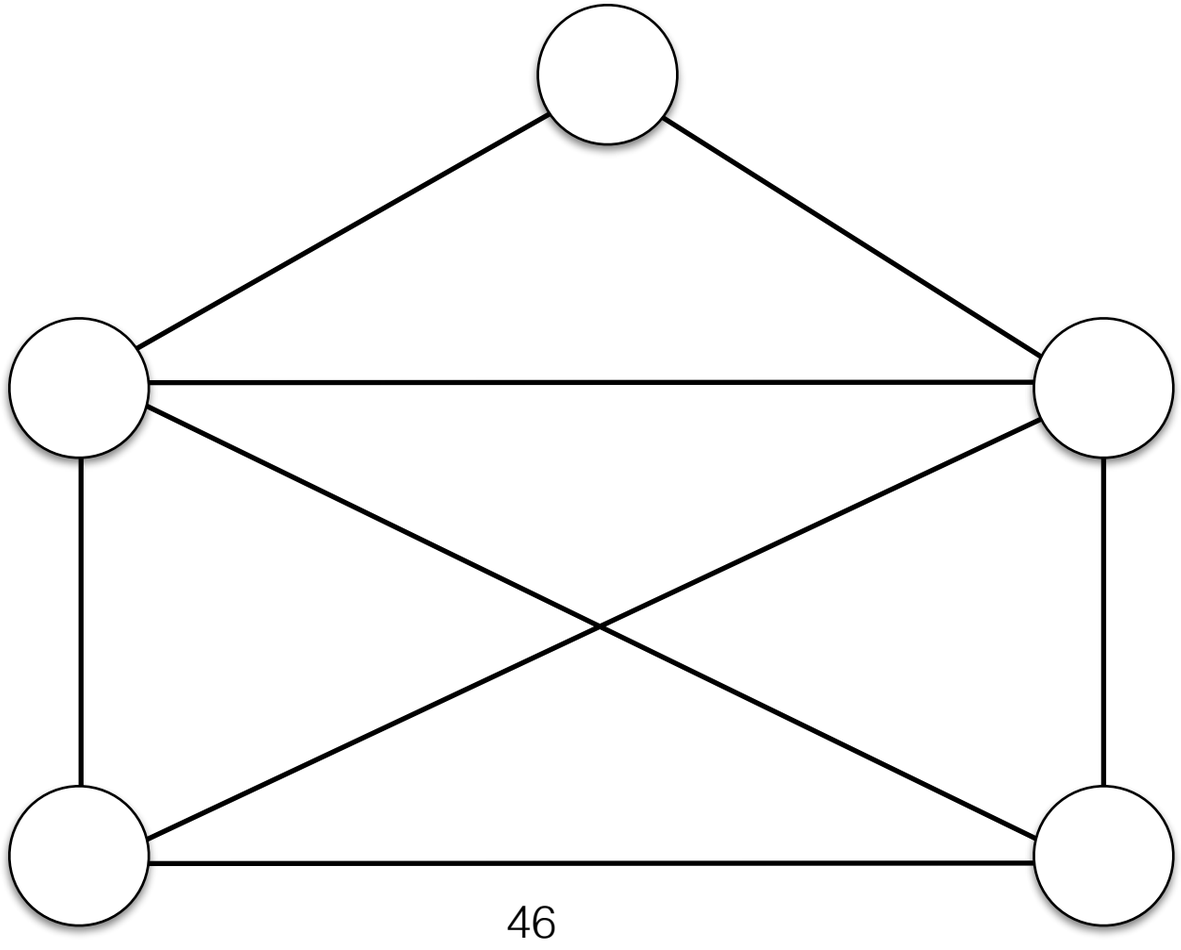
Draw this Figure

Without lifting your pen off the paper.



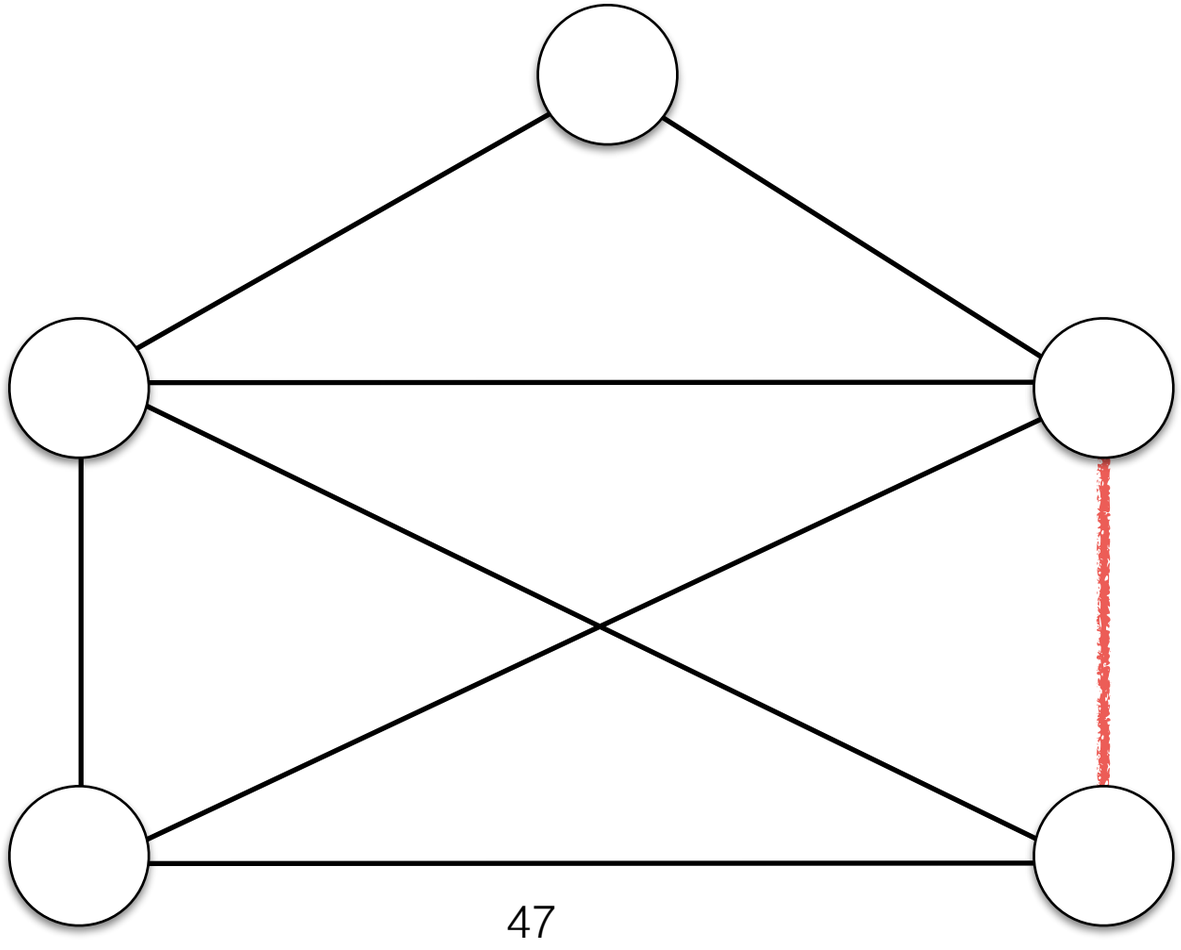
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



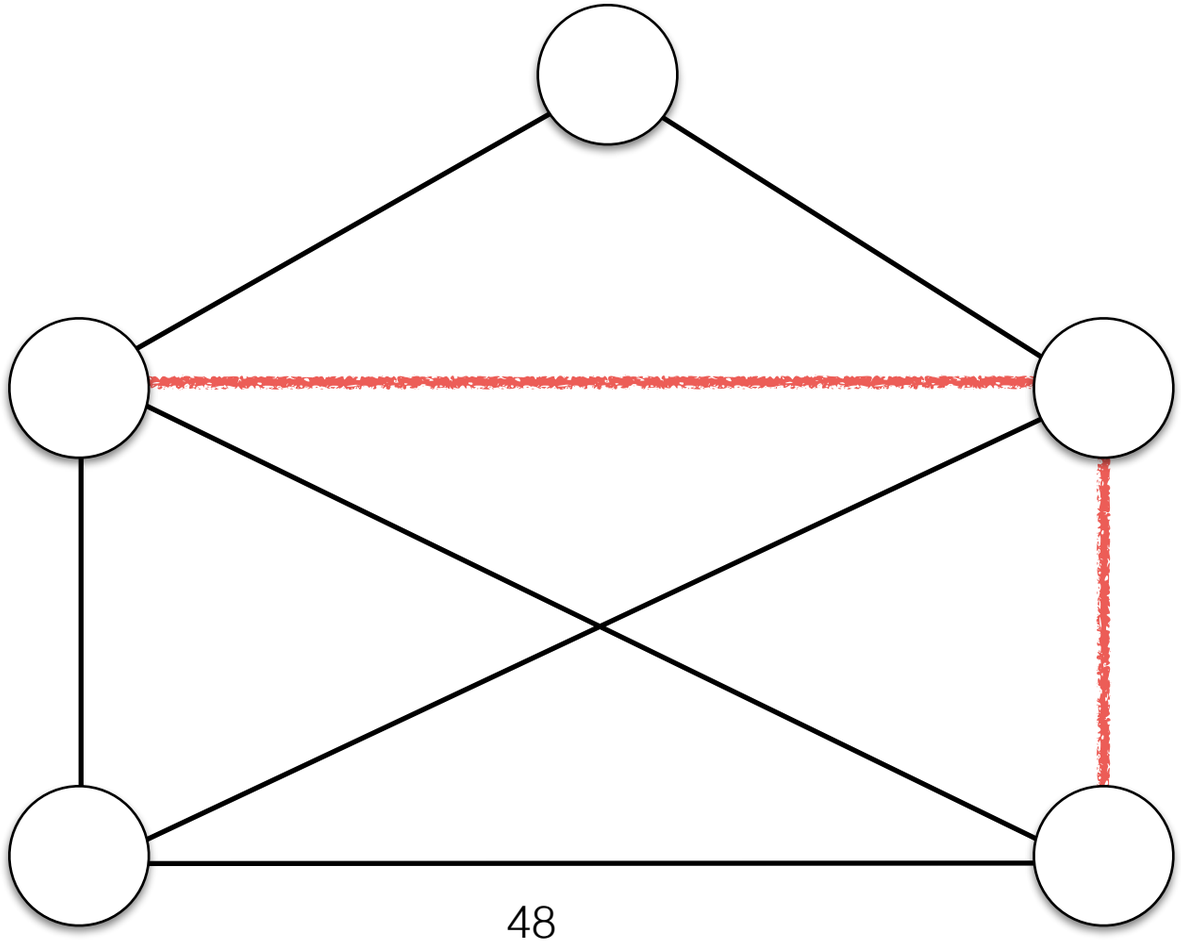
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



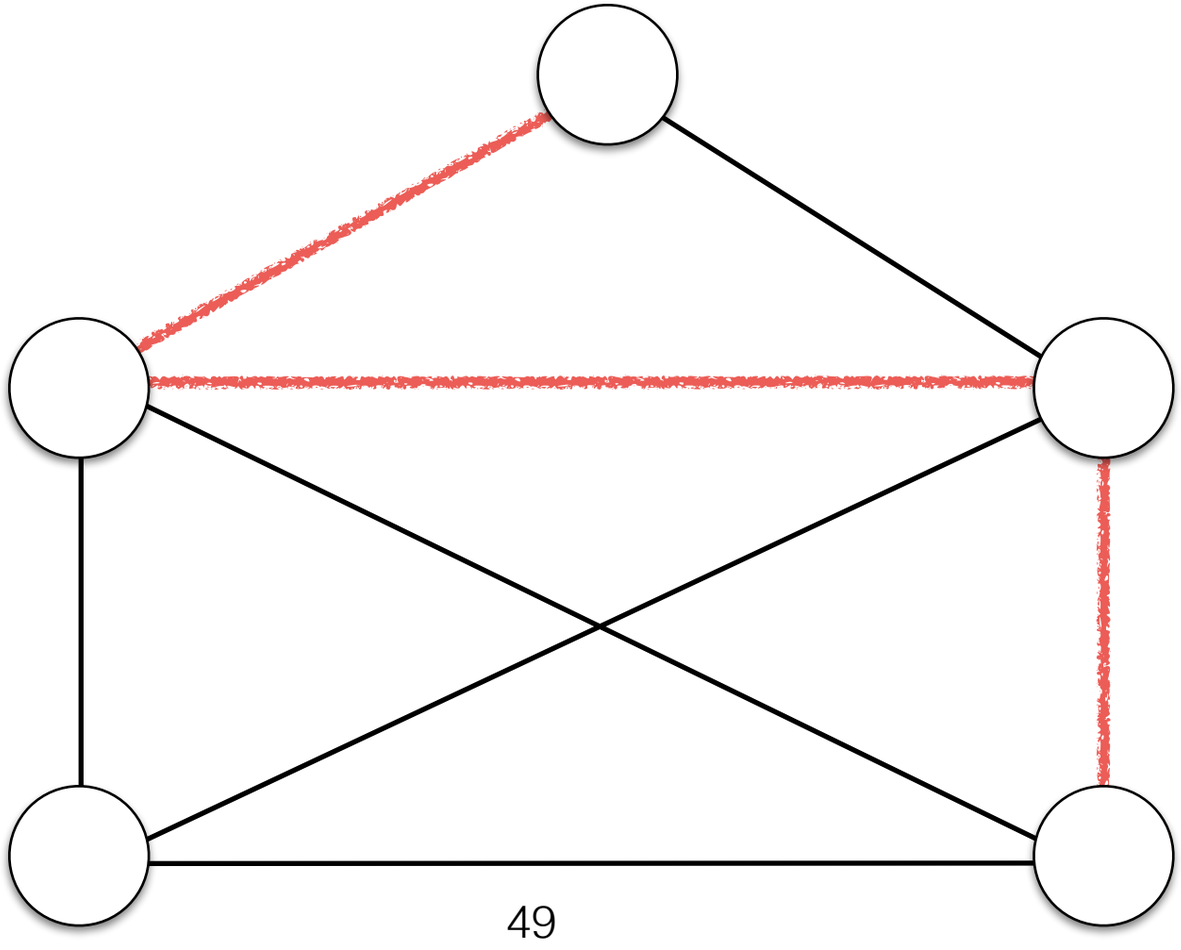
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



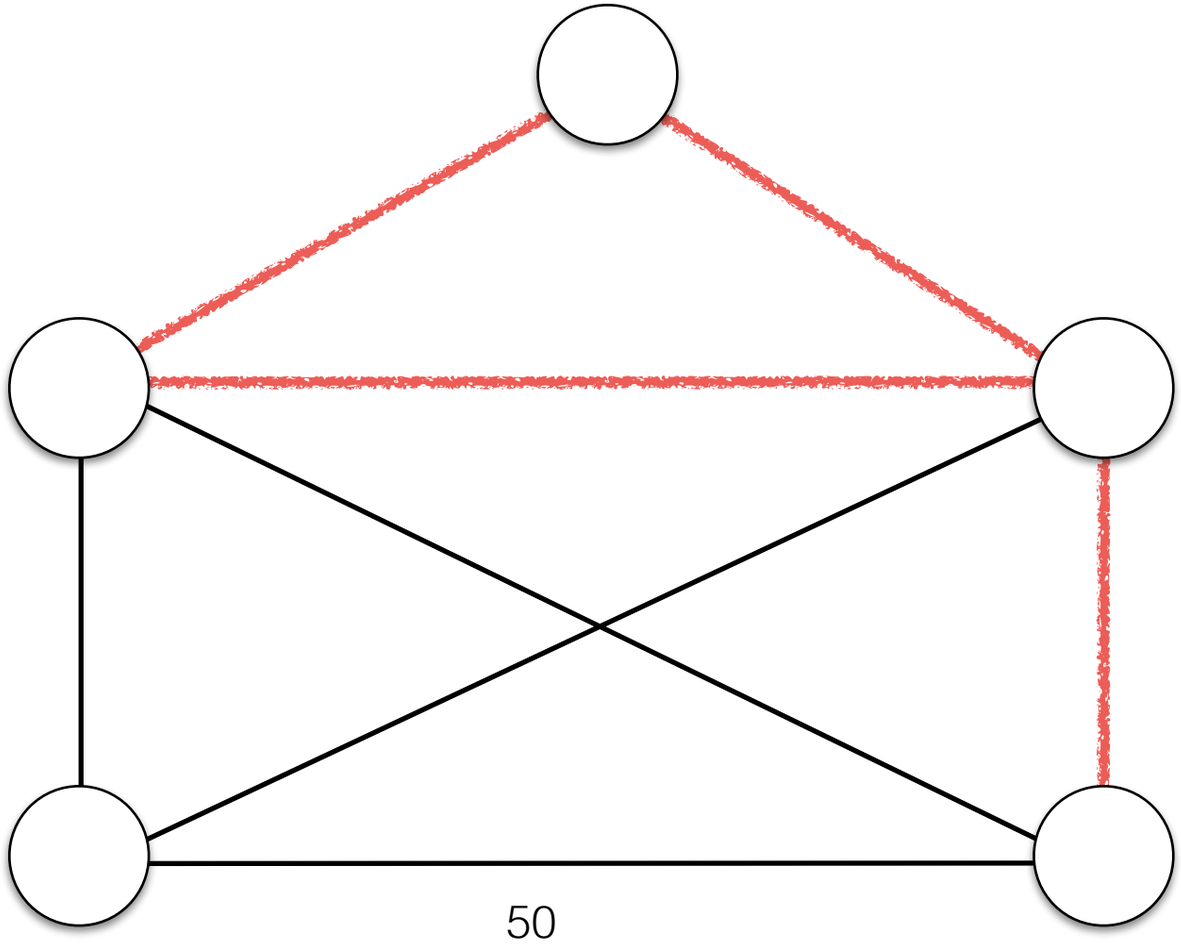
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



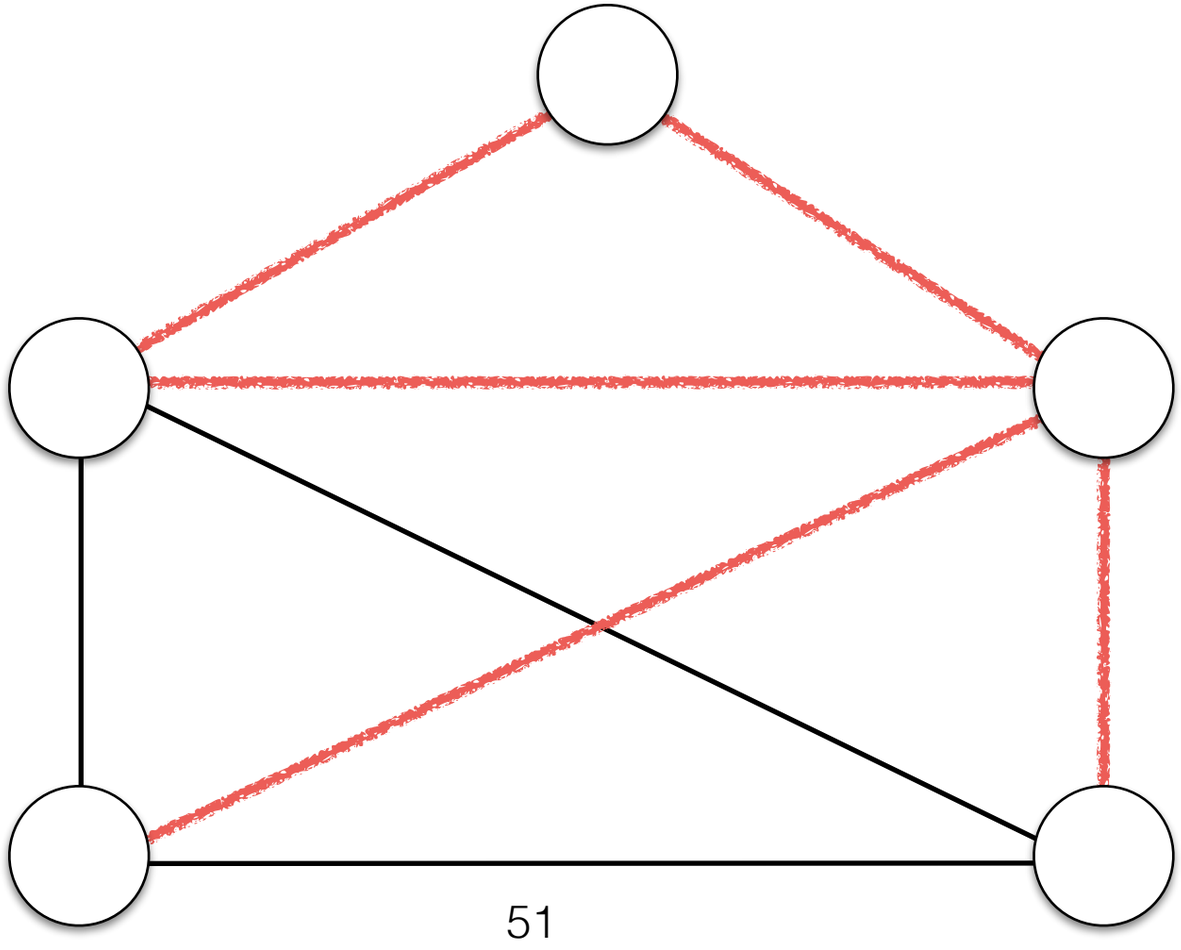
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



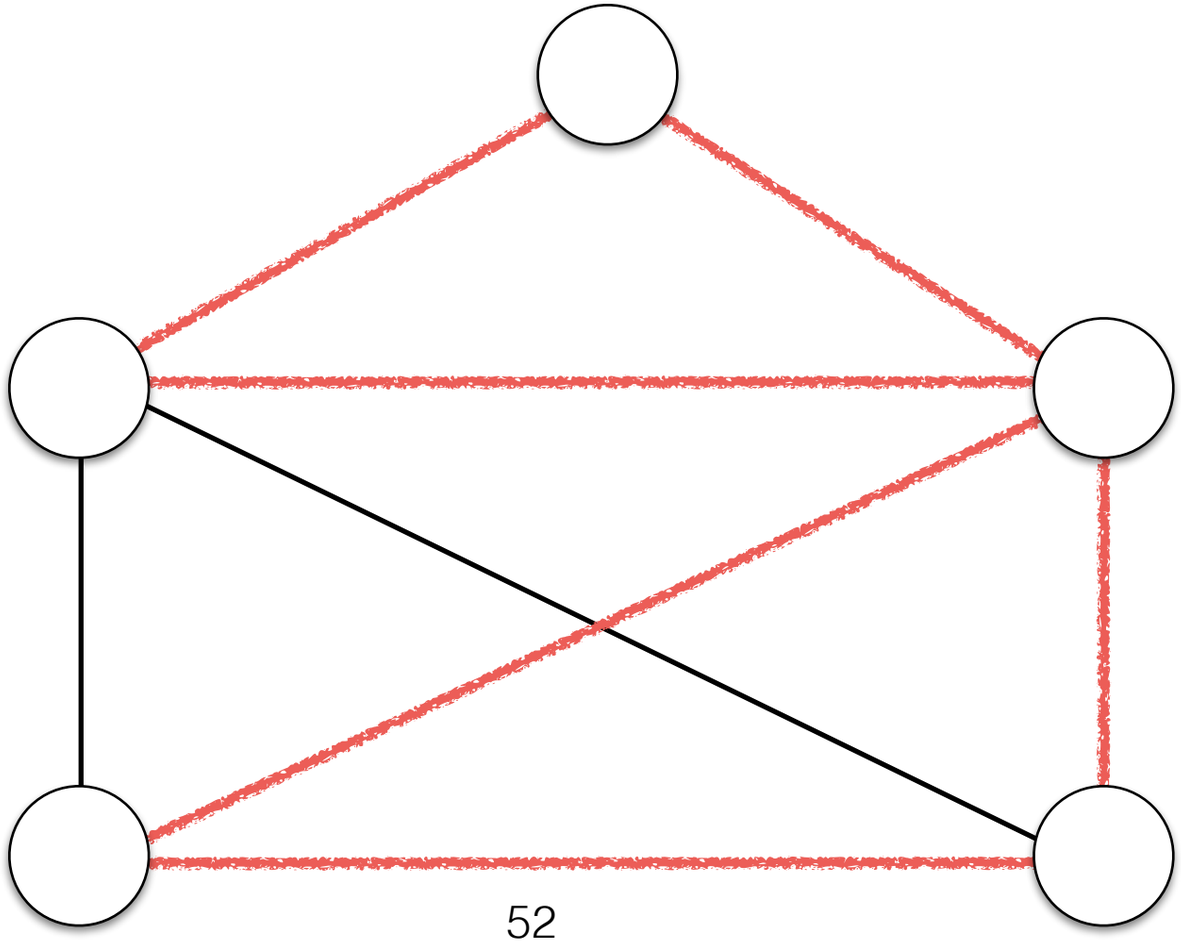
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



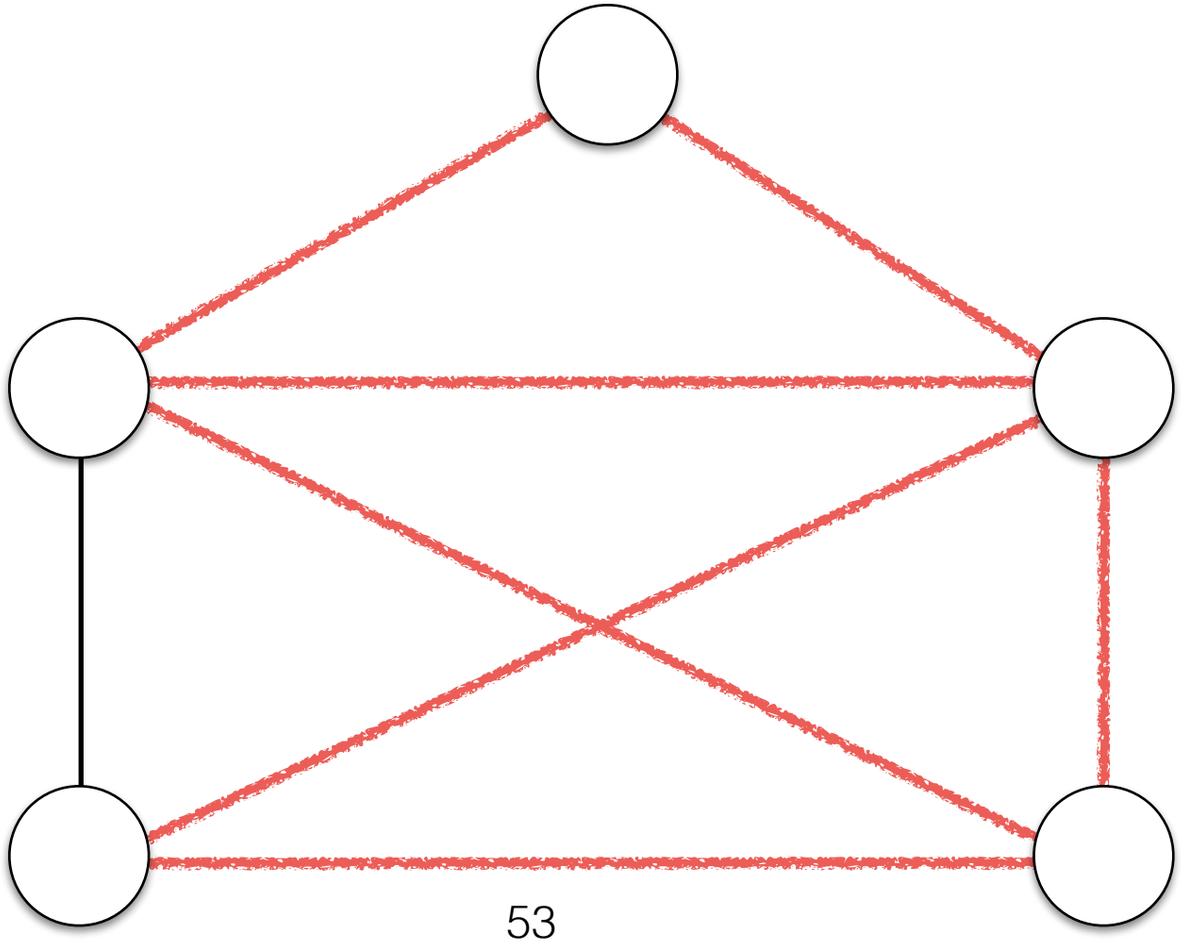
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



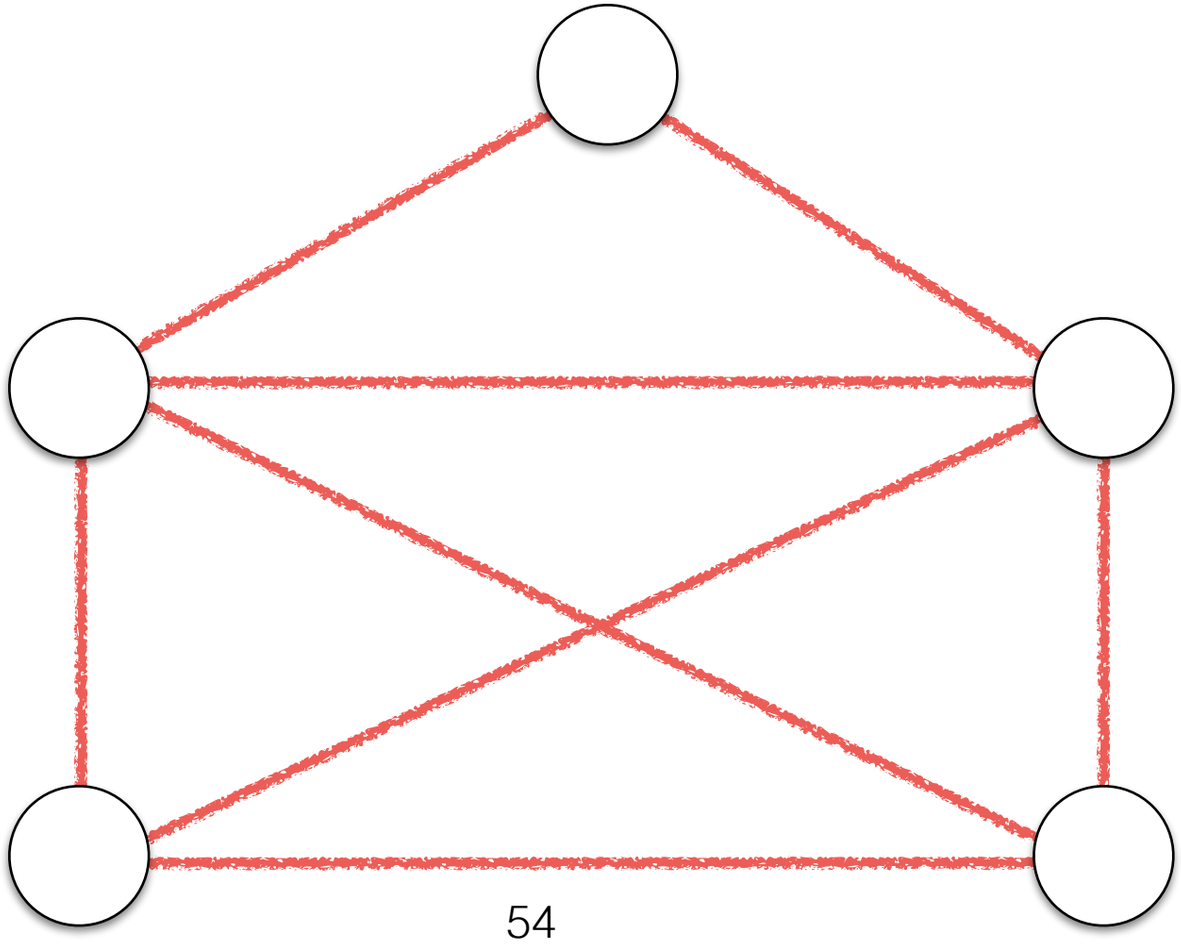
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



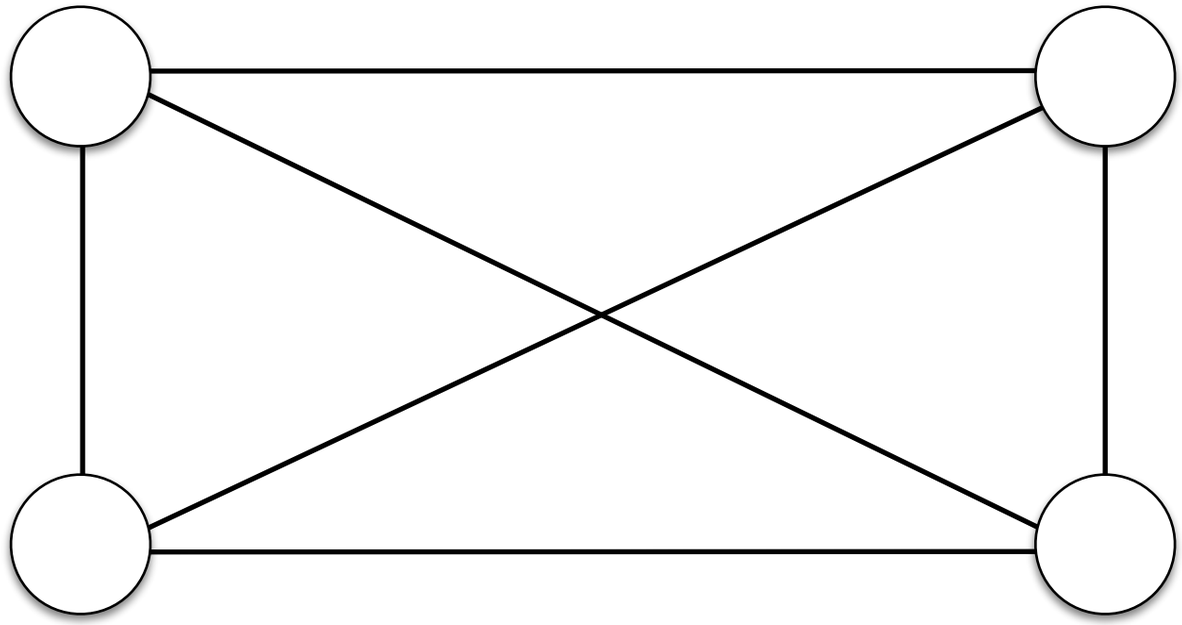
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



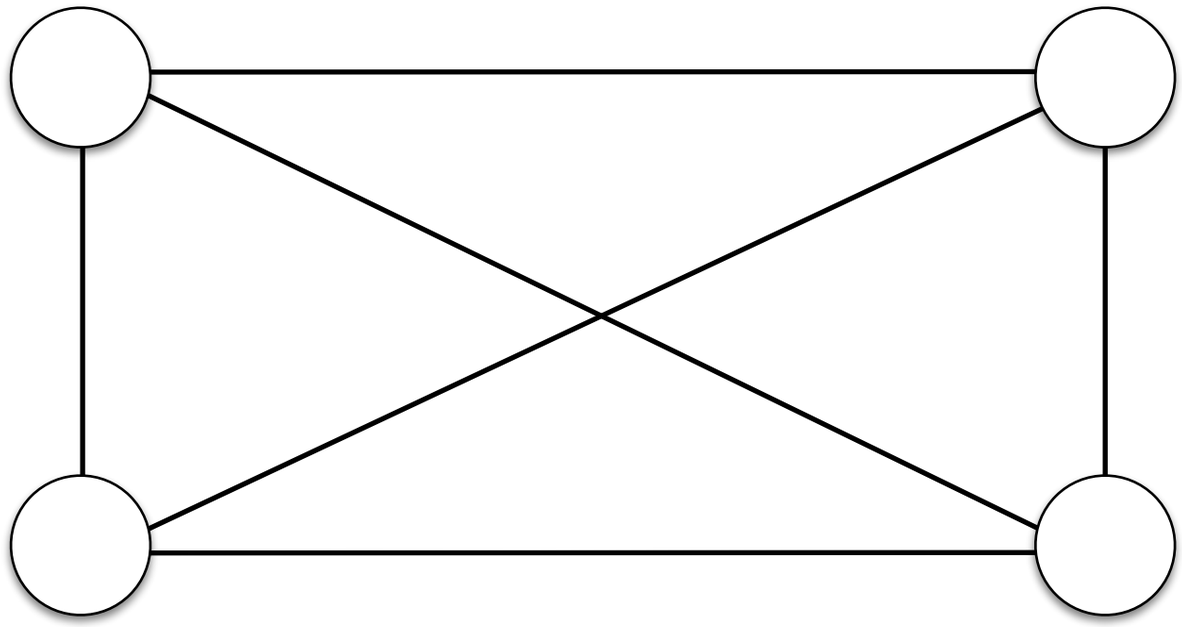
Euler Paths

- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



Euler Paths

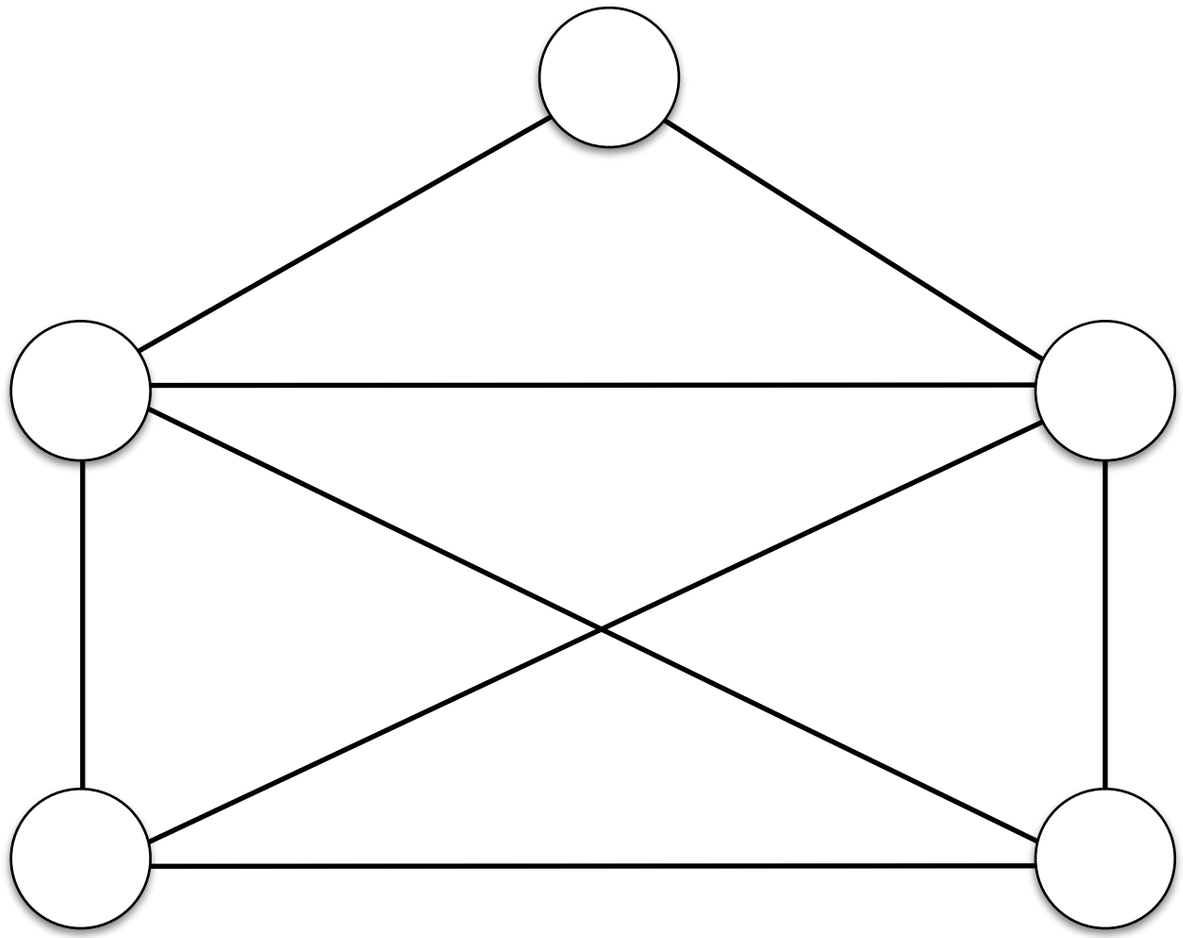
- An Euler Path is a path through an undirected graph that visits *every edge* exactly once.



This graph does not have an Euler Path.

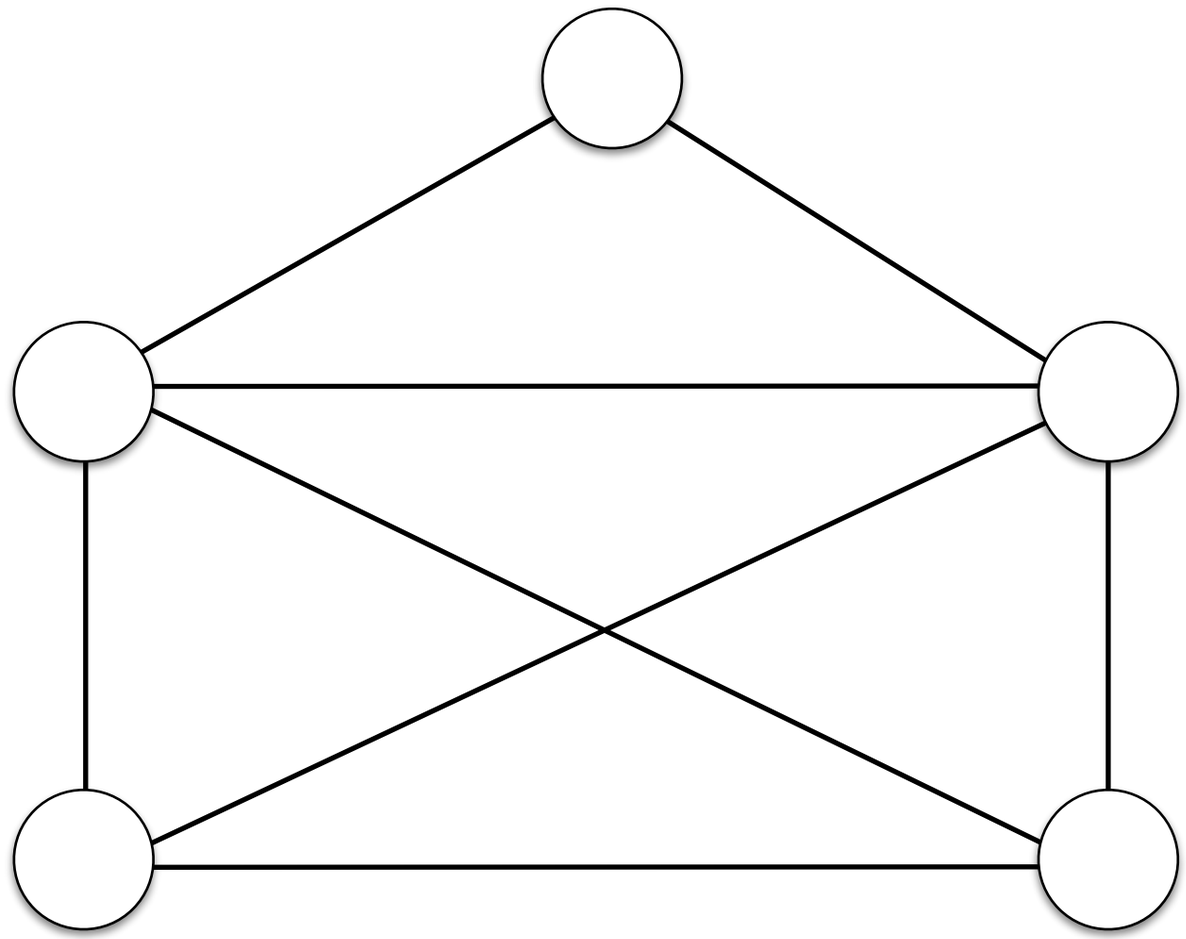
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



Euler Circuit

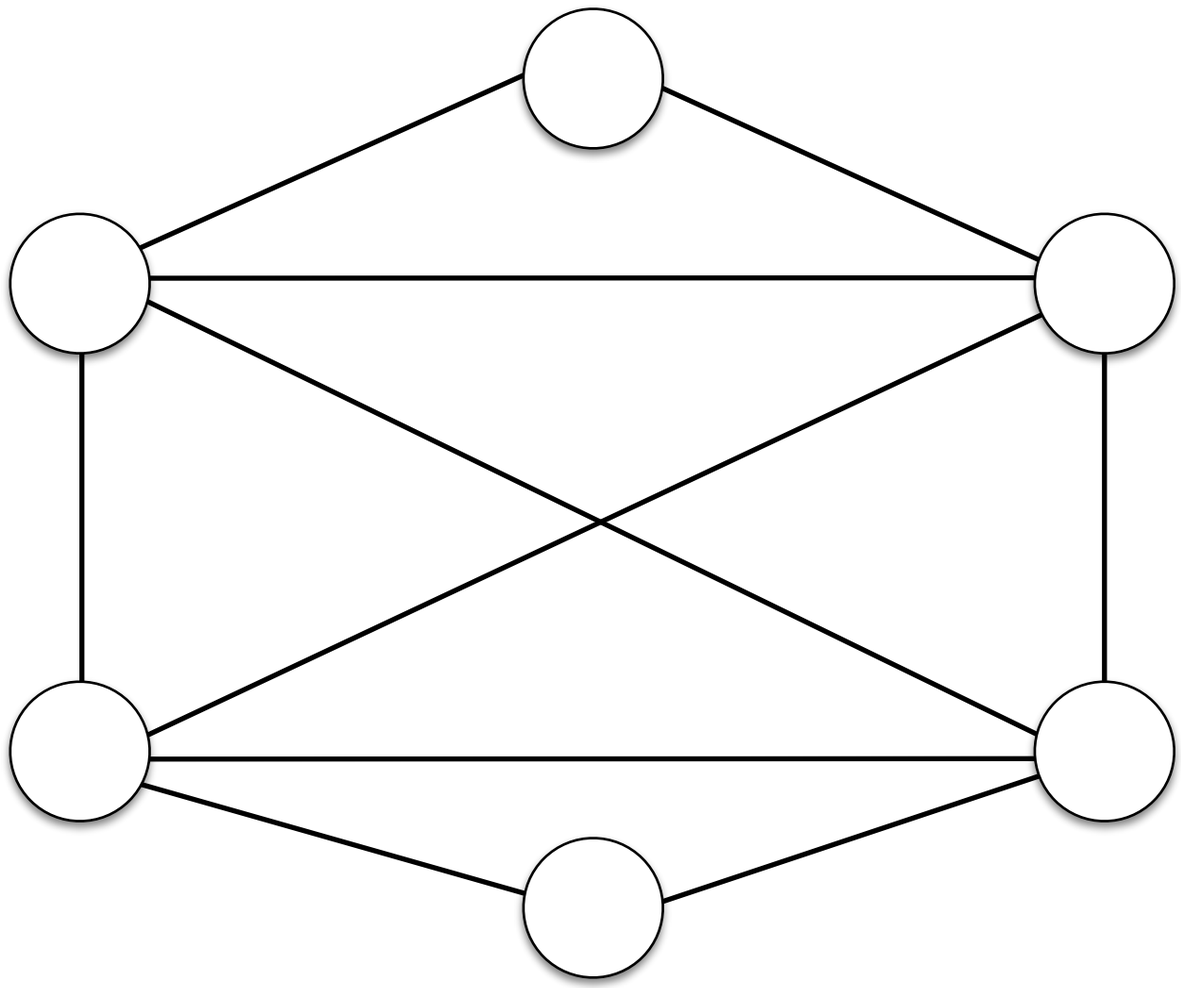
- An Euler Circuit is an Euler path that begins and ends at the same node.



This graph does NOT have an Euler Circuit

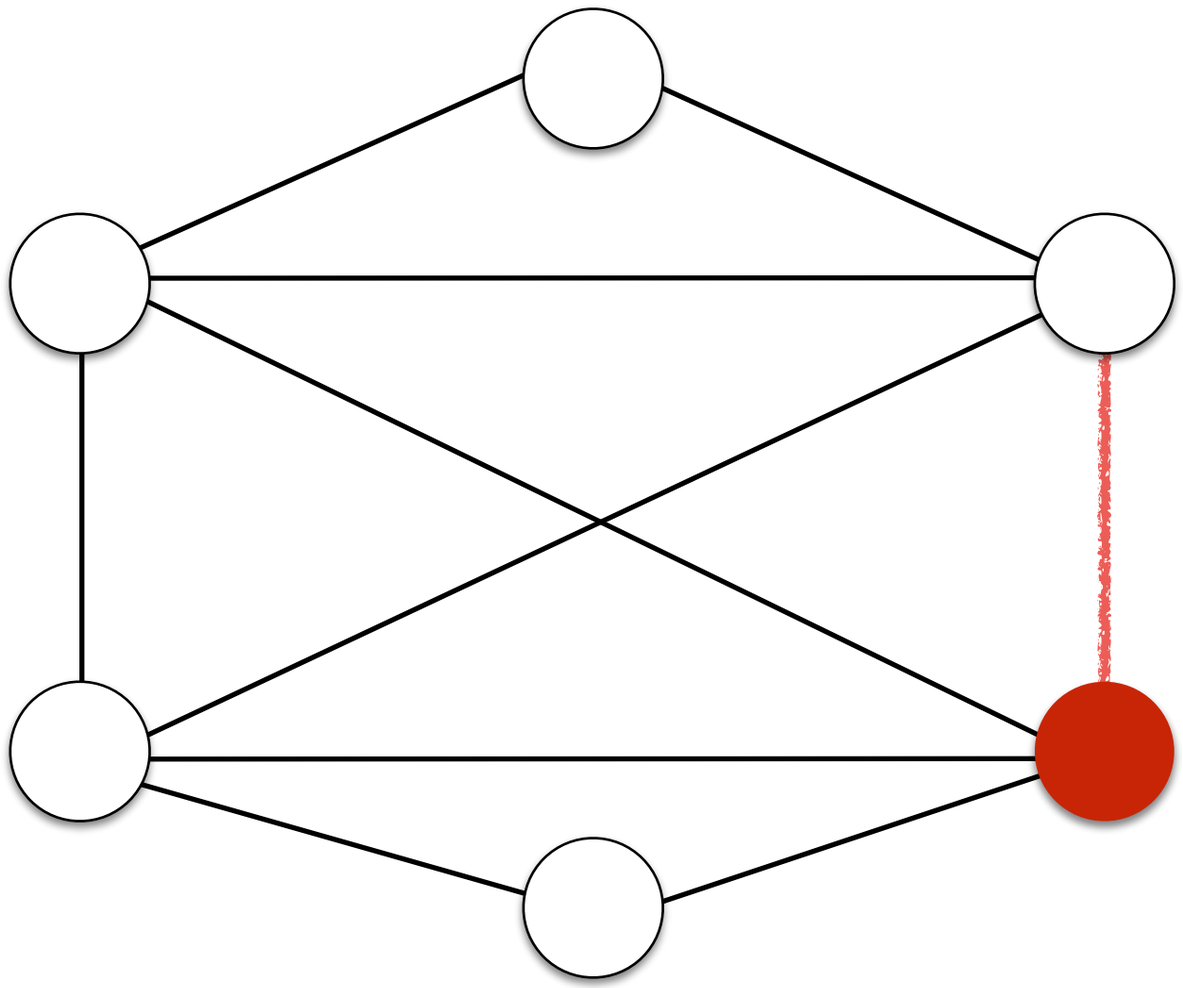
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



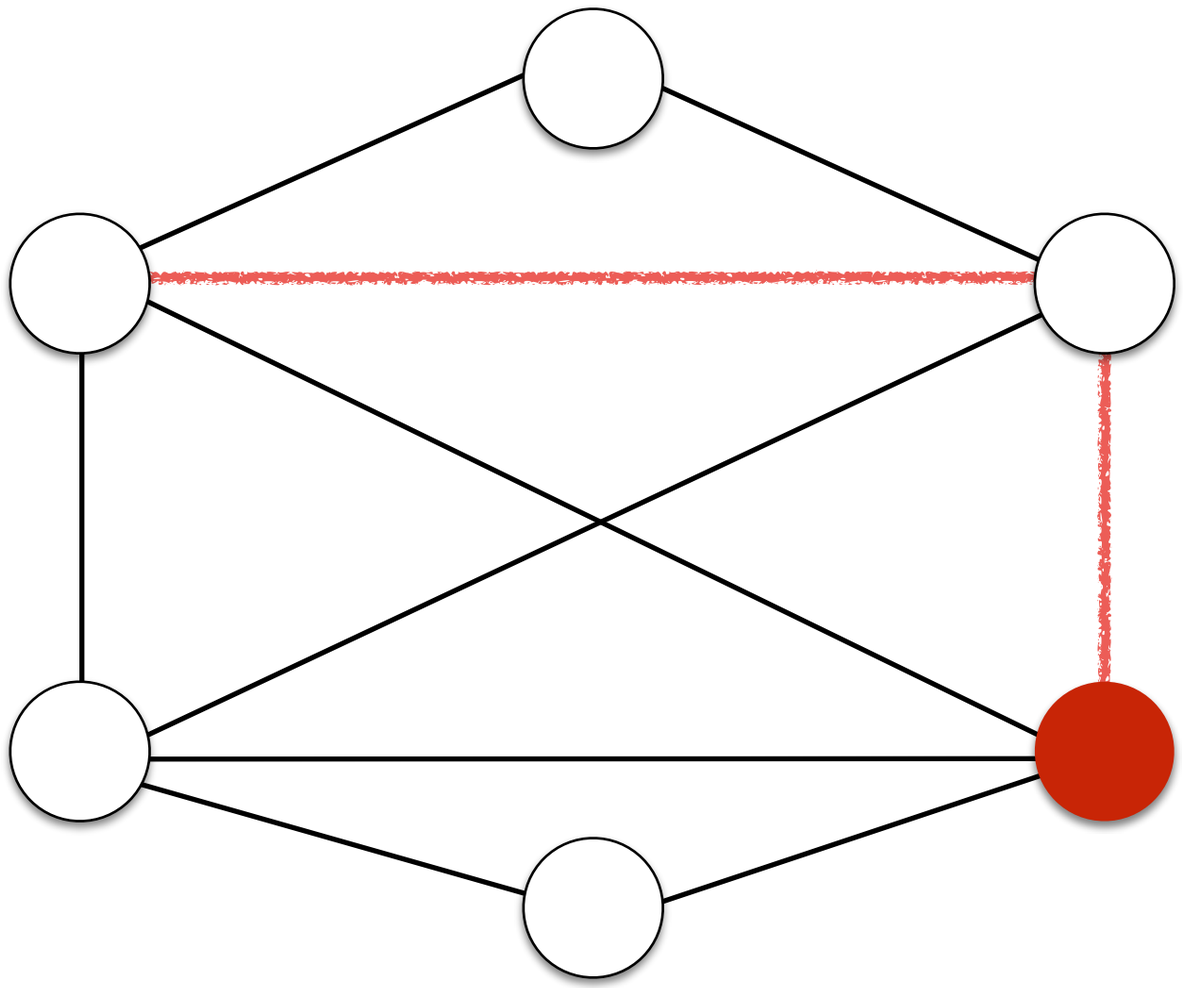
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



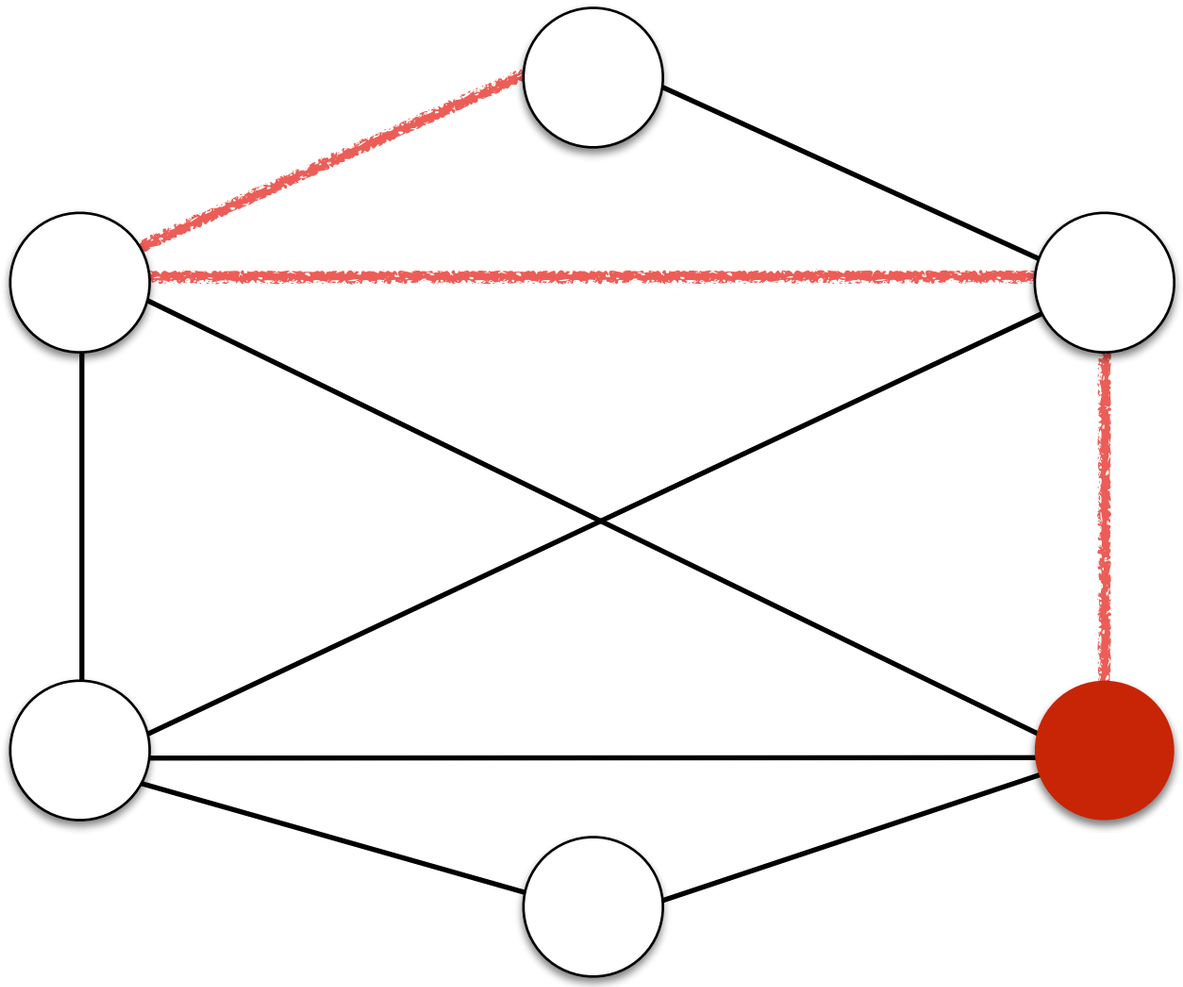
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



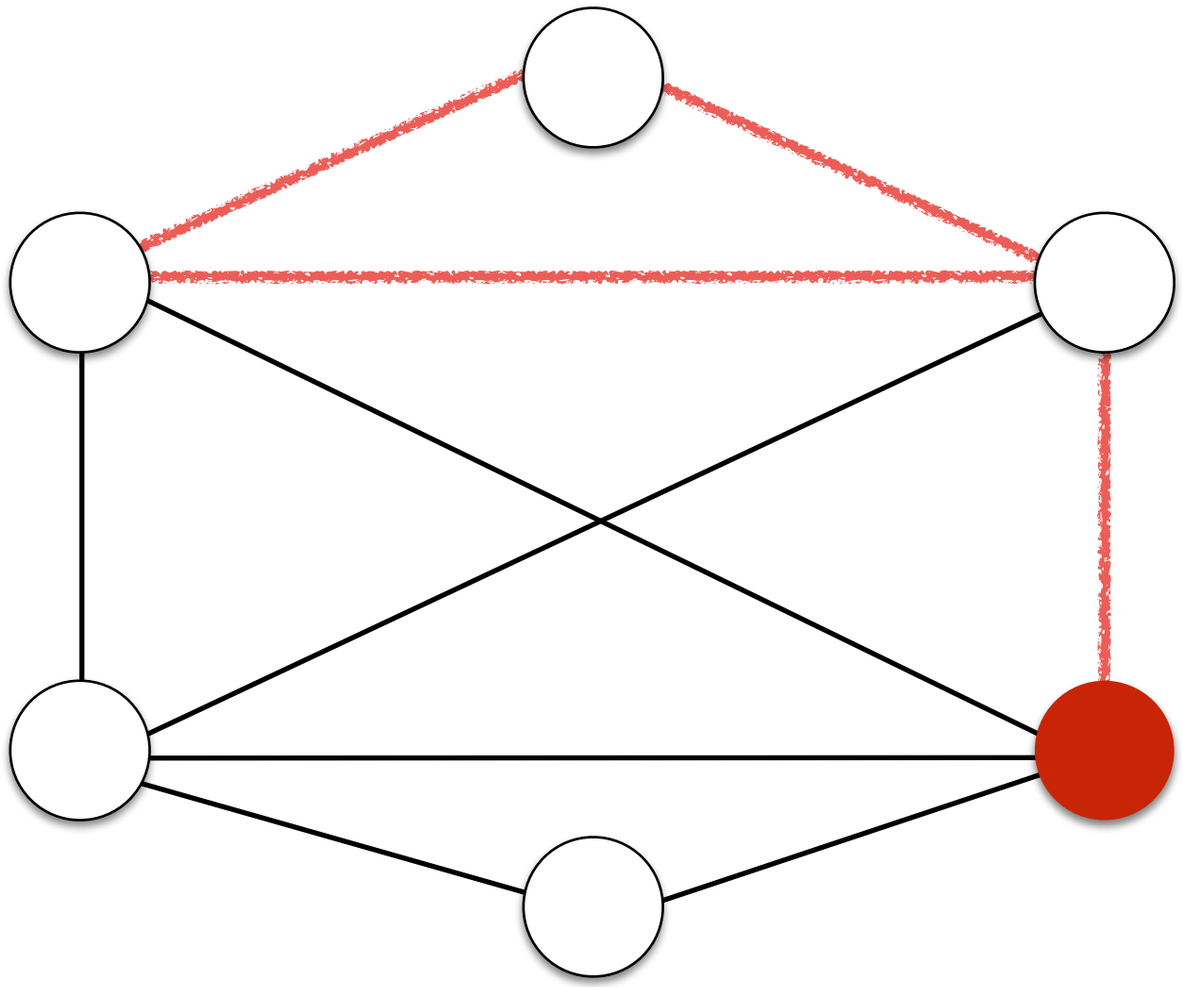
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



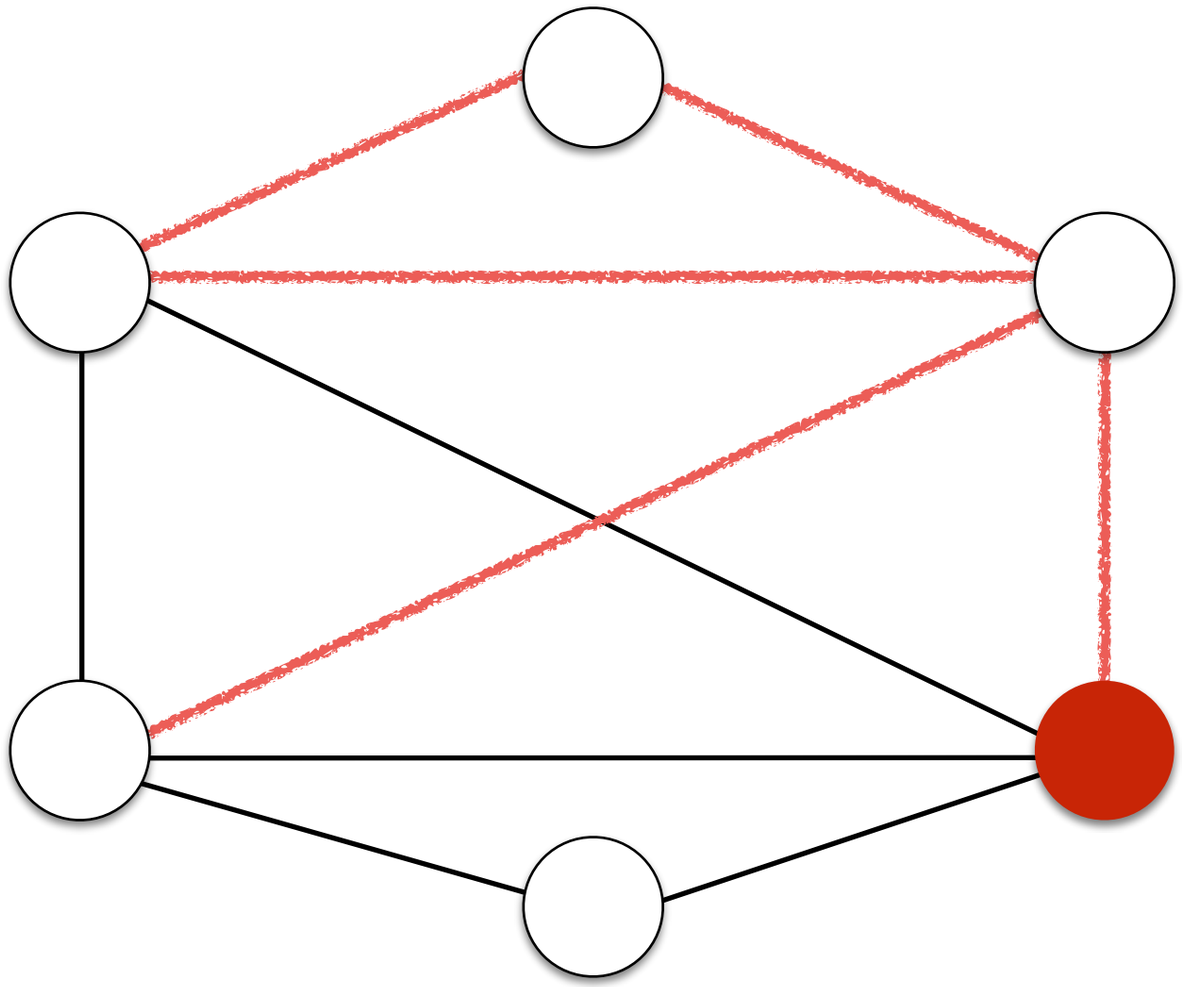
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



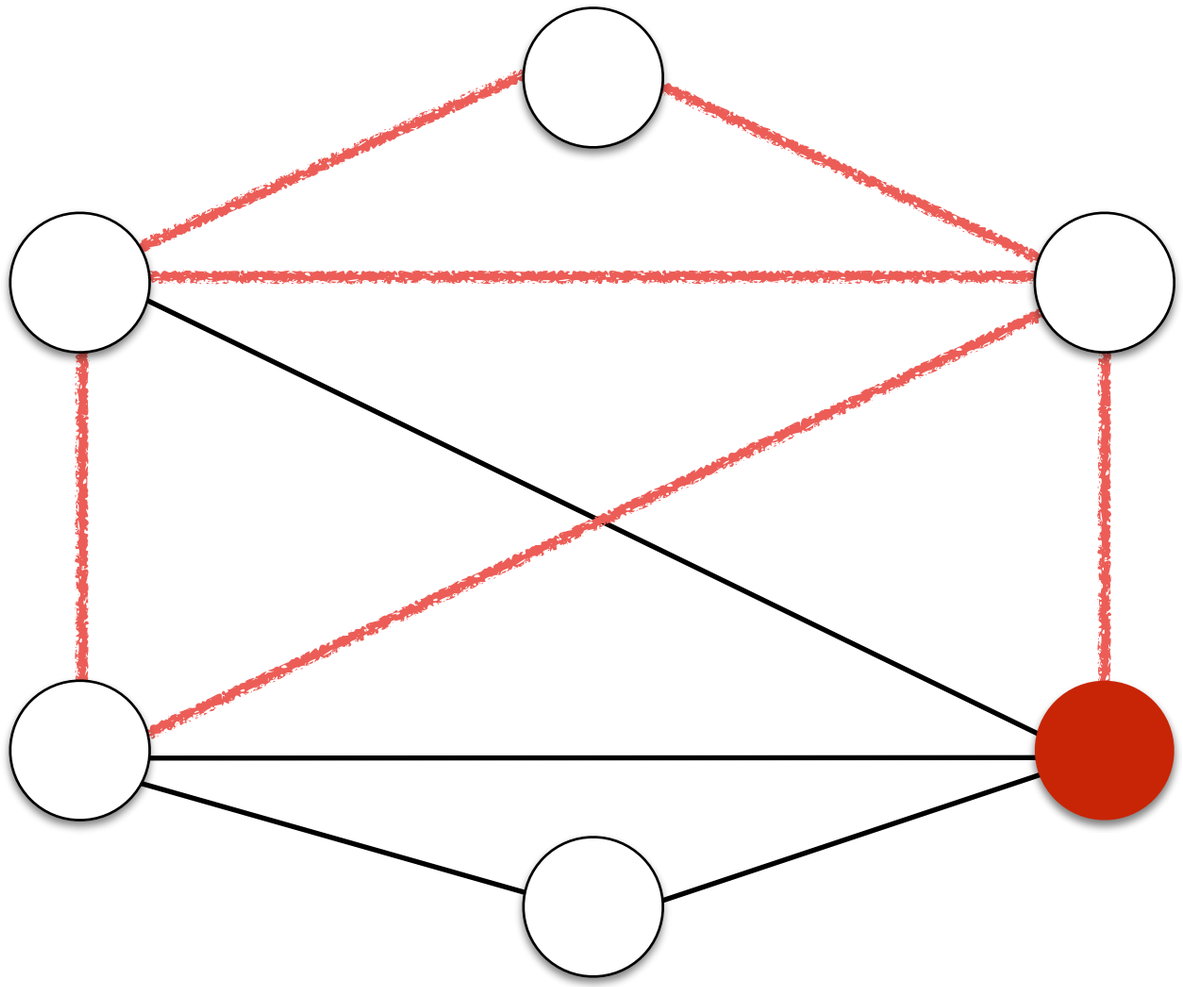
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



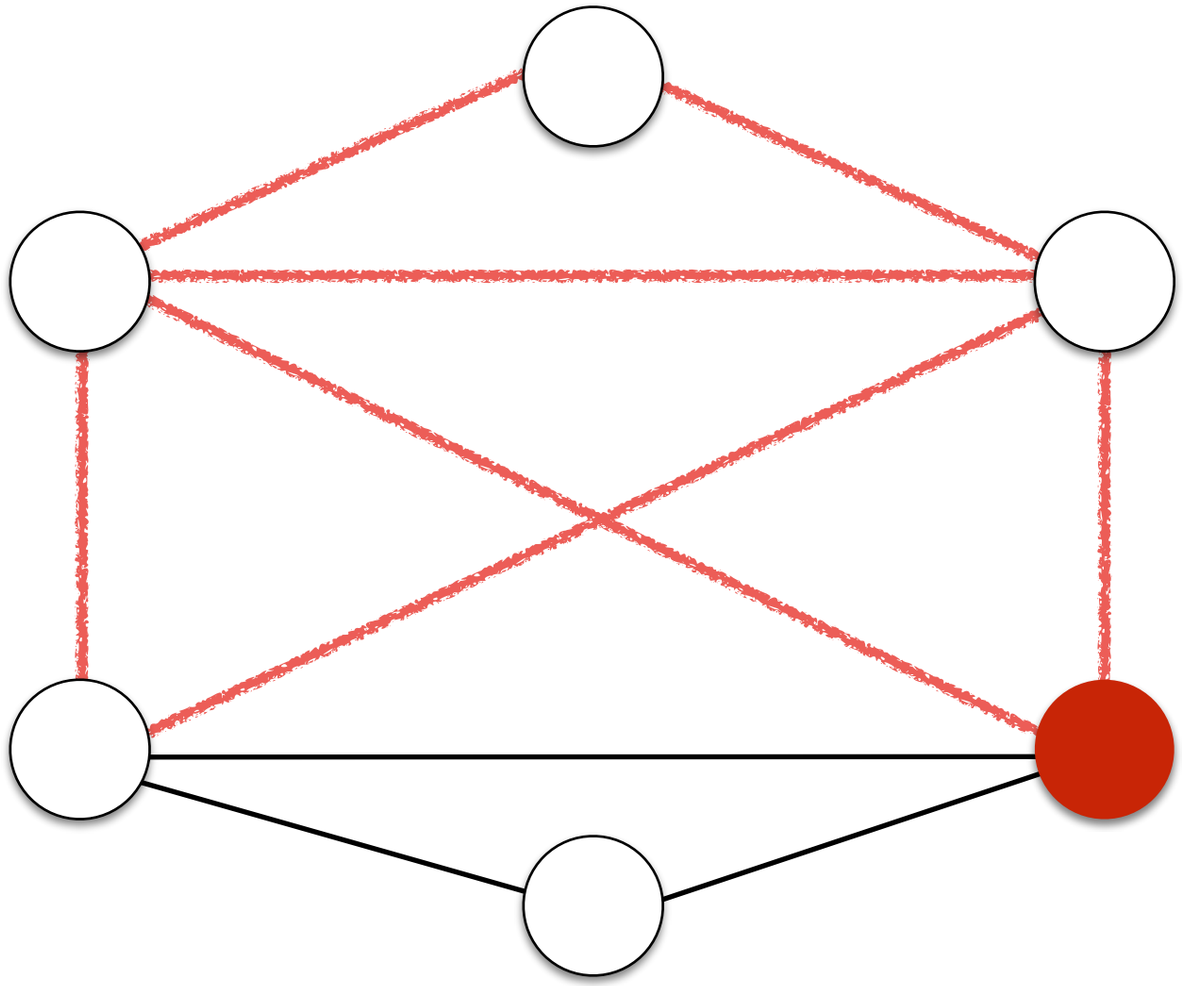
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



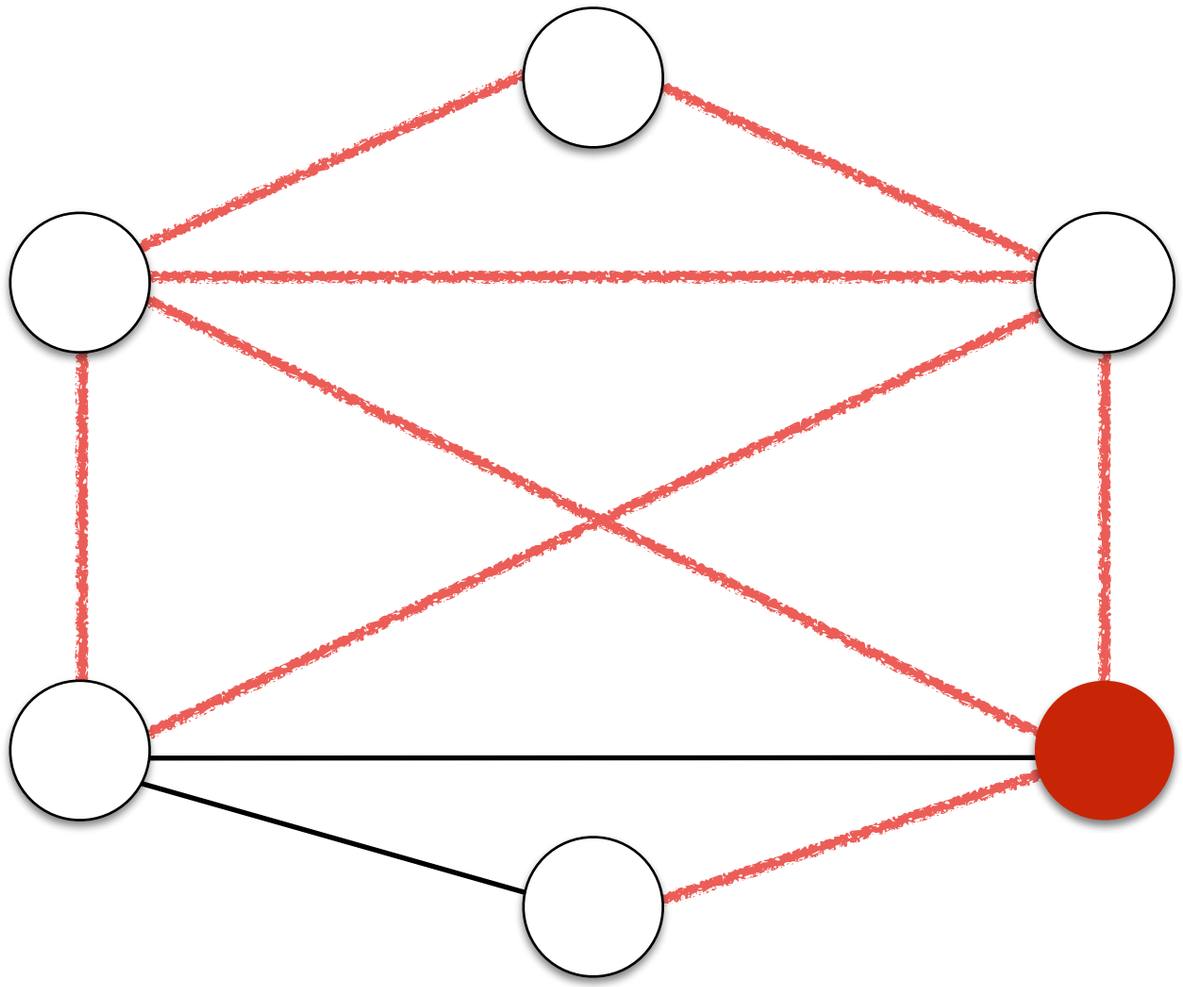
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



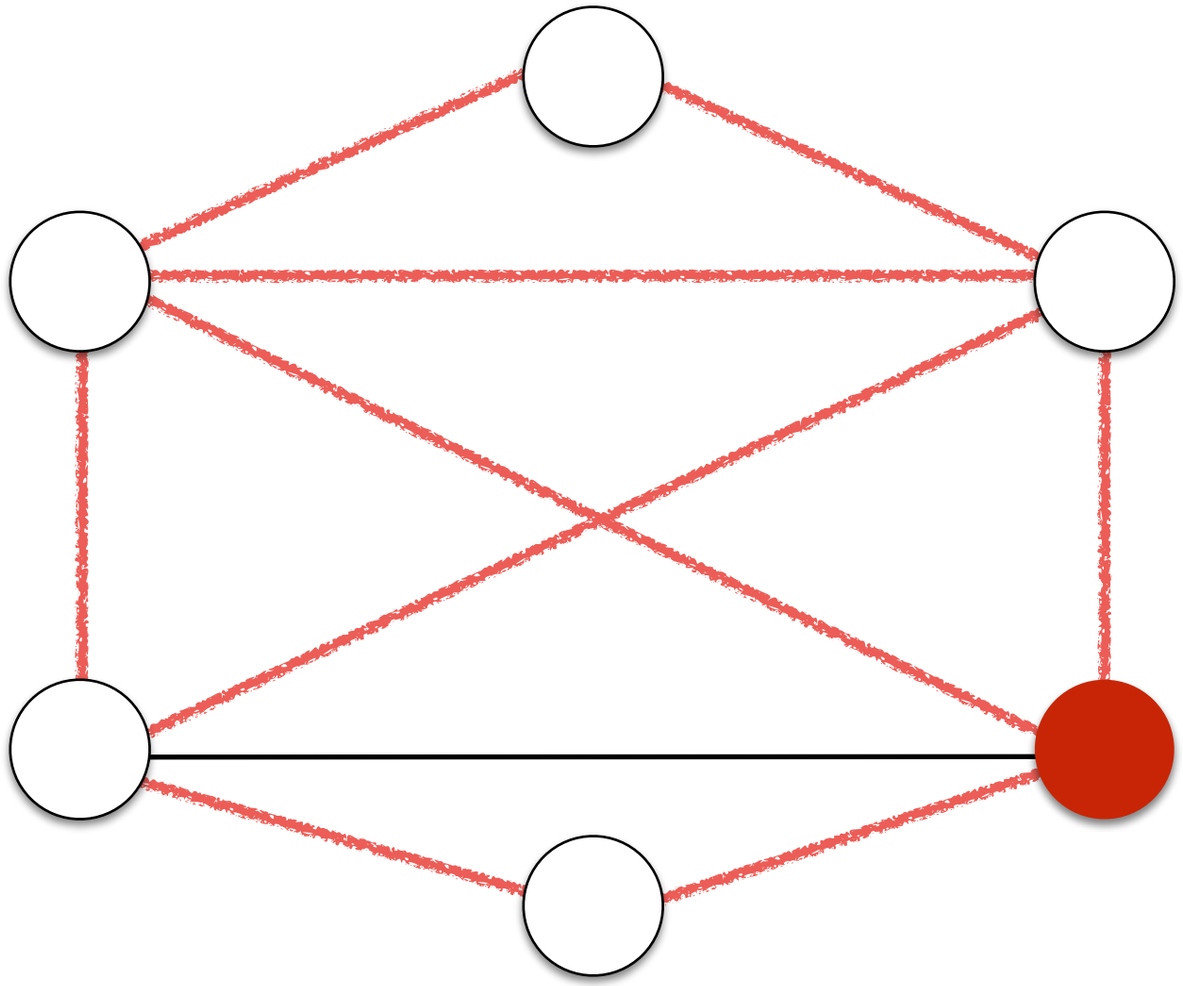
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



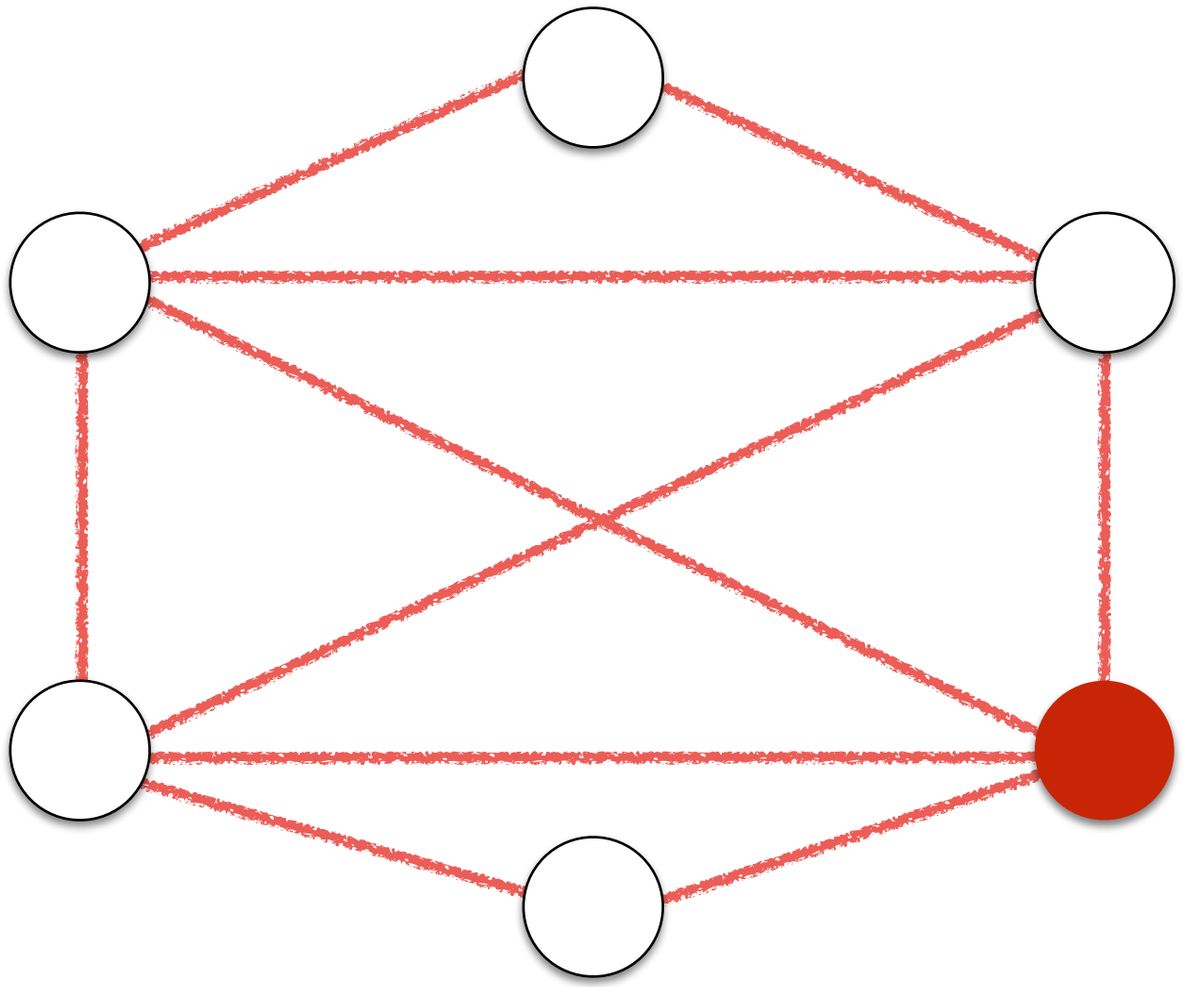
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



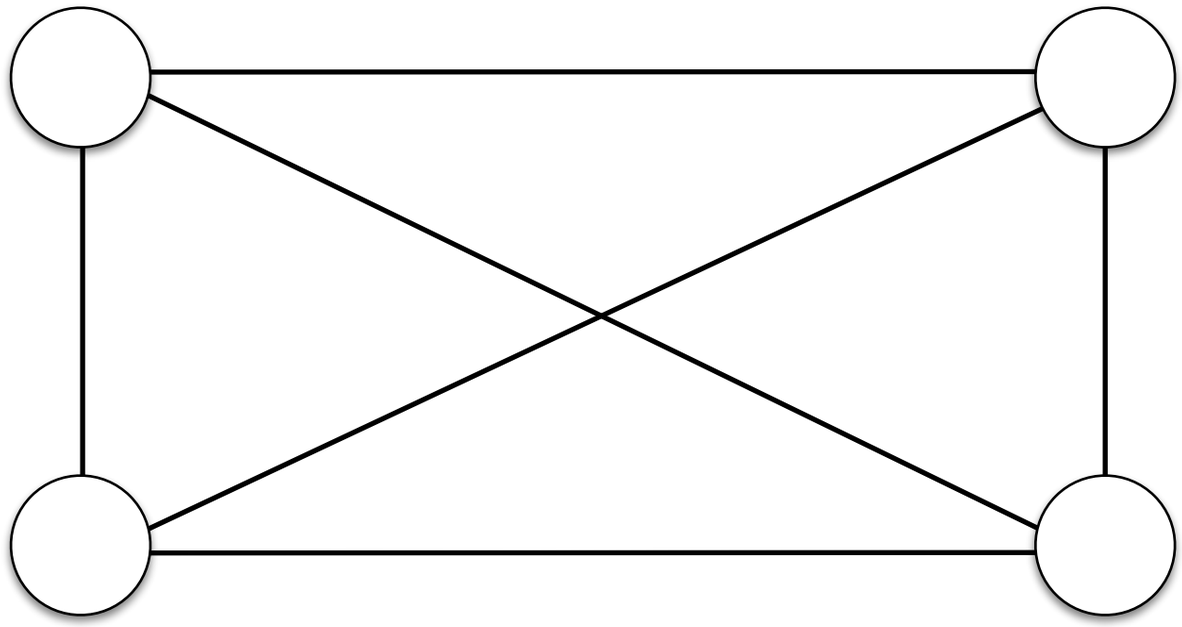
Euler Circuit

- An Euler Circuit is an Euler path that begins and ends at the same node.



Euler Circuit

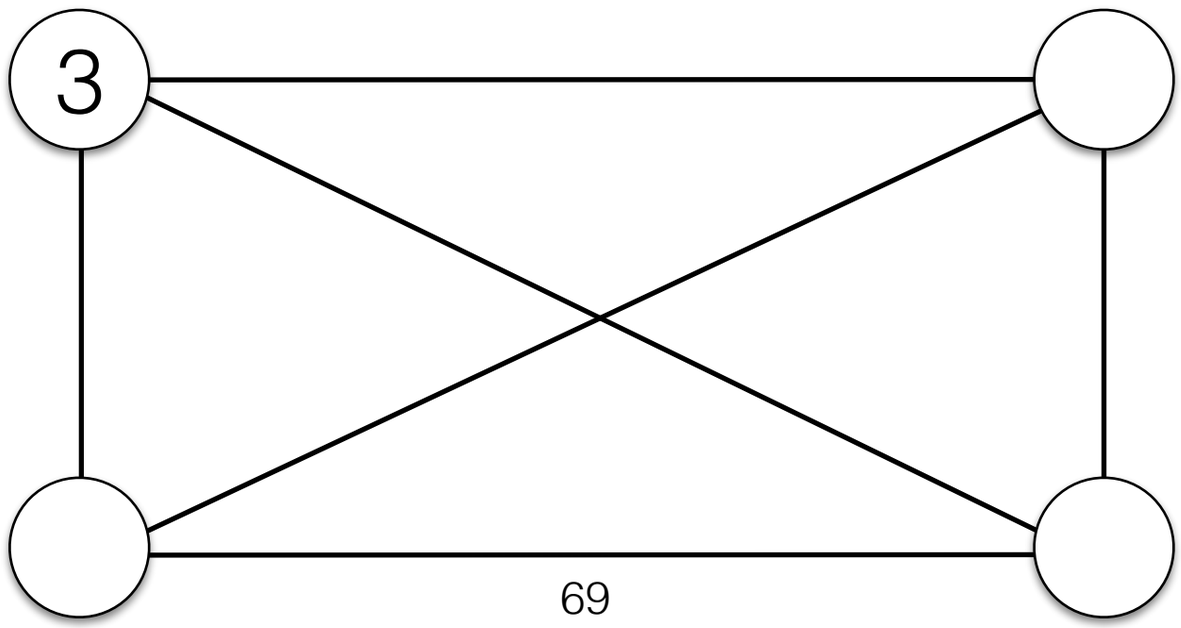
- An Euler Circuit is an Euler path that begins and ends at the same node.



This graph does not have an Euler Path.

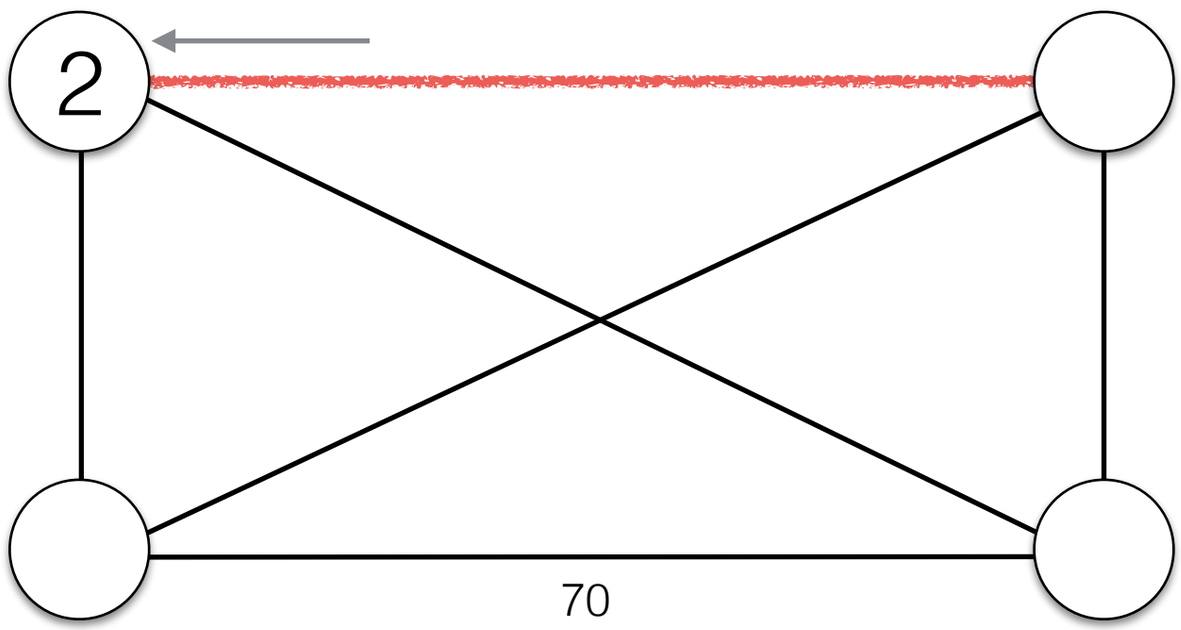
Necessary Condition for Euler Circuits

- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



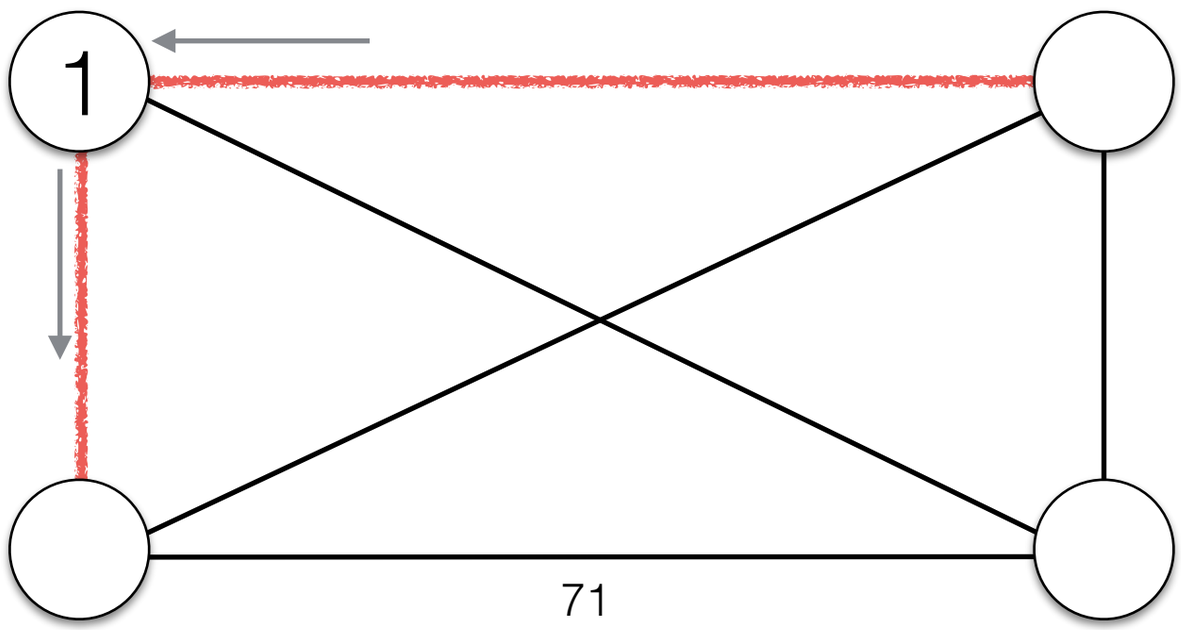
Necessary Condition for Euler Circuits

- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



Necessary Condition for Euler Circuits

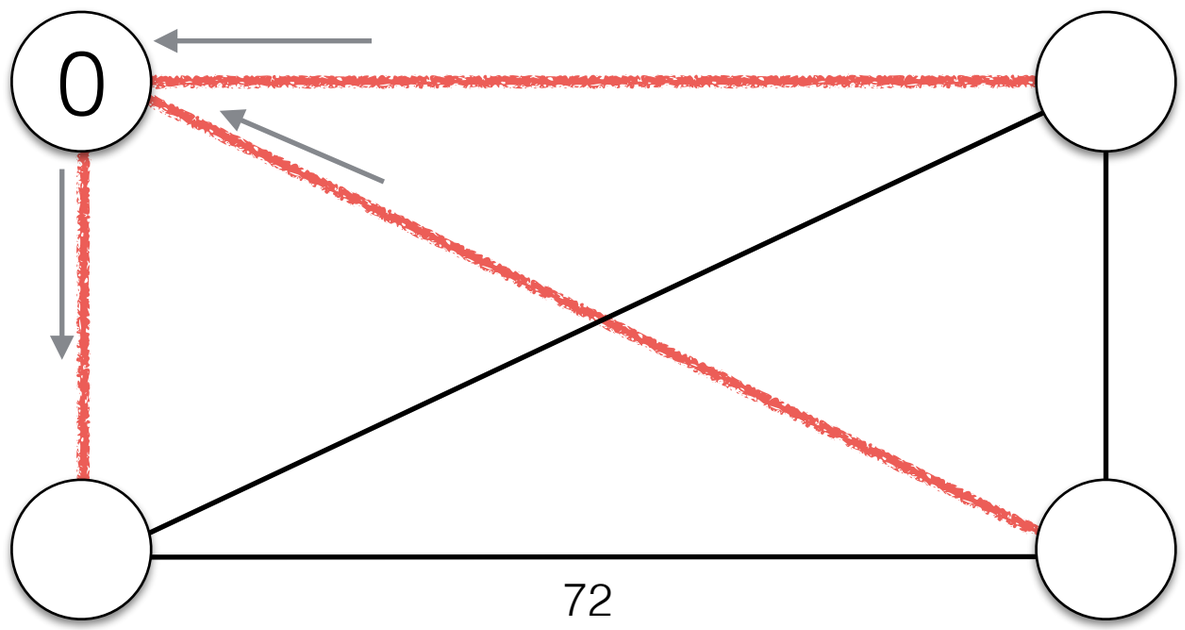
- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!



Necessary Condition for Euler Circuits

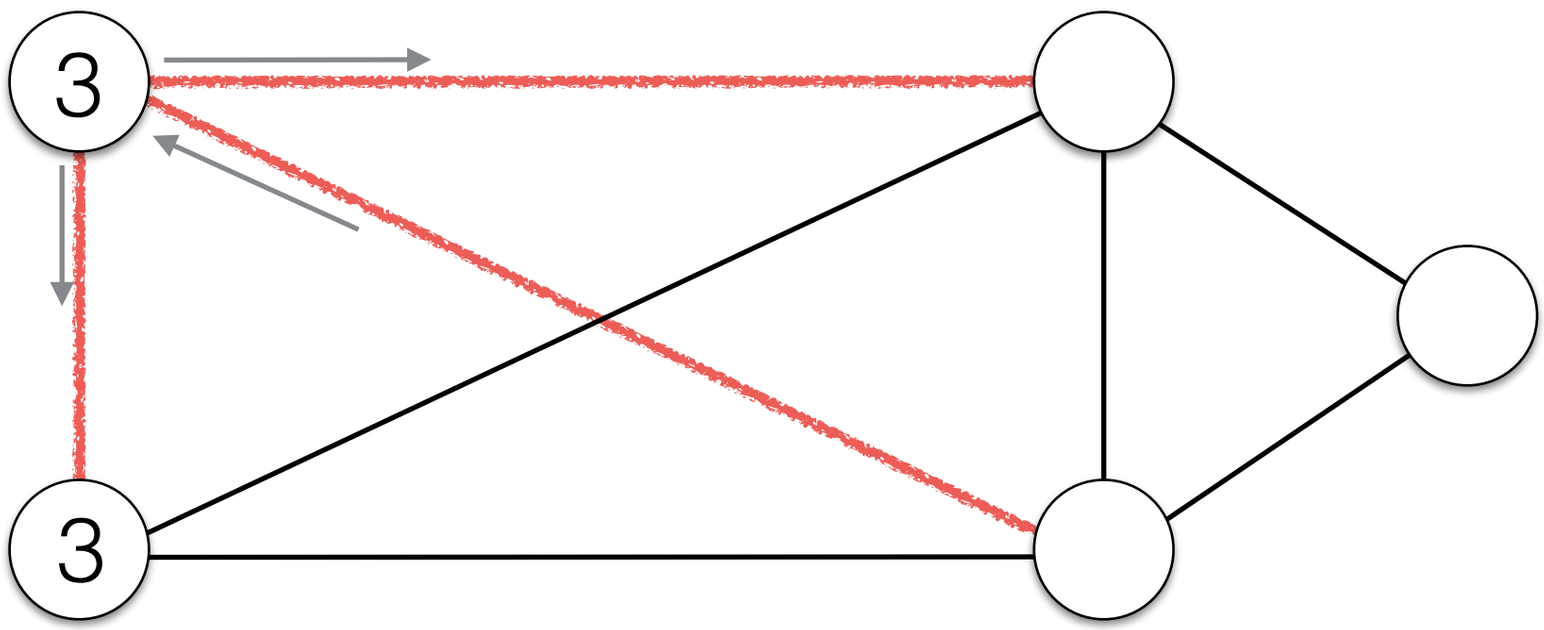
- Observation:
  - Once we enter v, we need another edge to leave it.
  - If v has an odd number of edges, the last time we enter v, we will be stuck!

- For an Euler Circuit to exist in a graph, all vertices need to have even degree (even number of edges).



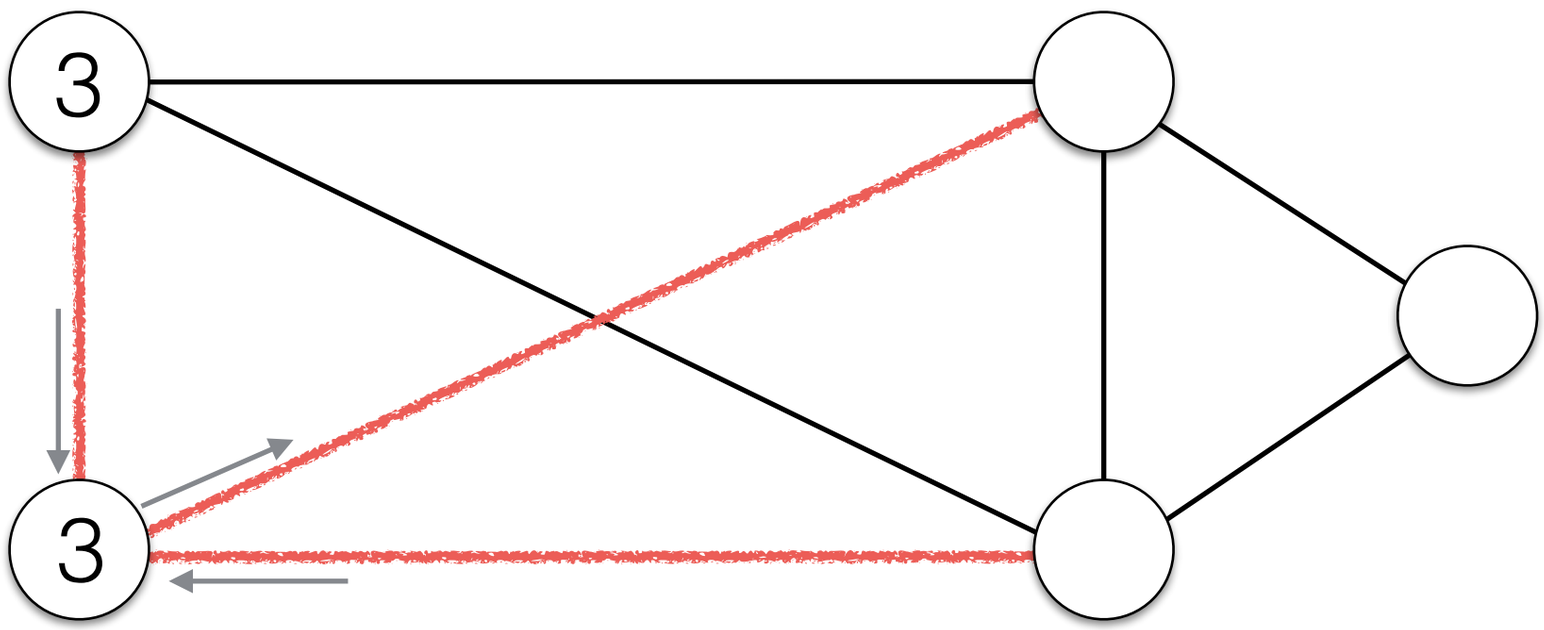
Necessary Condition for Euler Paths

- For an Euler Path to exist in a graph, exactly 0 or 2 vertices must have odd degree.
    - Start with one of the odd vertices.
    - End in the other one.



Necessary Condition for Euler Paths

- For an Euler Path to exist in a graph, exactly 0 or 2 vertices must have odd degree.
    - Start with one of the odd vertices.
    - End in the other one.



Conditions for Euler Paths and Circuits

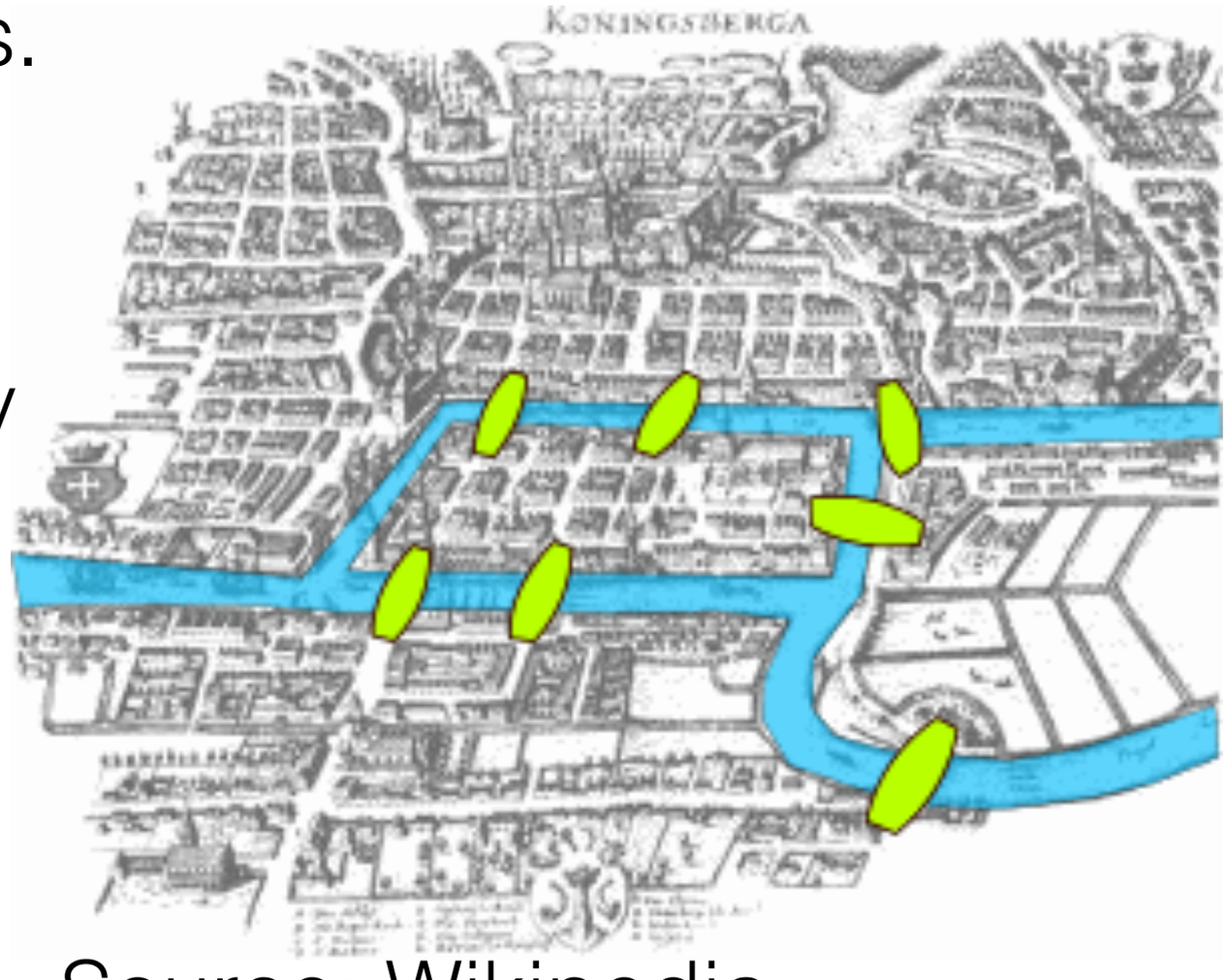
- For an Euler Circuit to exist in a graph, all vertices need to have even degree (even number of edges).
- For an Euler Path to exist in a graph, exactly 0 or 2 vertices need to have odd degree.

- These conditions are also sufficient!
  (i.e. every graph that contains only vertices of even degree has an Euler circuit). *(Hierholzer, 1873)*

Seven Bridges of Königsberg

Leonhard Euler, 1735

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges.

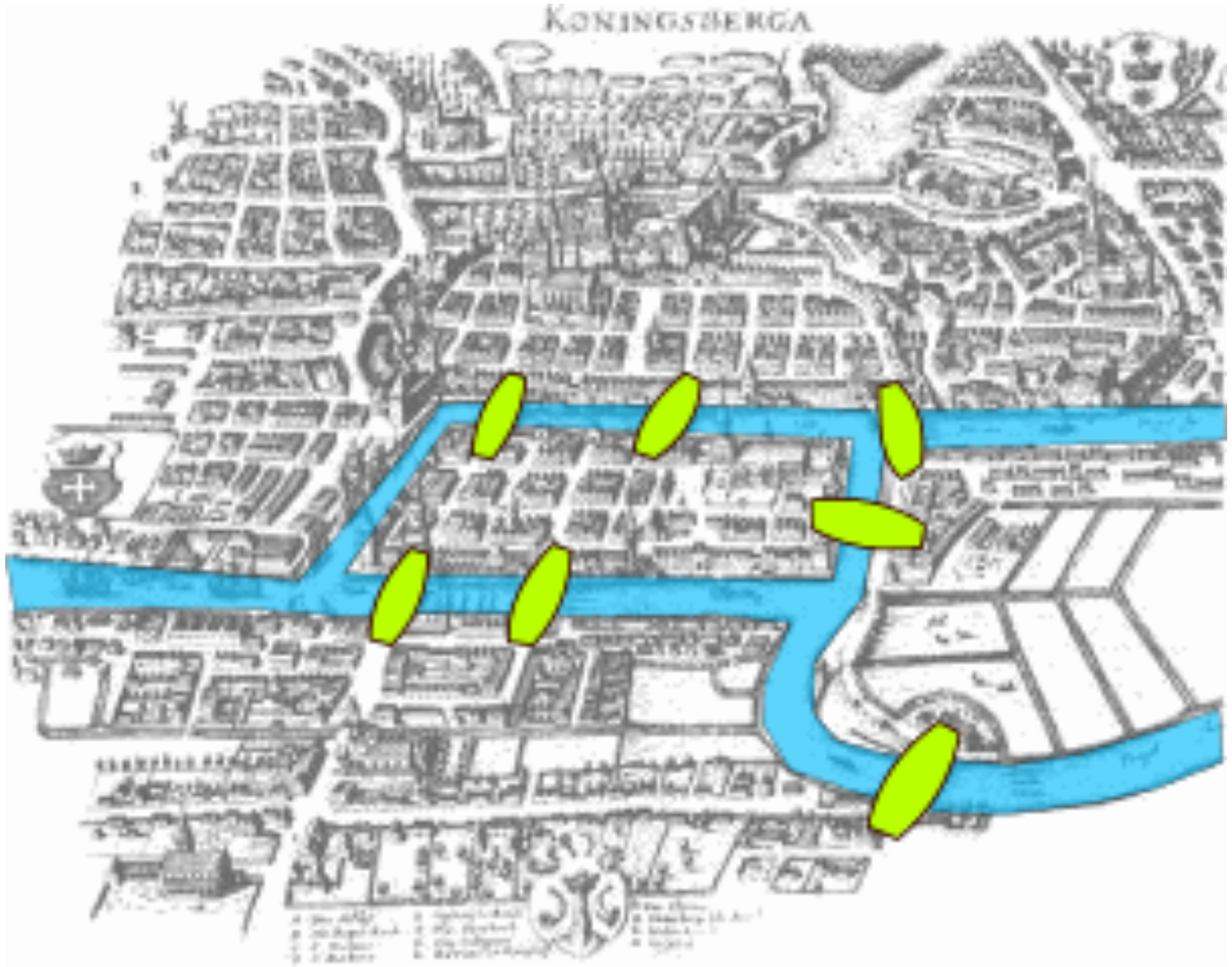


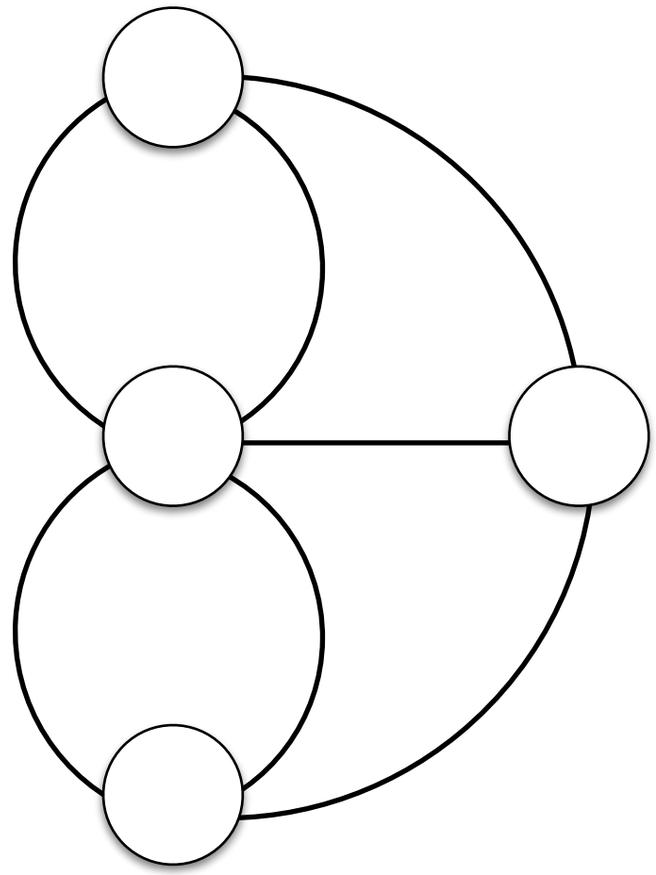
Euler's Problem:
Find a walk through the city that crosses every bridge exactly once!

Source: Wikipedia

Seven Bridges of Königsberg

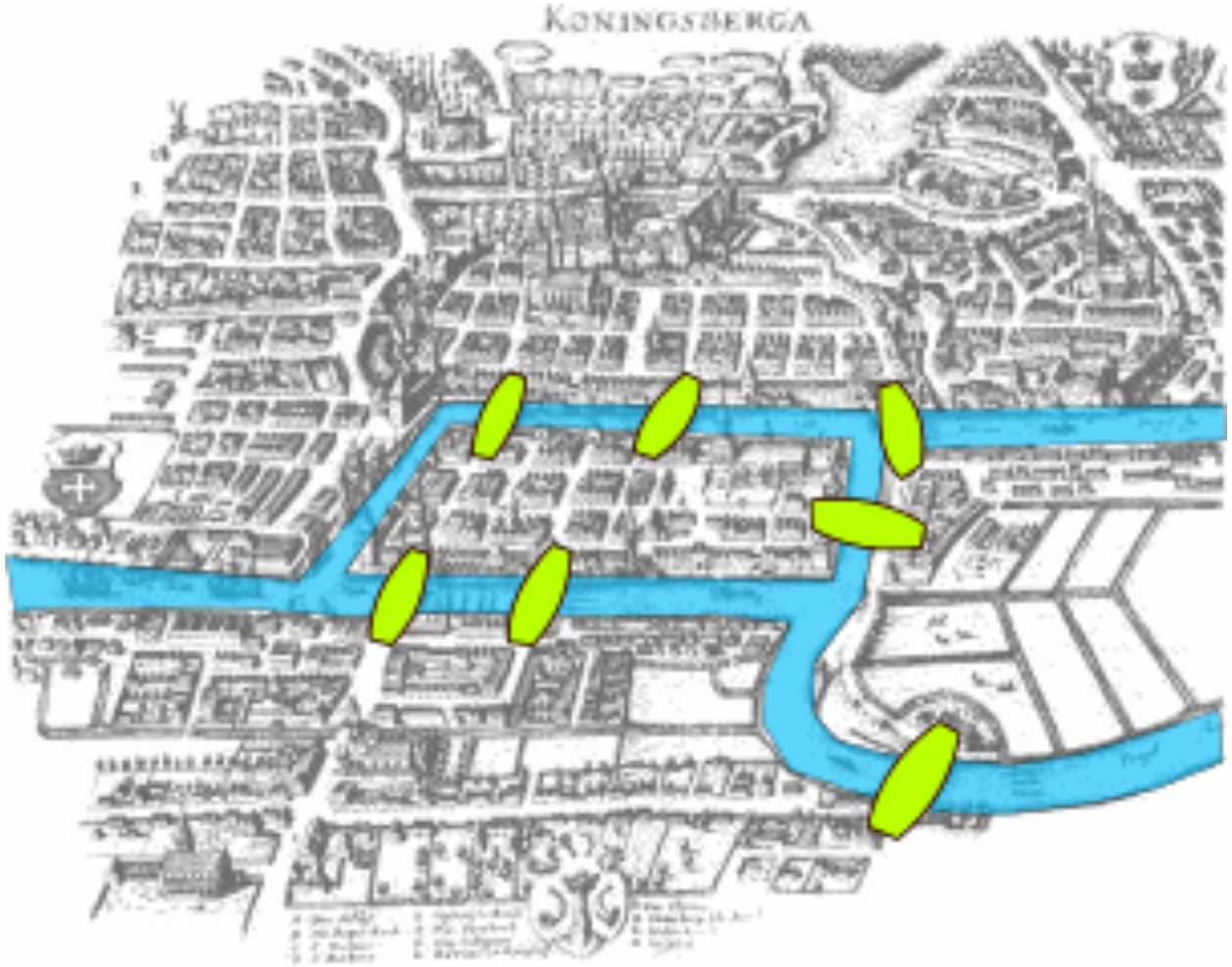
Leonhard Euler, 1735

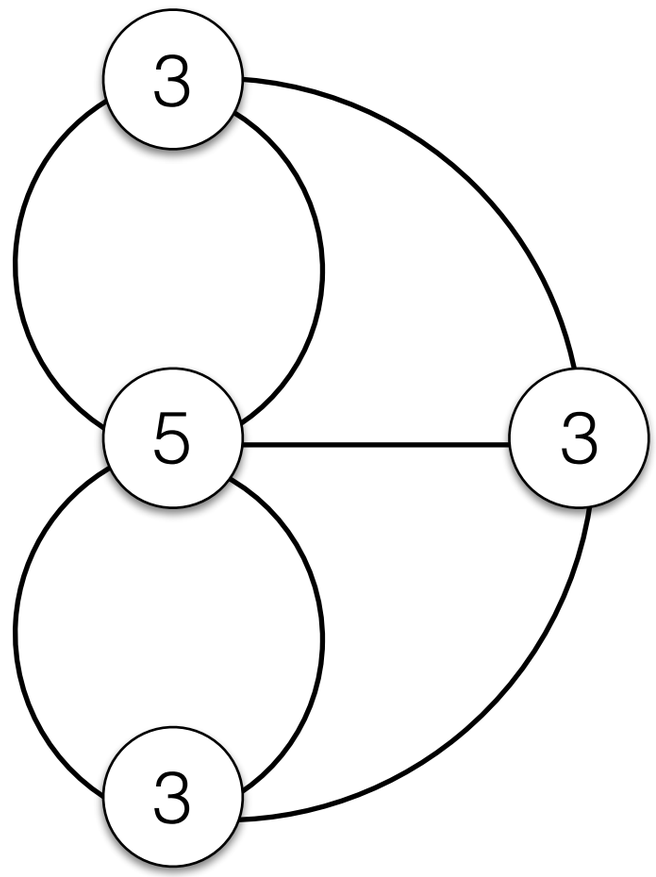




Seven Bridges of Königsberg

Leonhard Euler, 1735





There is no Euler Path in this graph.

Finding Euler Circuits

Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.

Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.
    - Because all vertices have even degree, we are guaranteed to not get stuck before we arrive at *s* again.

Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.
    - Because all vertices have even degree, we are guaranteed to not get stuck before we arrive at *s* again.
- While there are still edges in the graph that are not marked visited:

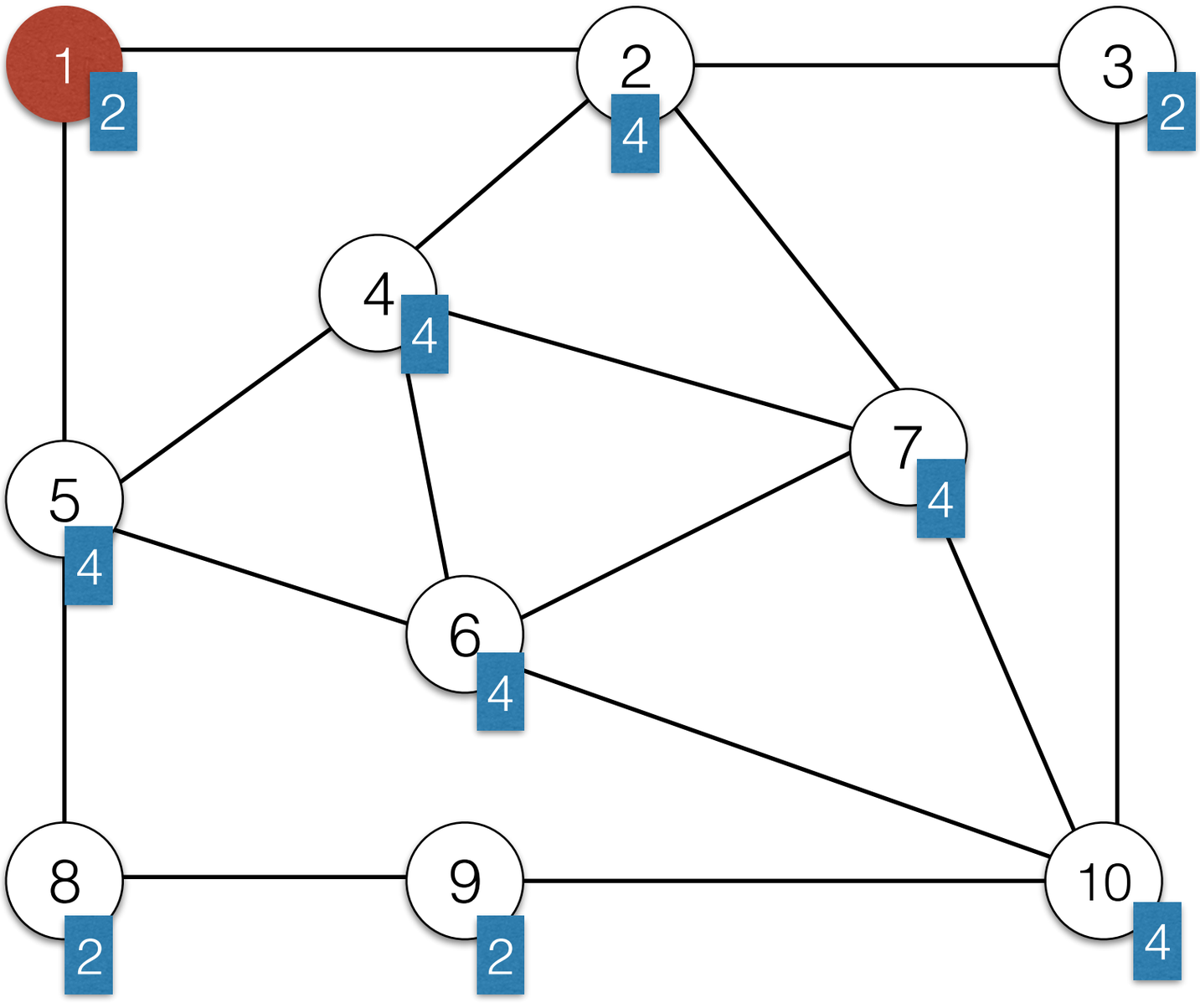
Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.
    - Because all vertices have even degree, we are guaranteed to not get stuck before we arrive at *s* again.
- While there are still edges in the graph that are not marked visited:
    - Find the first vertex v on the circuit that has unvisited edges.

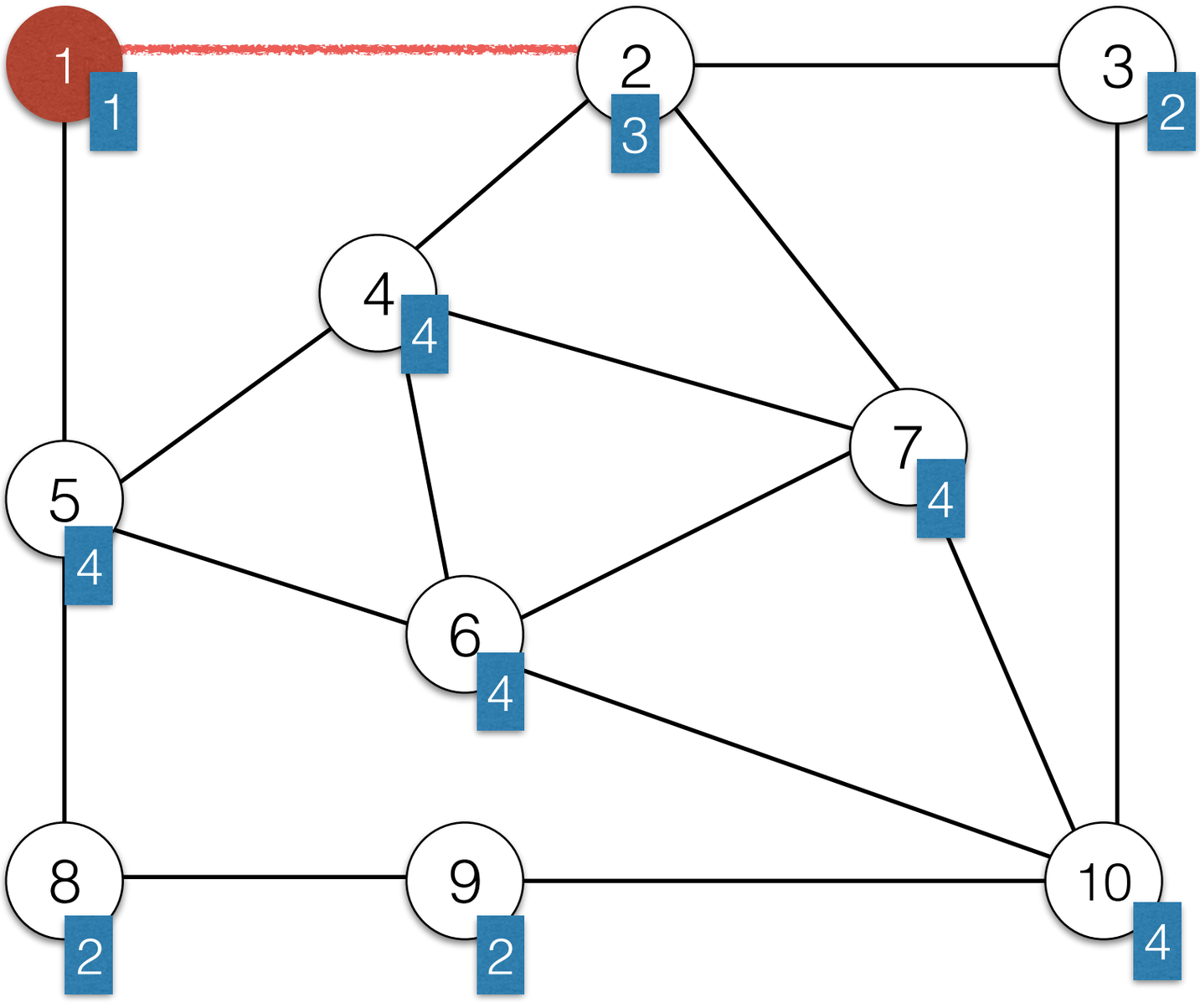
Finding Euler Circuits

- Start with any vertex *s*. First, using DFS find *any* circuit starting and ending in *s*. Mark all edges on the circuit as visited.
    - Because all vertices have even degree, we are guaranteed to not get stuck before we arrive at *s* again.
- While there are still edges in the graph that are not marked visited:
    - Find the first vertex v on the circuit that has unvisited edges.
    - Find a circuit starting in v and *splice* this path into the first circuit

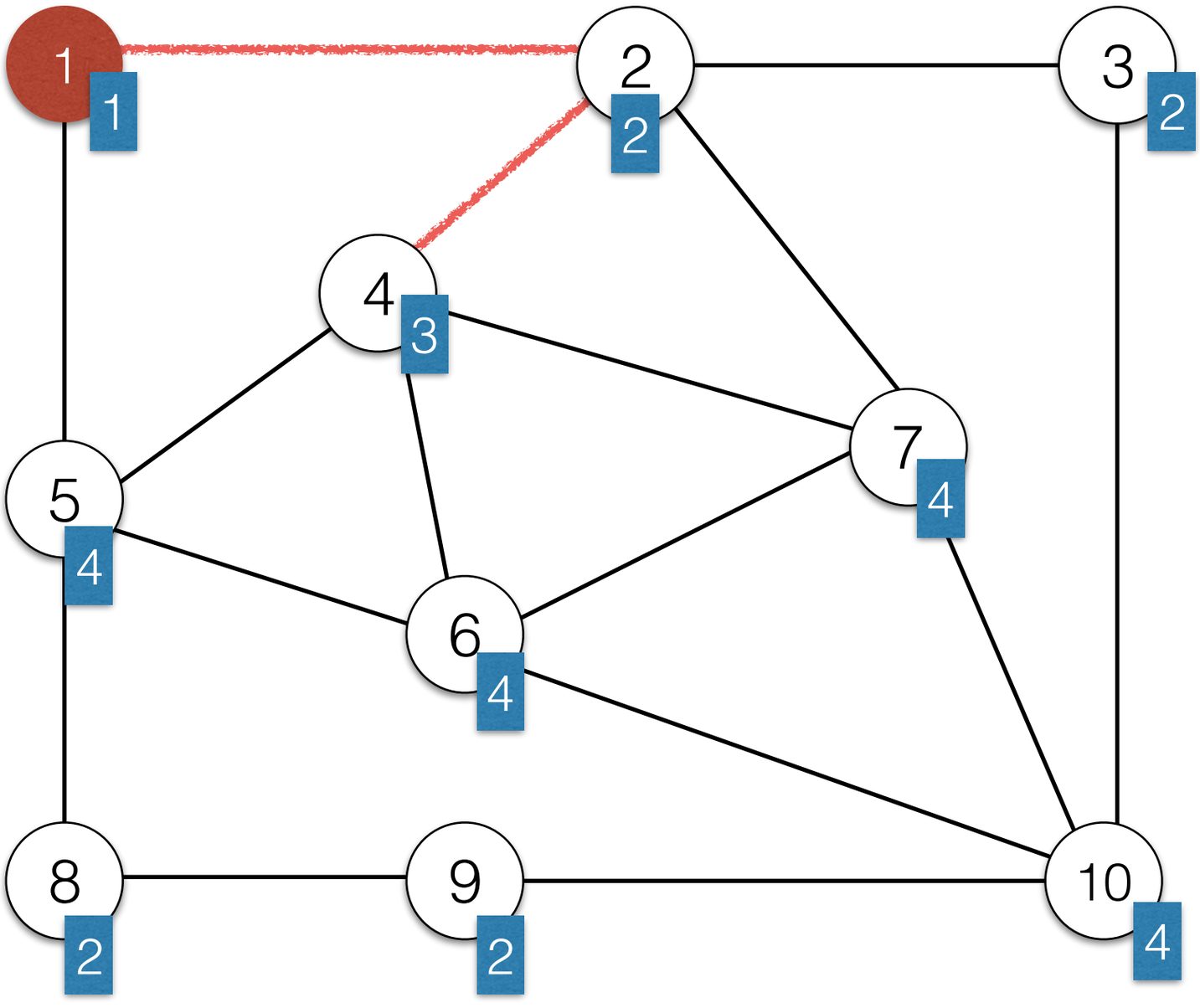
Finding Euler Circuits



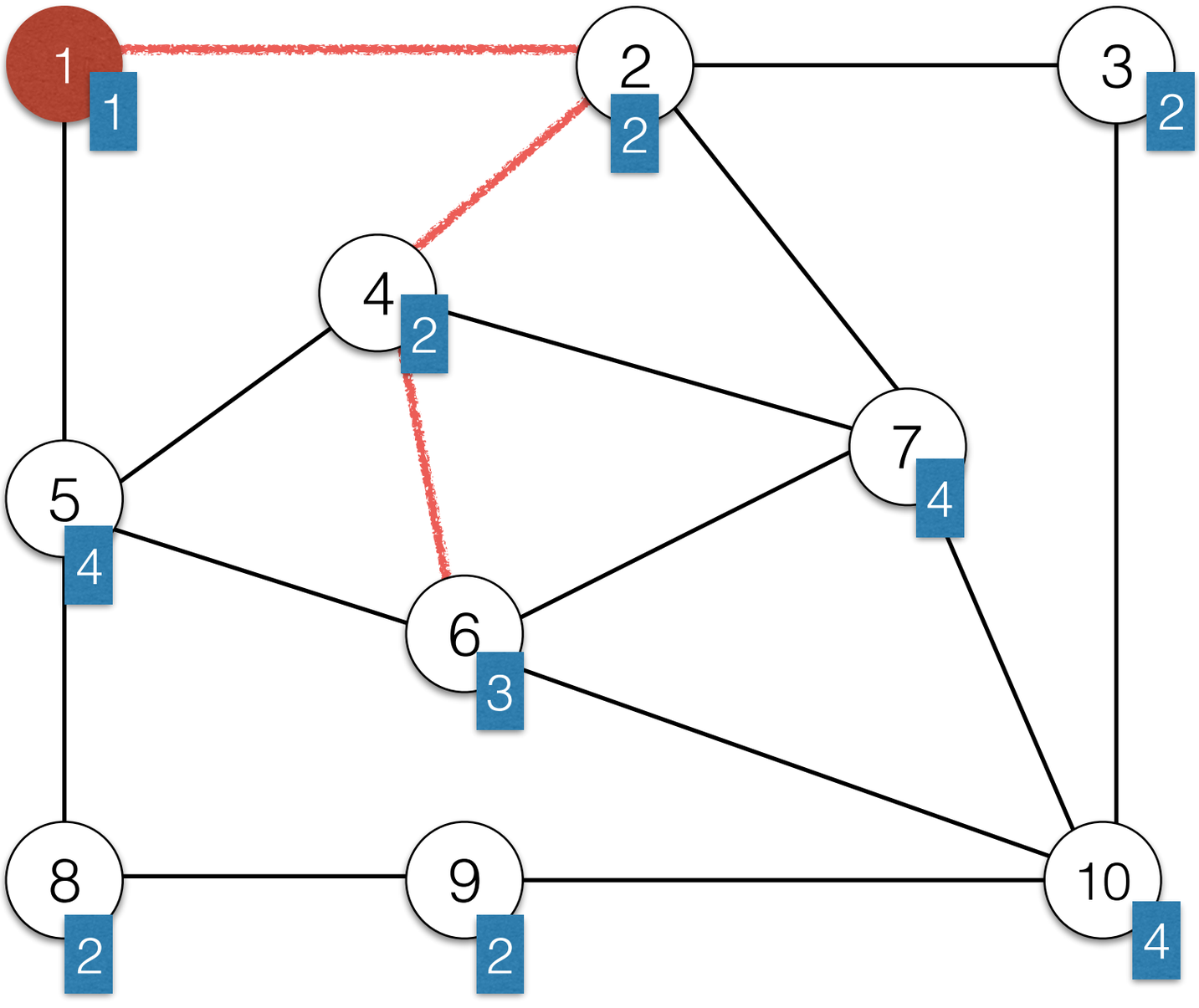
Finding Euler Circuits



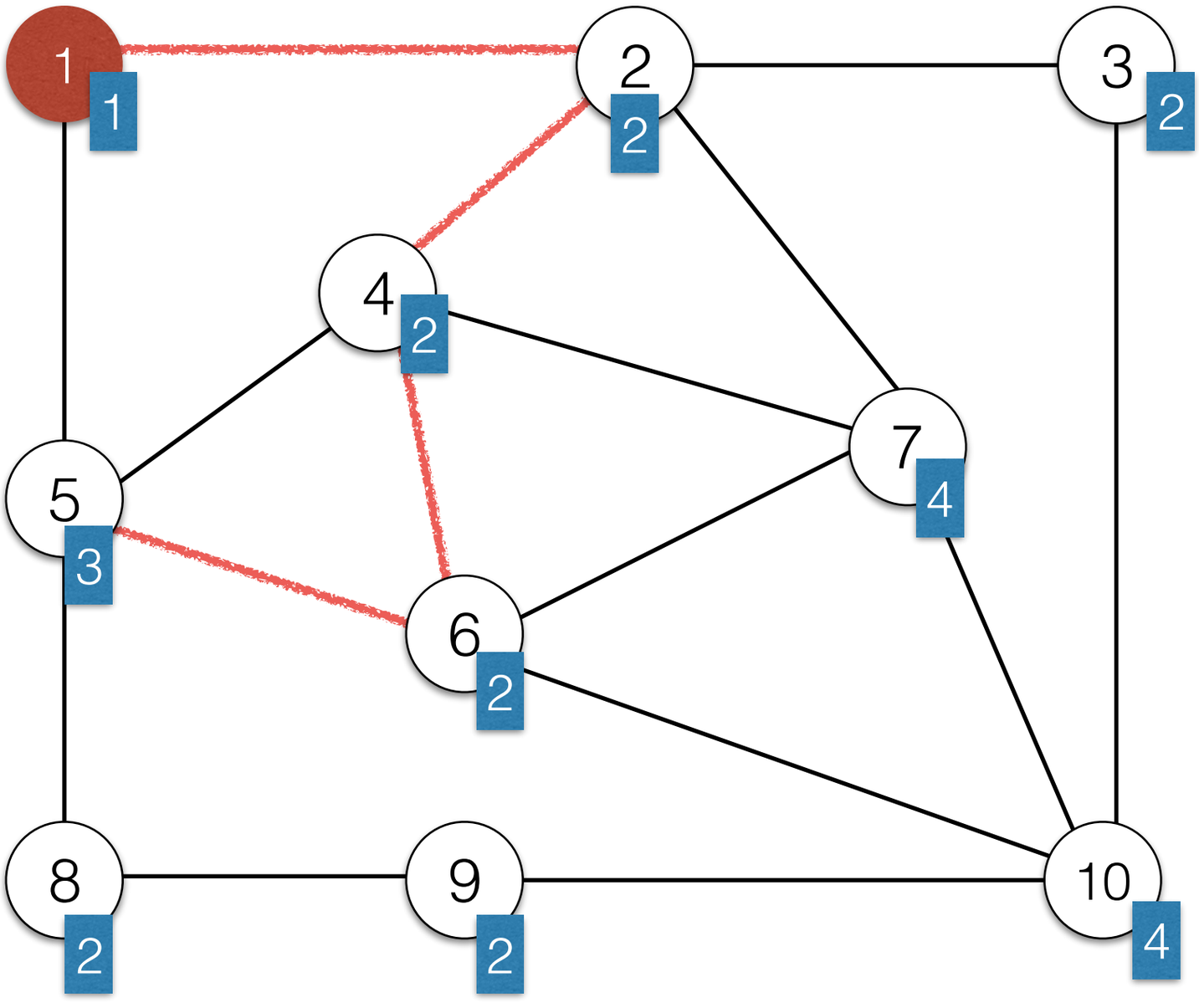
Finding Euler Circuits



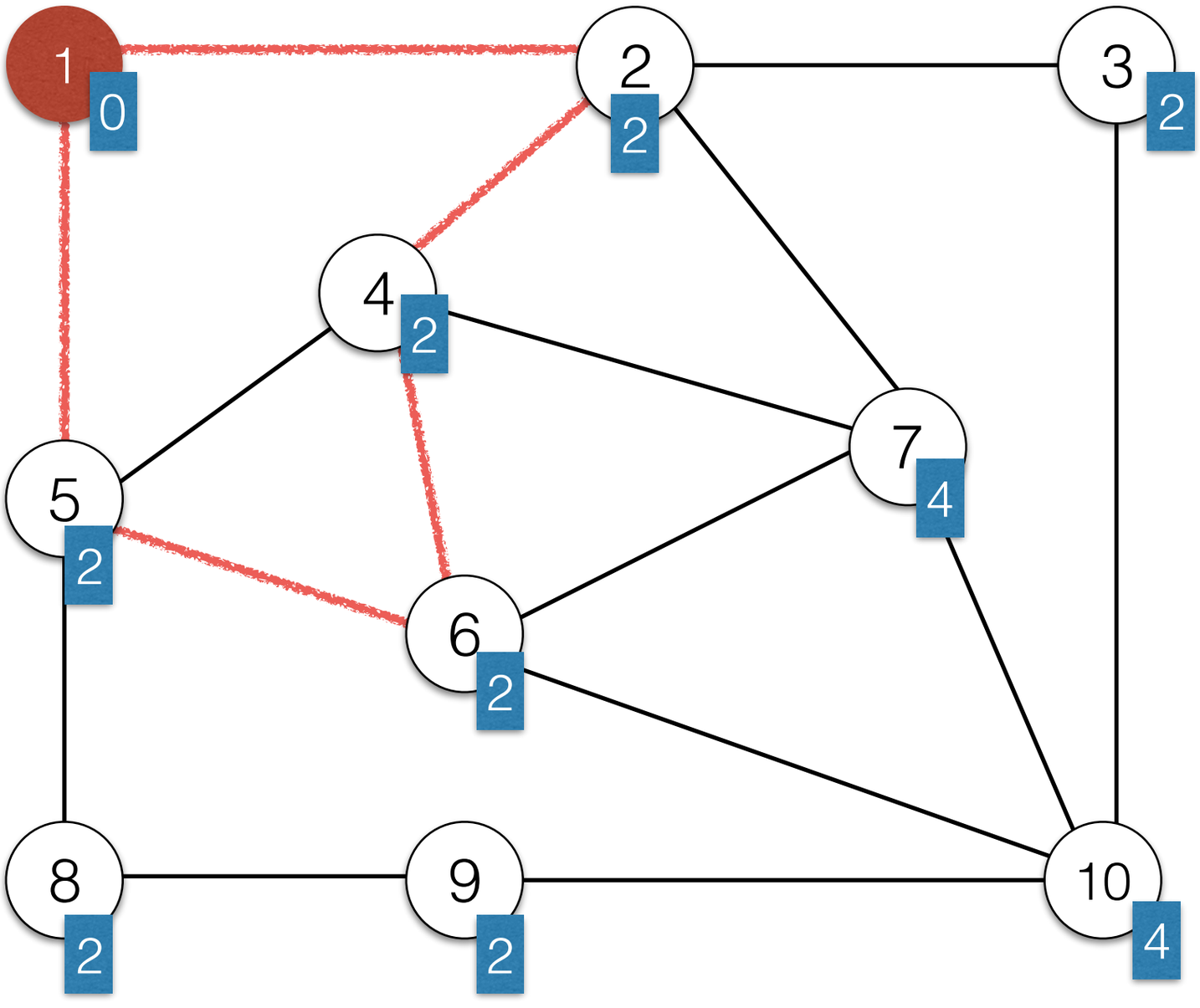
Finding Euler Circuits



Finding Euler Circuits

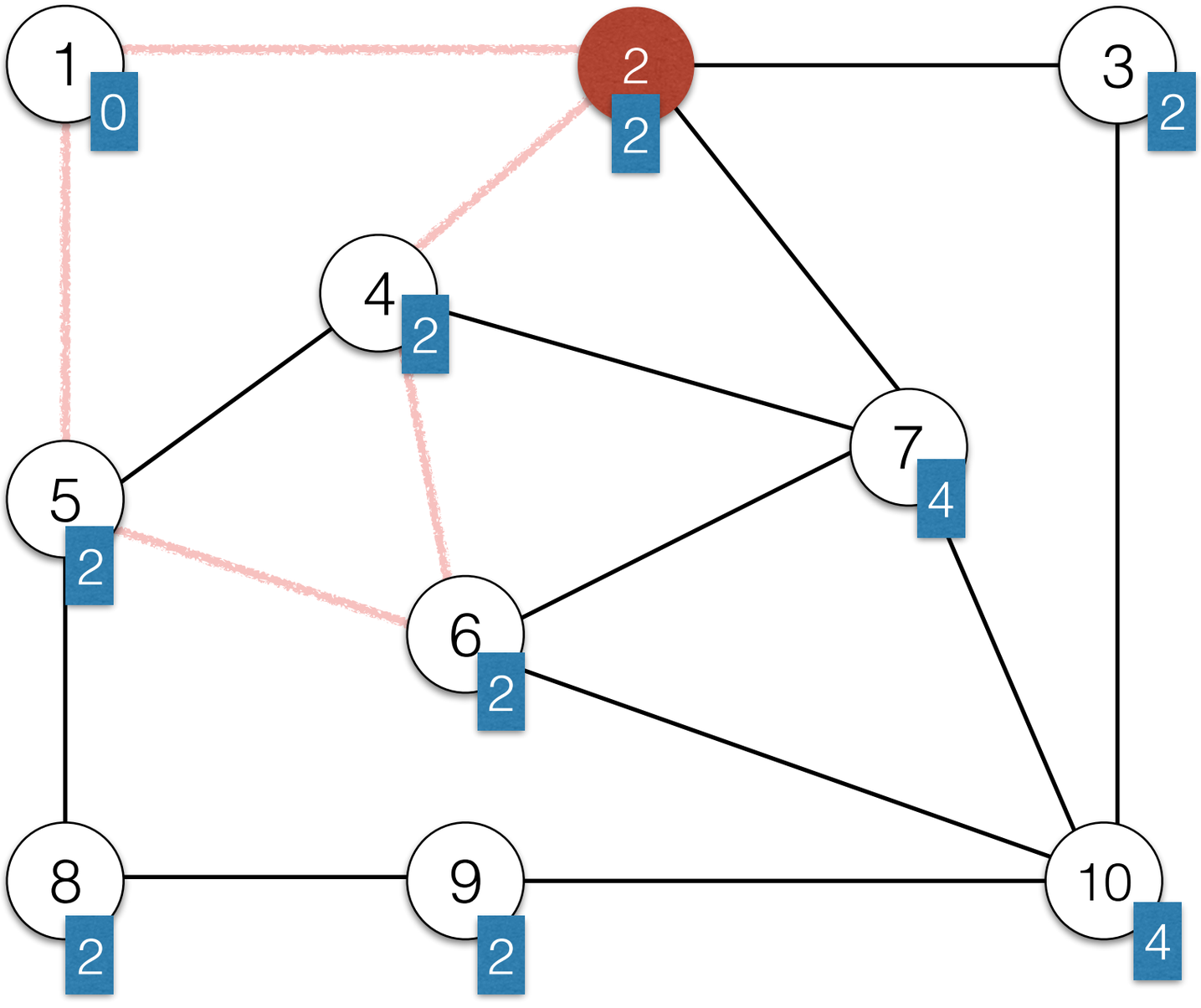


Finding Euler Circuits



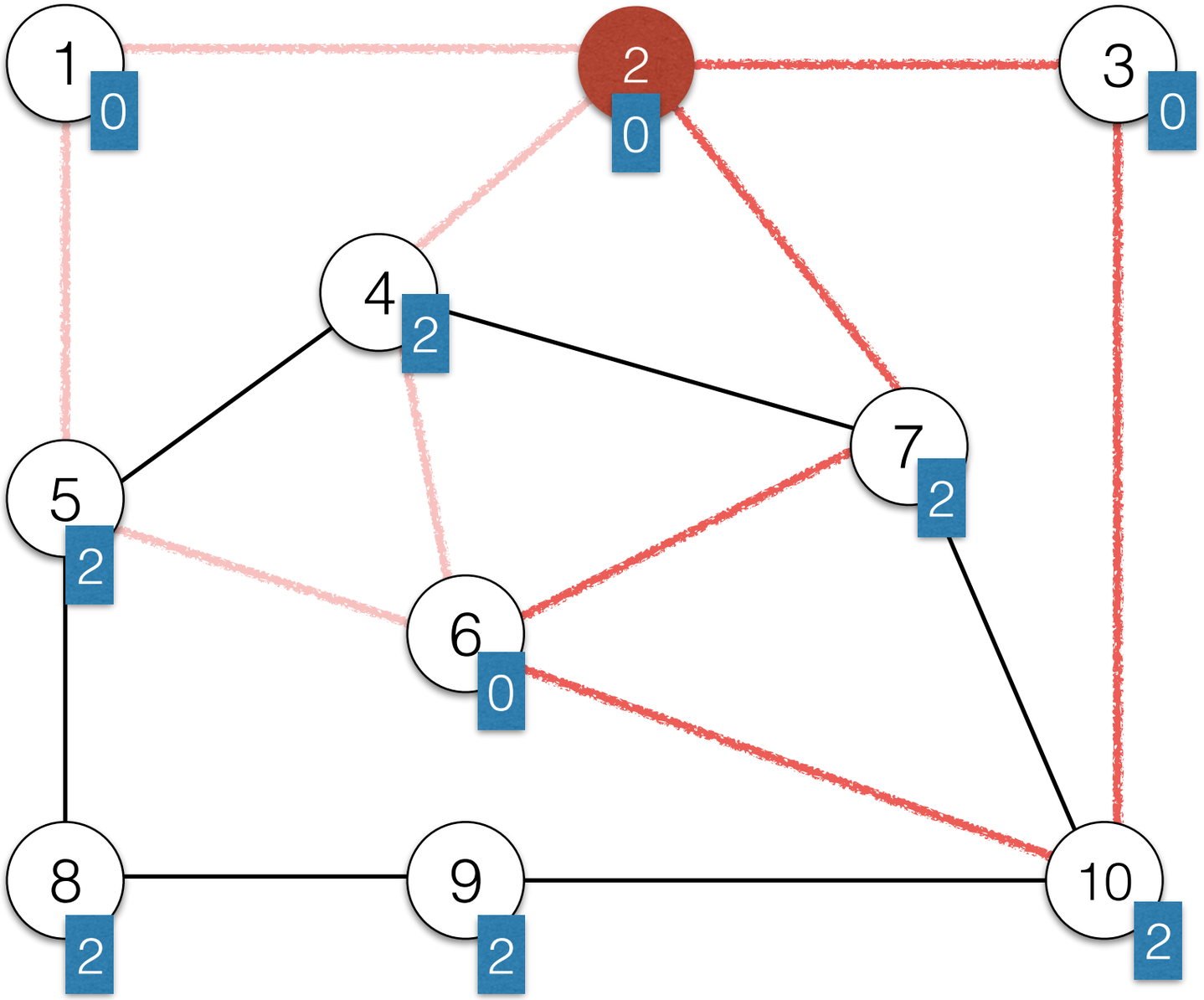
1 2 4 6 5 1

Finding Euler Circuits



1 2 4 6 5 1

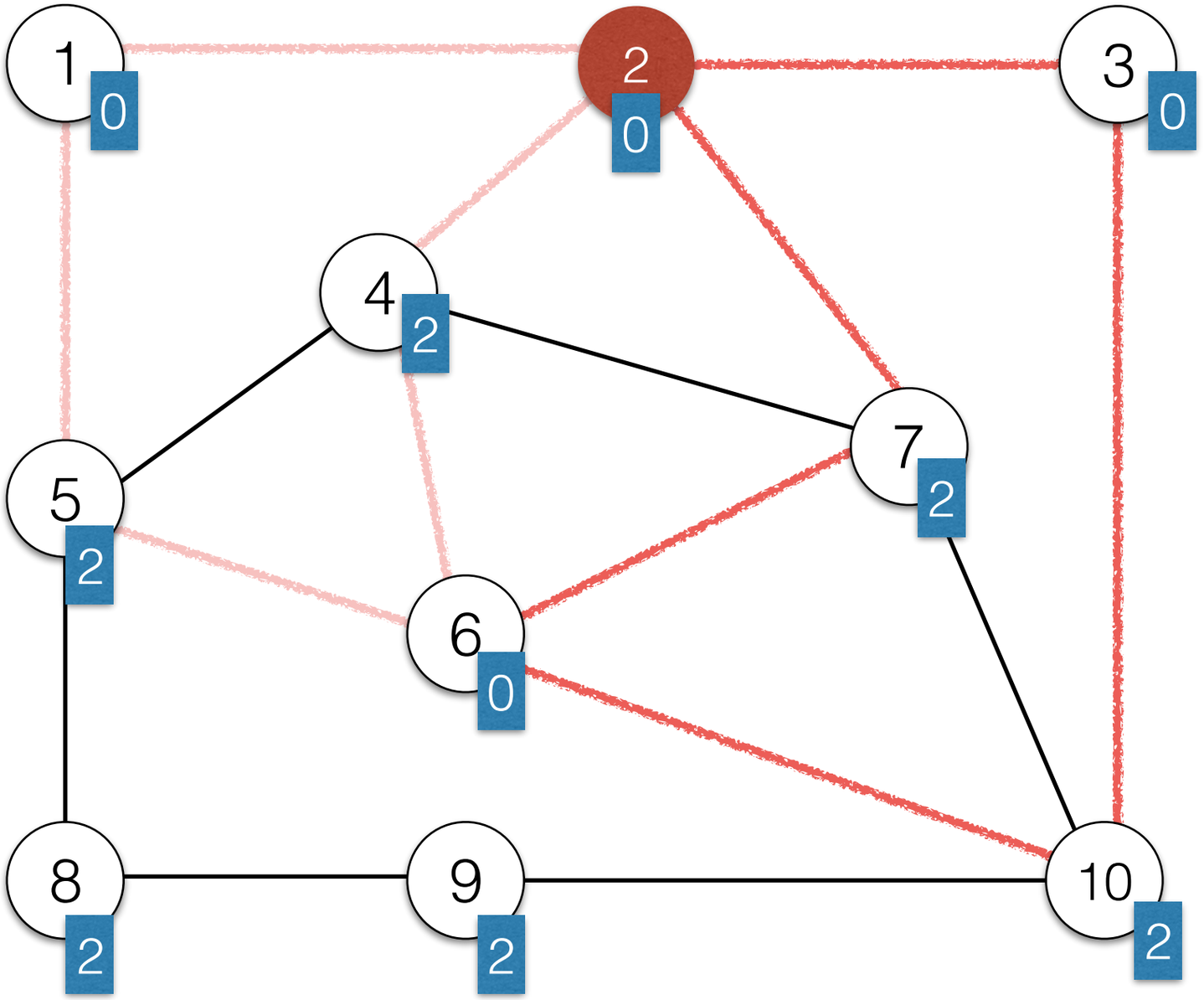
Finding Euler Circuits



2 7 6 10 3 2

1 2 4 6 5 1

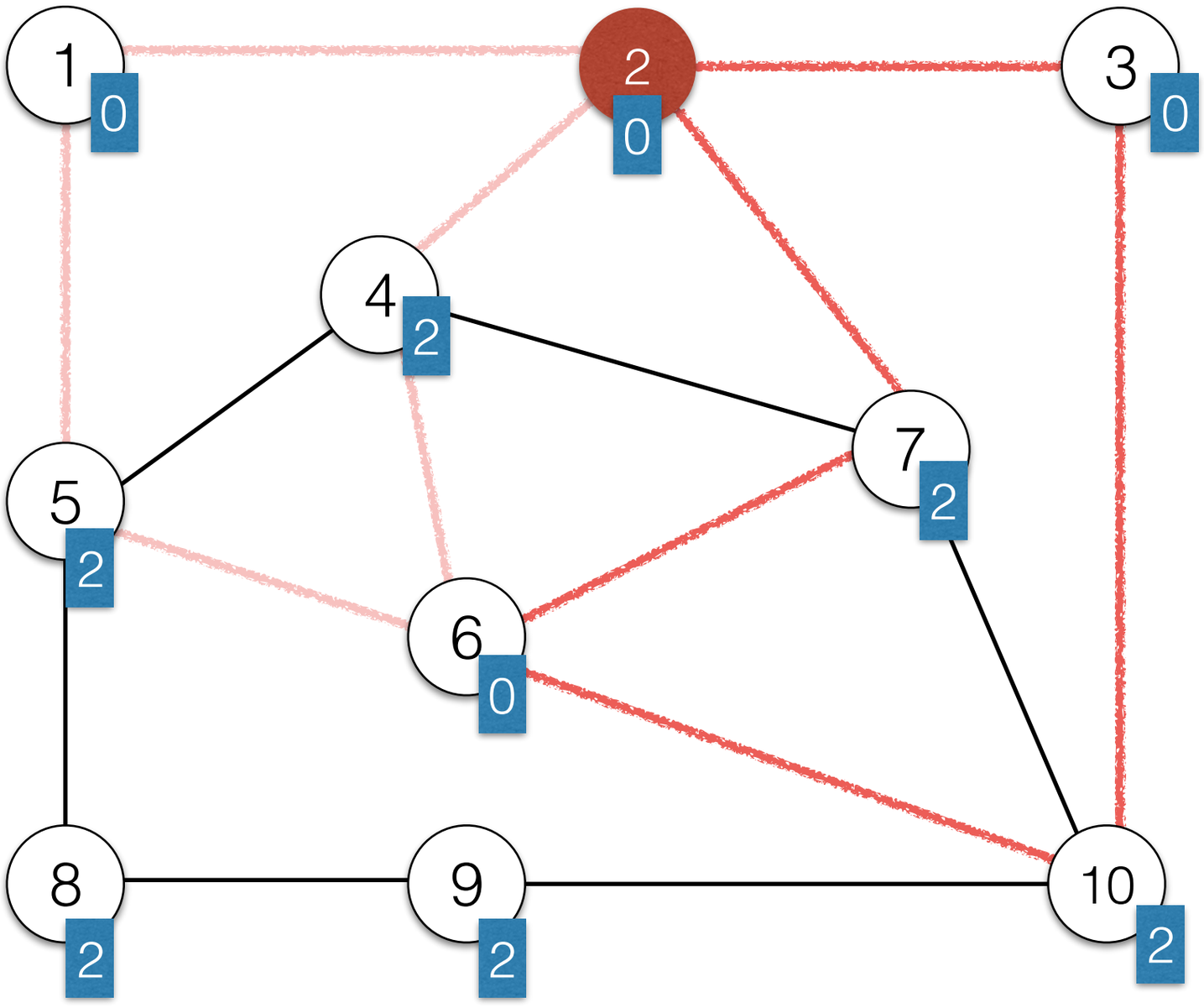
Finding Euler Circuits



2 7 6 10 3 2

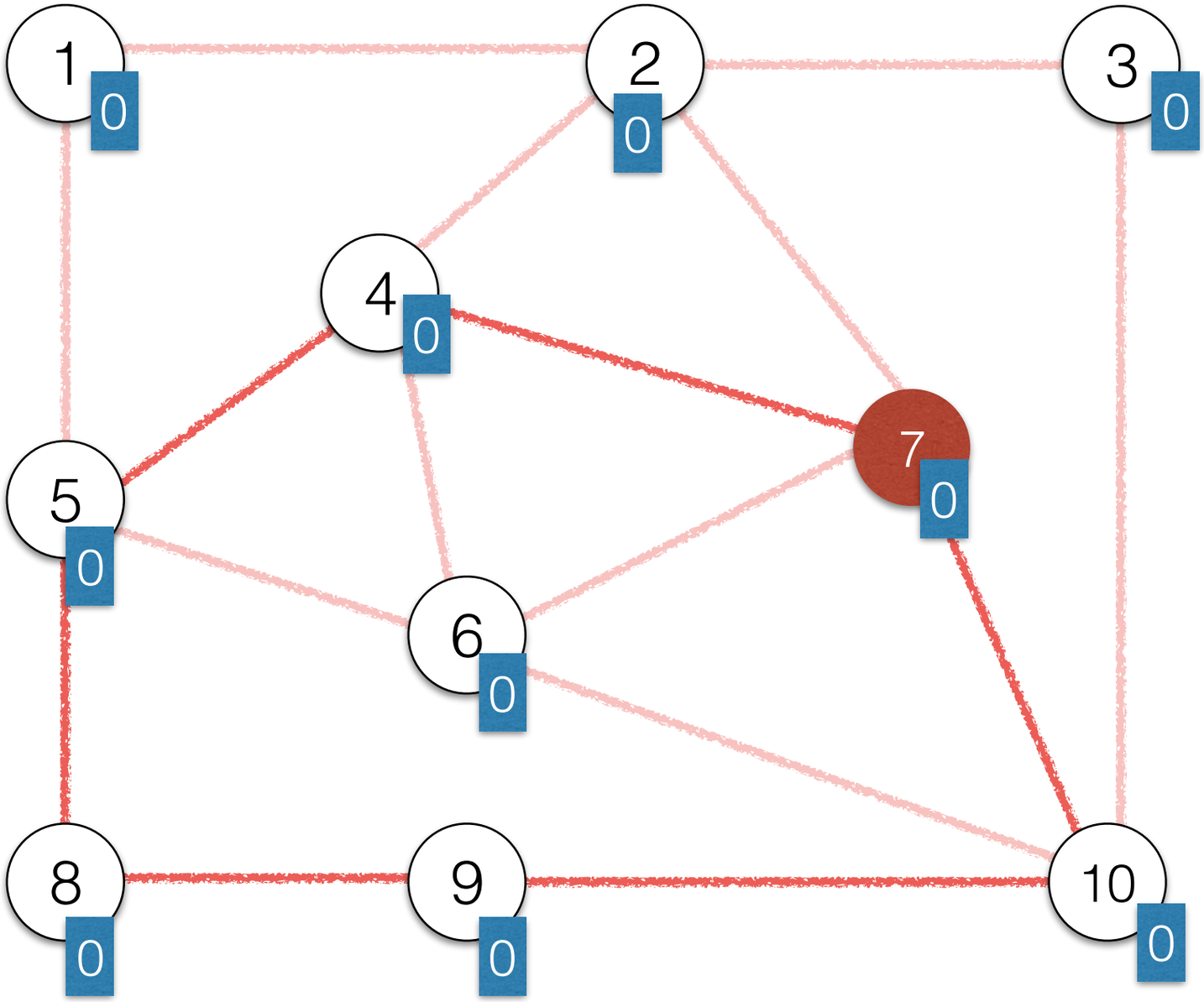
1 4 6 5 1

Finding Euler Circuits



1 **2 7 6 10 3 2** 4 6 5 1

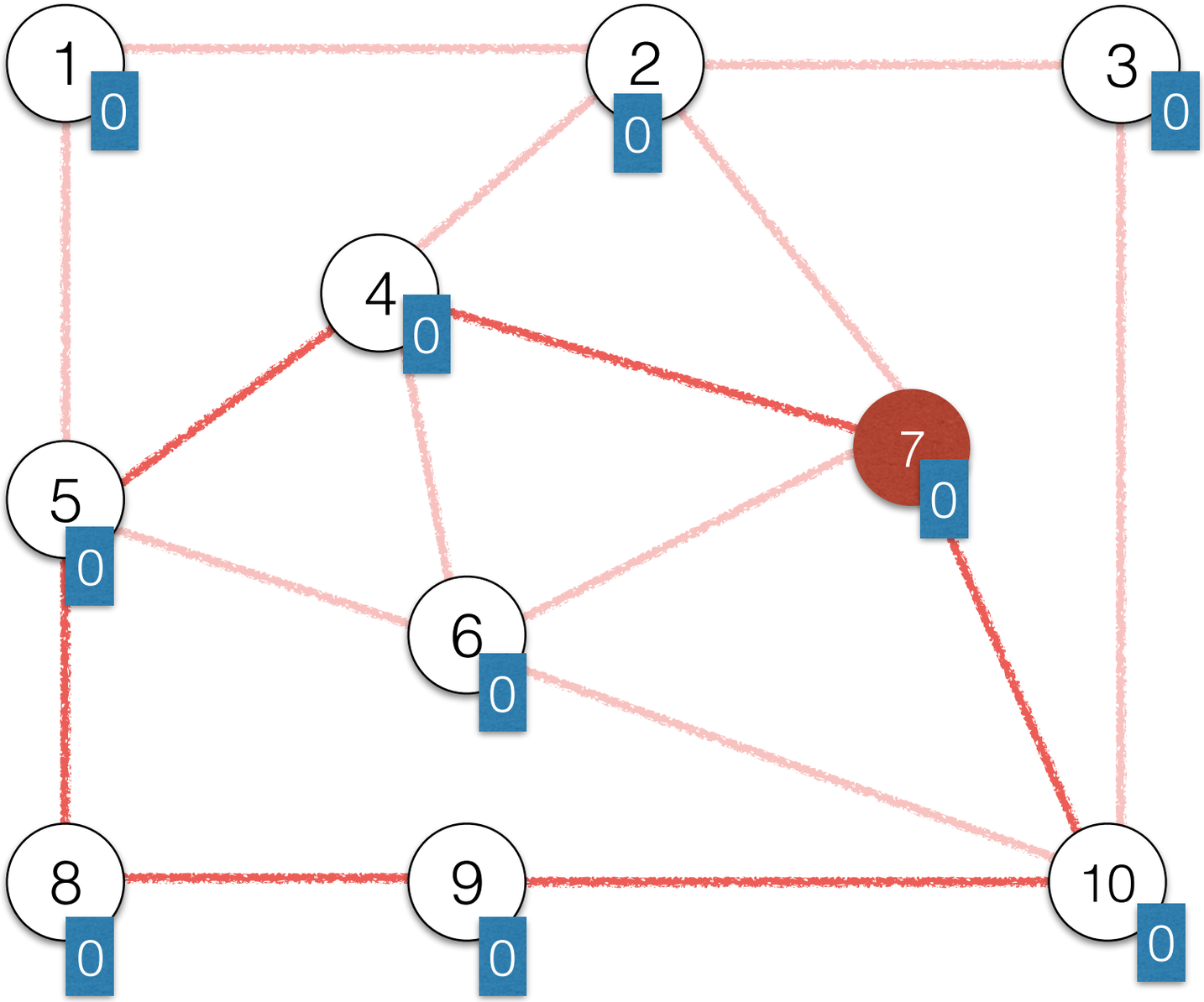
Finding Euler Circuits



7 4 5 8 9 10 7

1 2 7 6 10 3 2 4 6 5 1

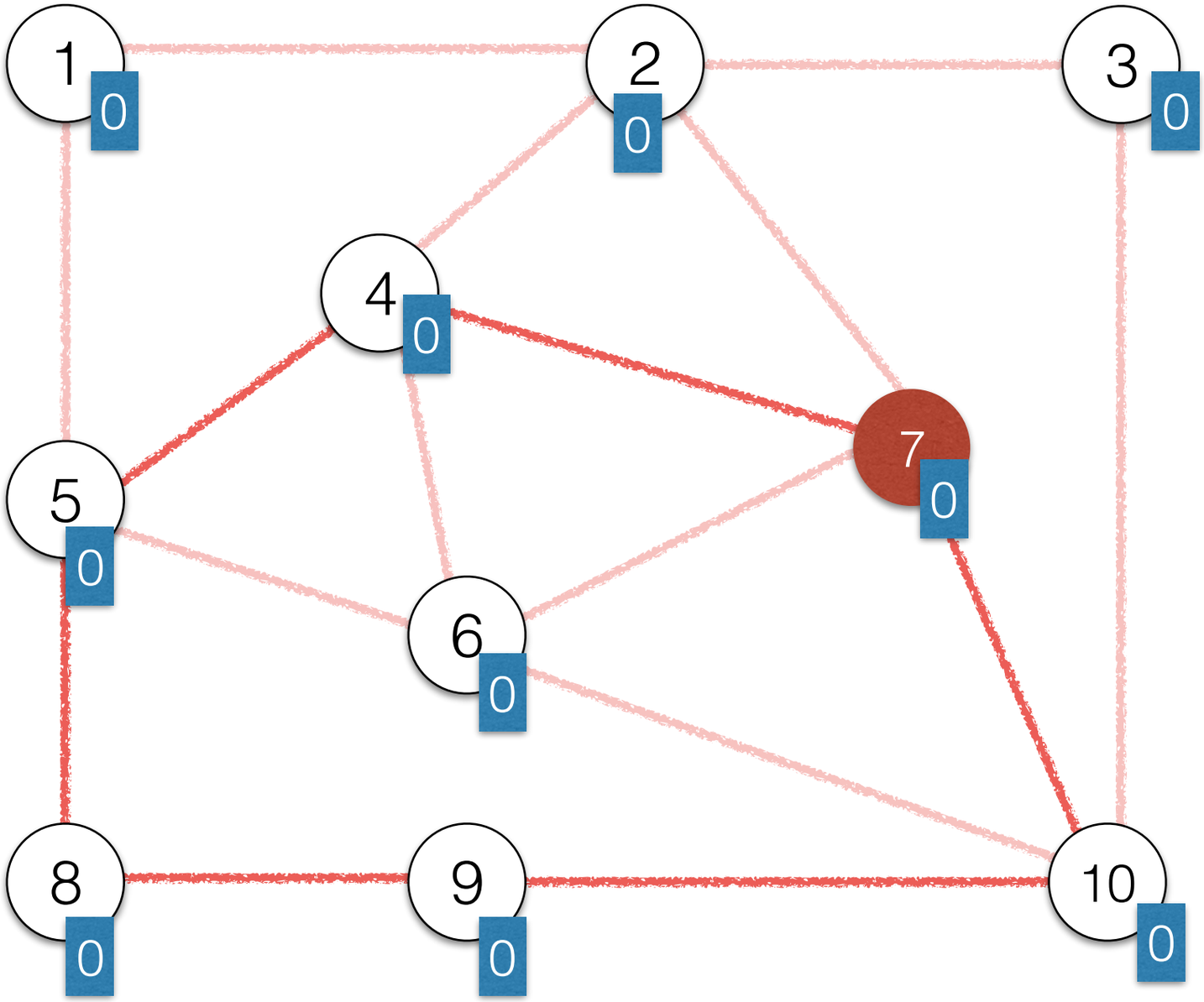
Finding Euler Circuits



7 4 5 8 9 10 7

1 2 6 10 3 2 4 6 5 1

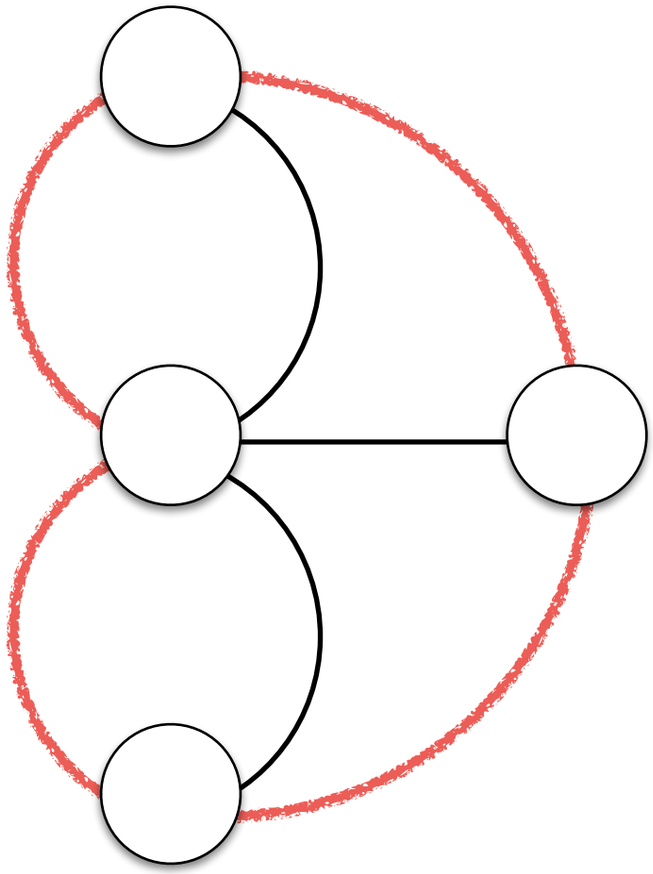
Finding Euler Circuits

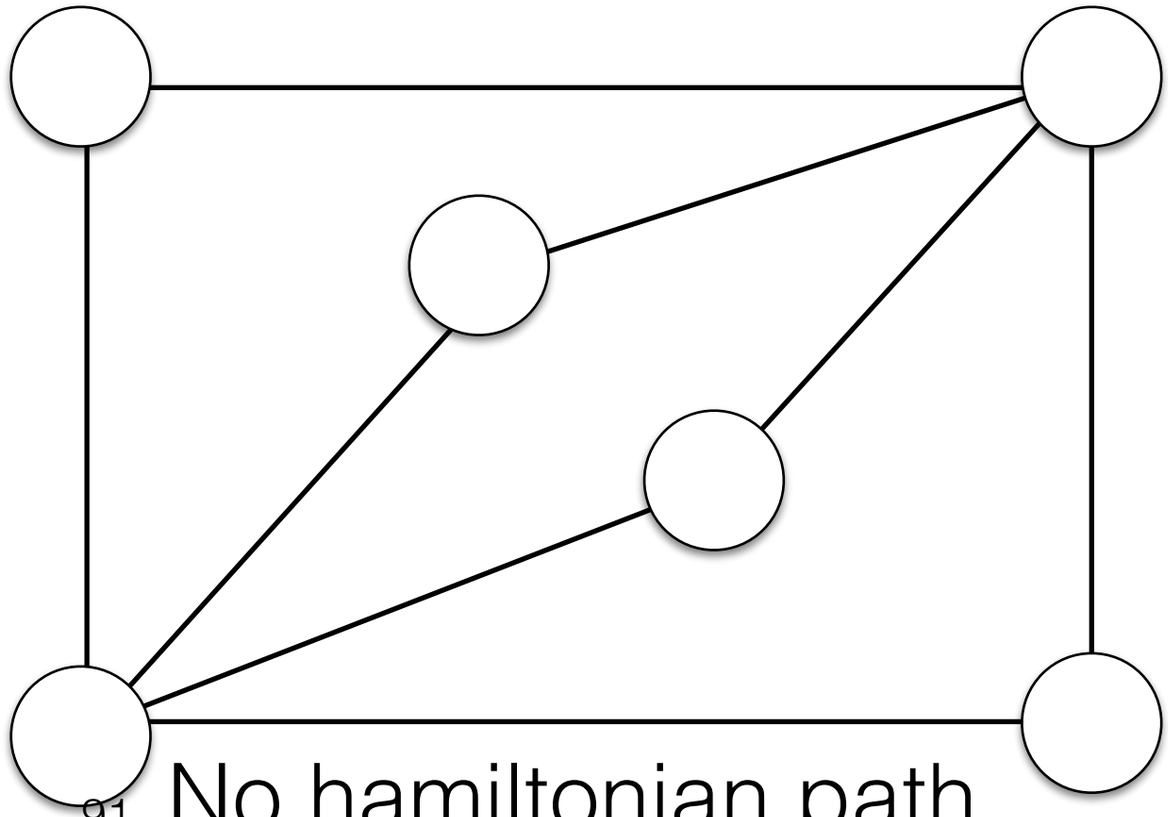


1 2 **7 4 5 8 9 10 7** 6 10 3 2 4 6 5 1

Hamiltonian Cycle

- A Hamiltonian Path is a path through an undirected graph that visits ***every vertex*** exactly once (except that the first and last vertex may be the same).
- A Hamiltonian Cycle is a Hamiltonian Path that starts and ends in the same node.

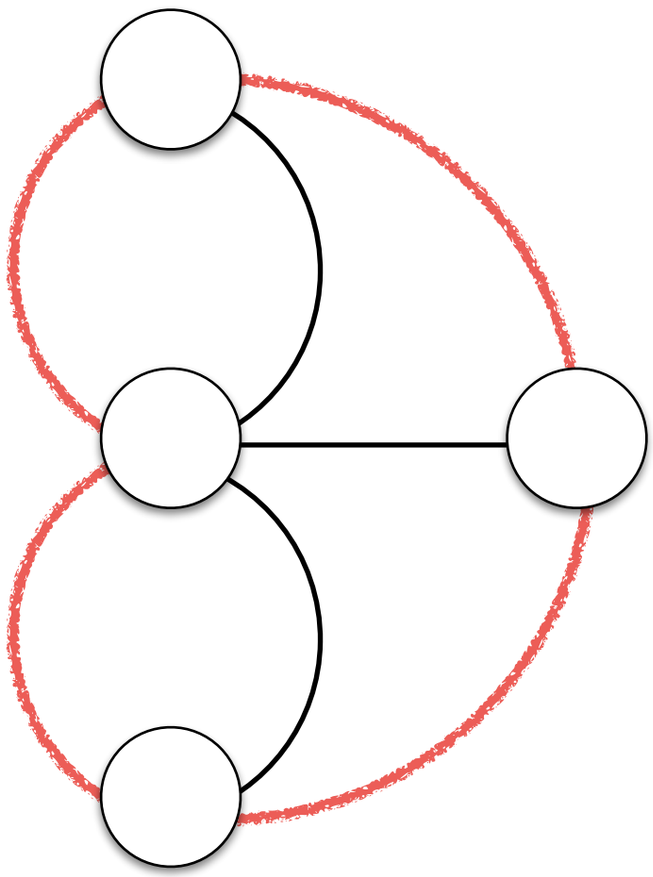


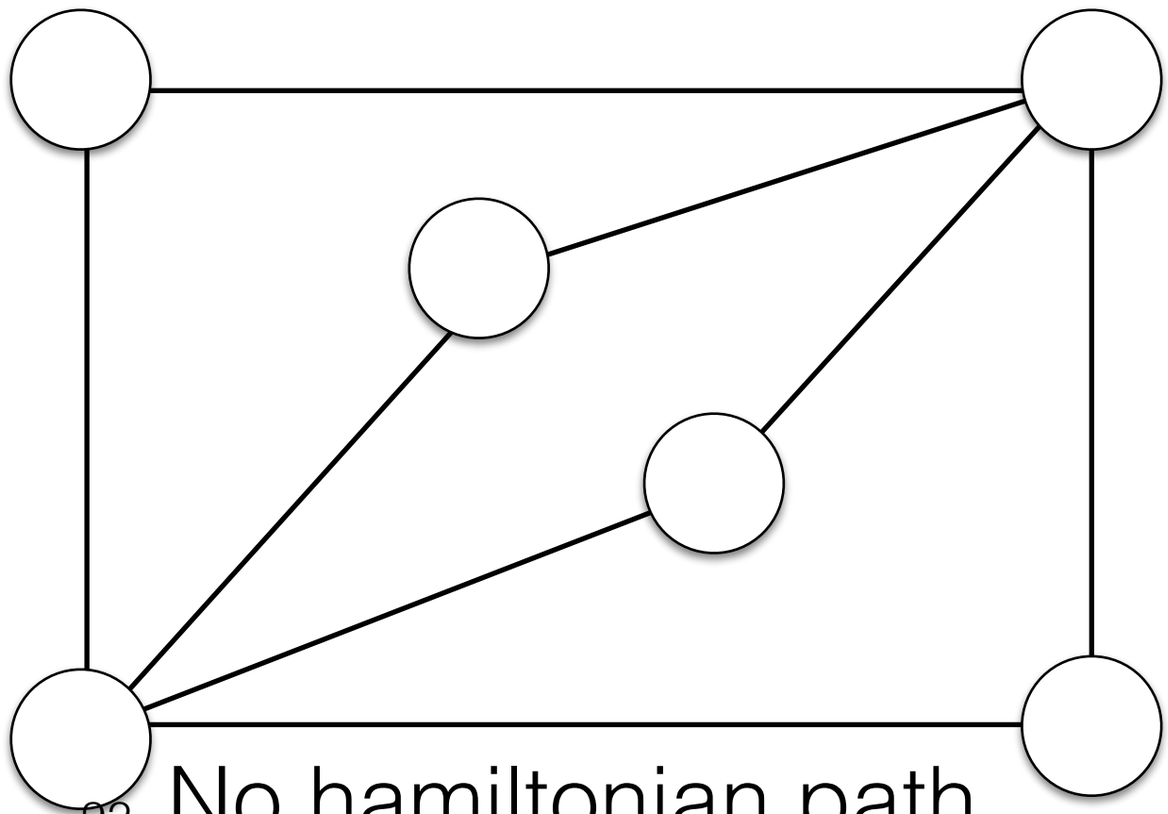


No hamiltonian path.

Hamiltonian Cycle

- We can check if a graph contains an Euler Cycle in linear time.
- Surprisingly, checking if a graph contains a Hamiltonian Path/Cycle is much harder!
- No polynomial time solution (i.e. $O(N^k)$) is known.





No hamiltonian path.