# Algorithms and Data Structures

## Graphs (Ch. 8)

Morgan Ericsson

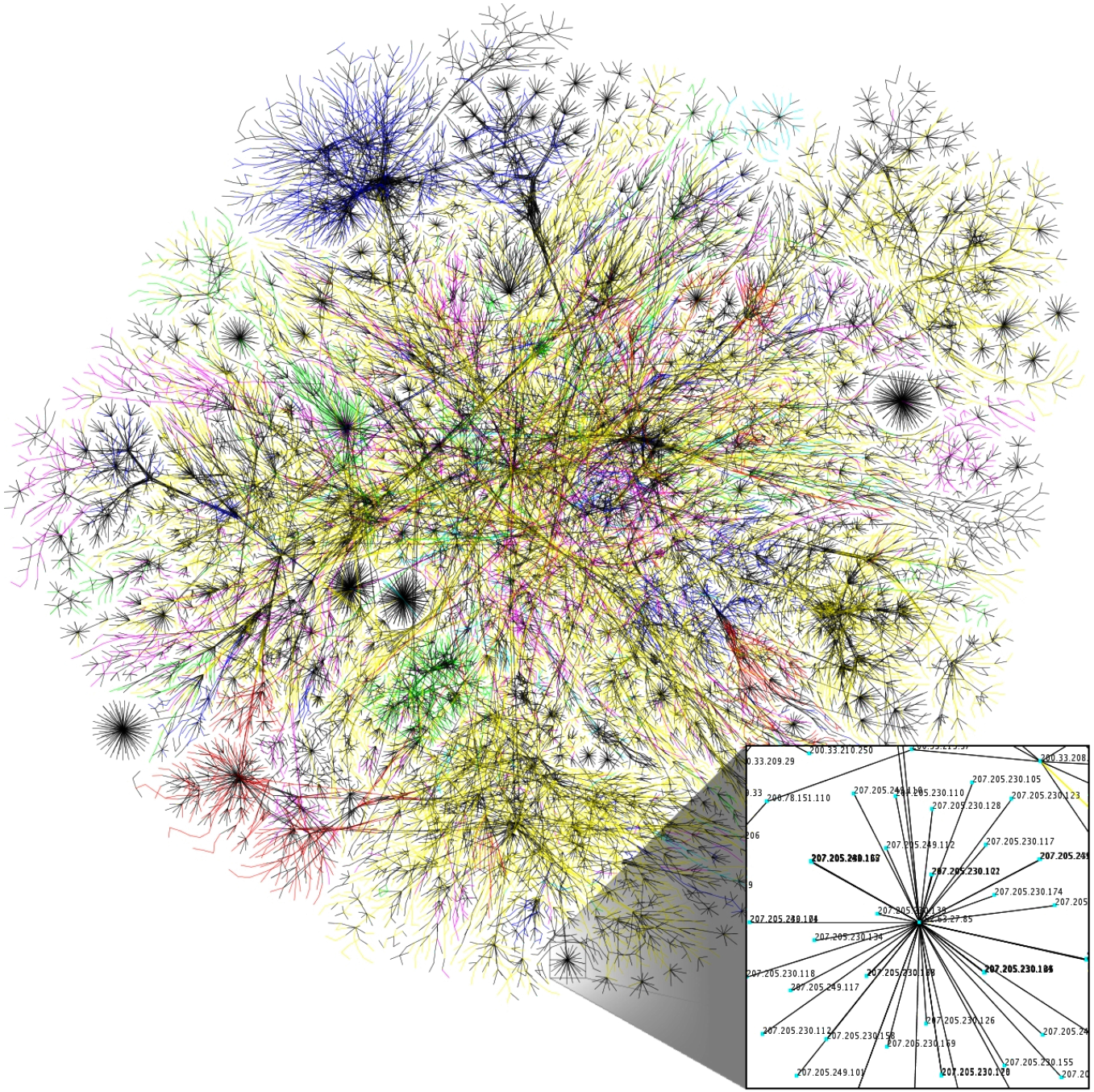
# Today

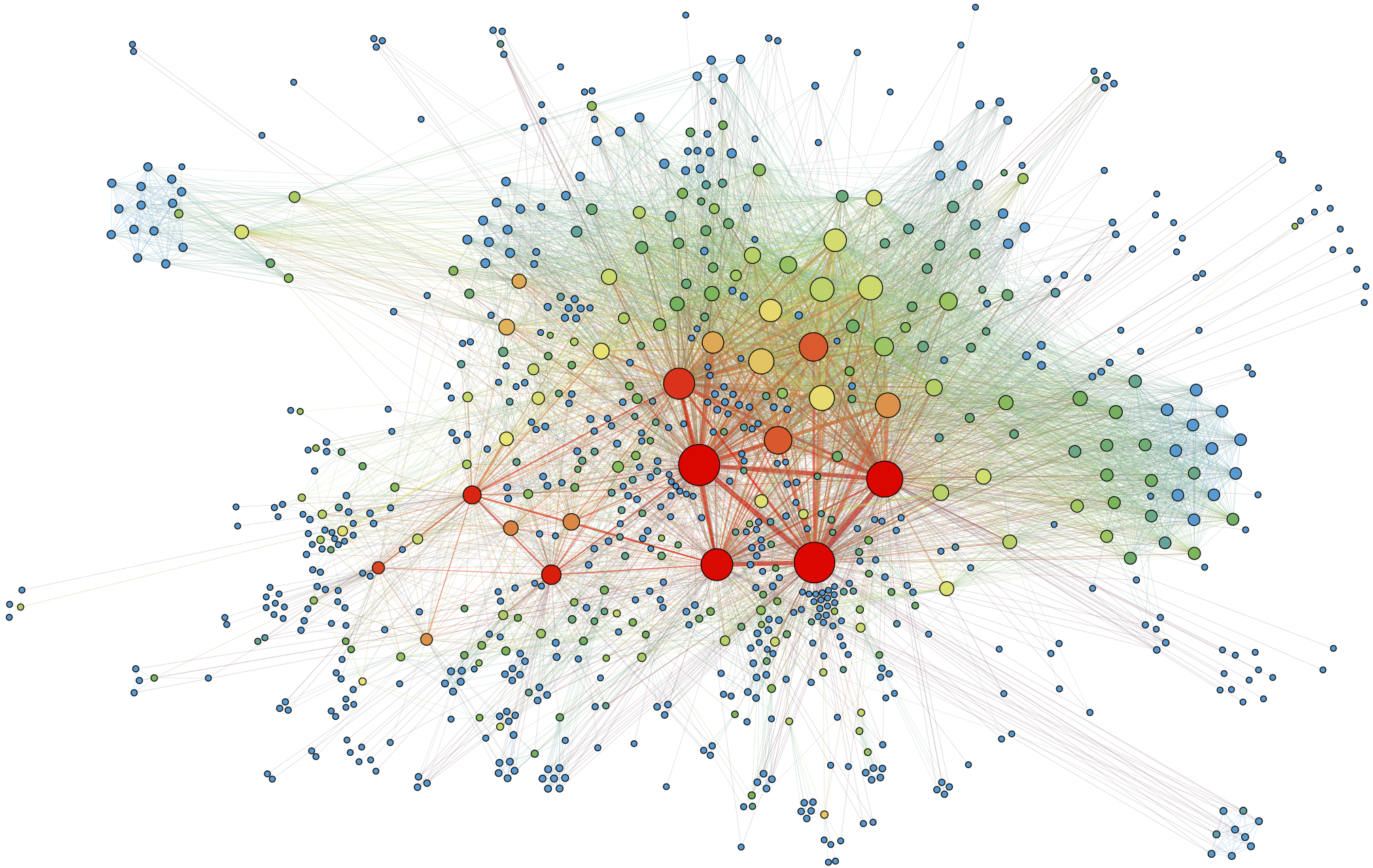
- Undirected graphs
  - Breadth- and depth-first searches
  - Connected components
- Directed graphs
  - Topological sort
  - Strong components
- Shortest paths

# Graphs

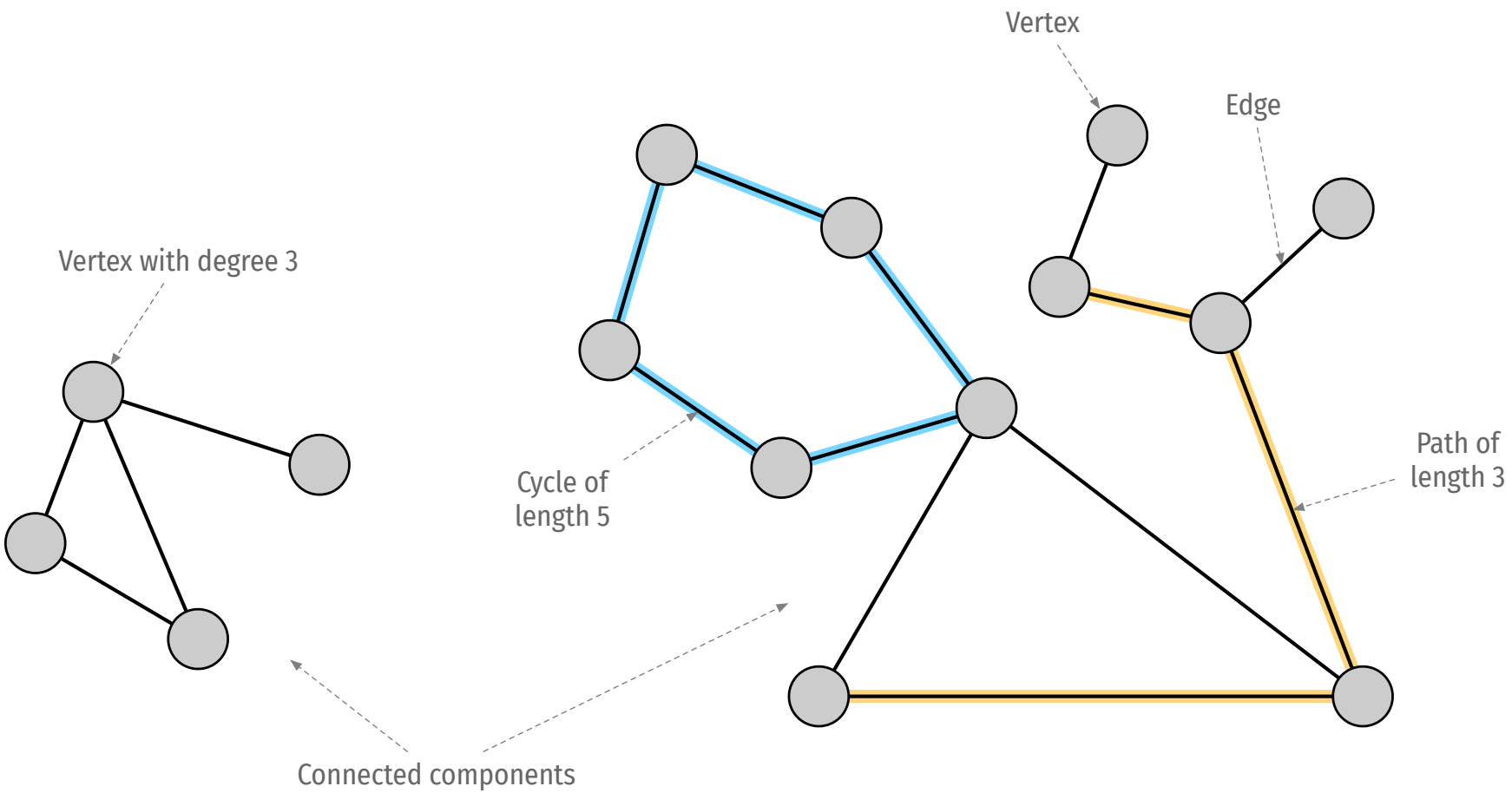
# Graphs

- A graph consists of vertices and edges
  - Nodes that are connected
- Many applications
  - Routing in networks
  - Protein interactions
  - Social networks
- And many algorithms...


200.33.210.250
207.205.230.105
207.205.230.110
207.205.230.128
207.205.230.123
207.205.230.117
207.205.230.174
207.205.230.134
207.205.230.118
207.205.249.117
207.205.230.112
207.205.230.126
207.205.230.155
207.205.249.101



# An undirected graph



# Some problems

- Is there a path between two vertices?
- What is the shortest path between two vertices?
- Is there a cycle in the graph?
- Is there a cycle that use each edge exactly once? Each vertice?
- What is the best way to connect all of the vertices?
- ...

# Undirected Graphs

# Undirected graphs

- We represent the vertices as integers
- We can represent the edges in different ways
  - List of edges
  - Adjacency matrix
  - Adjacency list

# Undirected graphs

- Operations
  - Add edge
  - Get add adjacent vertices
  - Number of vertices
  - Number of edges

# First implementation

```
 1  class Graph:
 2    def __init__(self, v:int) -> None:
 3      self.v = v
 4      self.el = []
 5
 6    @property
 7    def V(self) -> int:
 8      return self.v
 9
10    @property
11    def E(self) -> int:
12      return len(self.el)
```

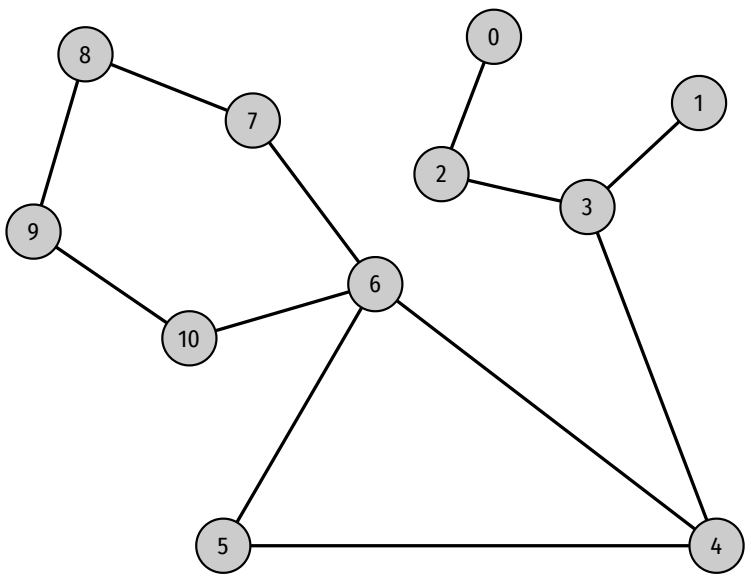
# First implementation

```
1  from fastcore.basics import patch
2
3  @patch
4  def add_edge(self:Graph, v:int, w:int) -> None:
5    if v < self.v and w < self.v:
6      self.el.append((v, w))
```

# First implementation

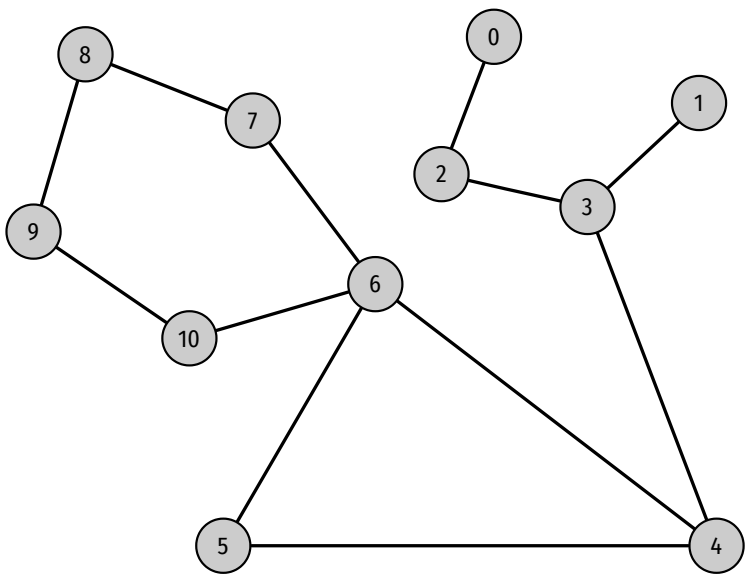
```
 1  @patch
 2  def adj(self:Graph, v:int) -> list[int]:
 3    res = []
 4    for a, b in self.el:
 5      if v == a:
 6        res.append(b)
 7      elif v == b:
 8        res.append(a)
 9
10    return res
```

# Creating a graph




```
 1  g = Graph(11)
 2  g.add_edge(0, 2)
 3  g.add_edge(2, 3)
 4  g.add_edge(1, 3)
 5  g.add_edge(3, 4)
 6  g.add_edge(4, 5)
 7  g.add_edge(4, 6)
 8  g.add_edge(5, 6)
 9  g.add_edge(6, 7)
10  g.add_edge(7, 8)
11  g.add_edge(8, 9)
12  g.add_edge(9, 10)
13  g.add_edge(10, 6)
```

# Checking our graph




```
1  assert g.E == 12
2  assert g.V == 11
3  assert 4 in g.adj(6)
4  assert 7 in g.adj(6)
5  assert 9 not in g.adj(6)
```

# Computing the degree

```
 1  @patch
 2  def degree(self:Graph, v:int) -> int:
 3    d = 0
 4    for a, b in self.el:
 5      if v == a:
 6        d += 1
 7      elif v == b:
 8        d += 1
 9
10    return d
```

# Max degree

```
1  @patch(as_prop=True)
2  def max_degree(self:Graph) -> int:
3    mx = 0
4    for v in range(self.v):
5      tmp = self.degree(v)
6      if tmp > mx:
7        mx = tmp
8
9    return mx
```

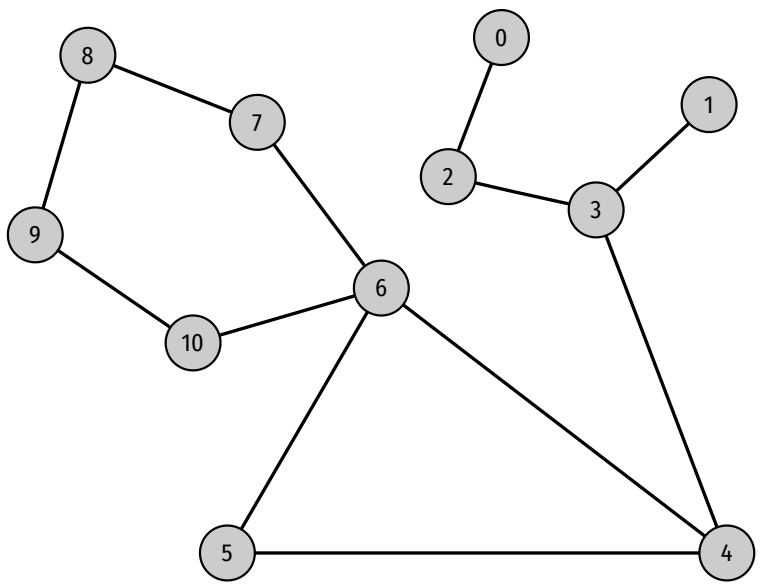
# Average degree

```
1  @patch(as_prop=True)
2  def mean_degree(self:Graph) -> int:
3    return 2 * self.E / self.V
```

# Has edge

```
1  @patch
2  def has_edge(self:Graph, v:int, w:int) -> bool:
3    return (v, w) in self.el or (w, v) in self.el
```

# Checking our algorithms



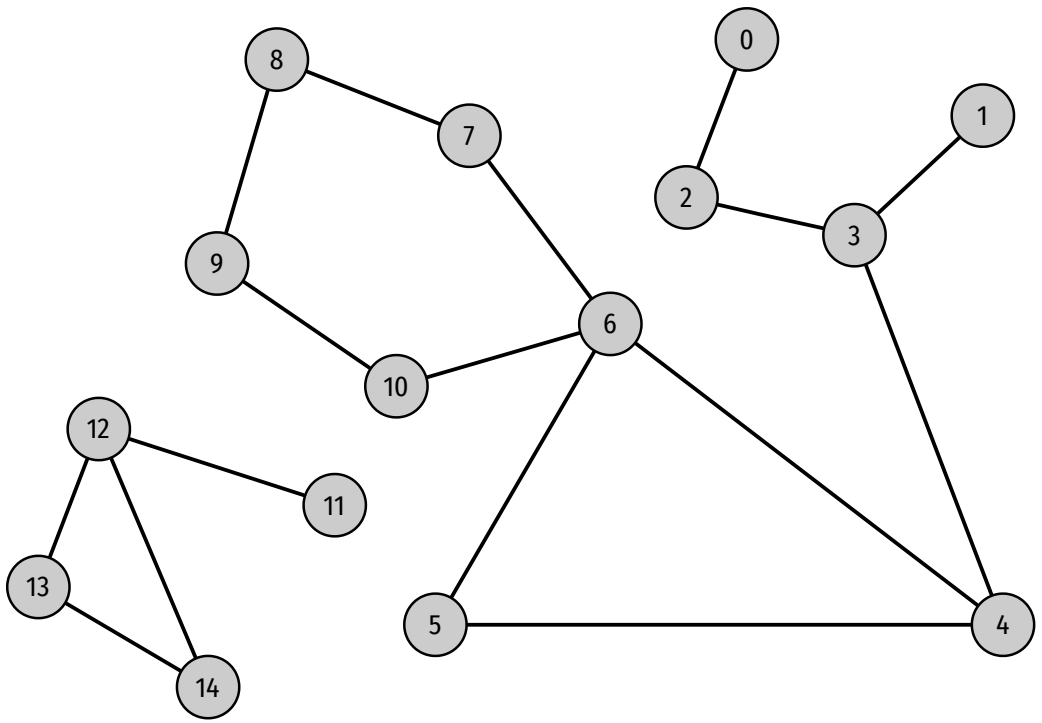

```python
assert g.degree(6) == 4
assert g.degree(0) == 1
assert g.max_degree == 4
assert g.has_edge(3, 2)
assert not g.has_edge(10, 4)
print(f'{g.mean_degree:.3f}')
```

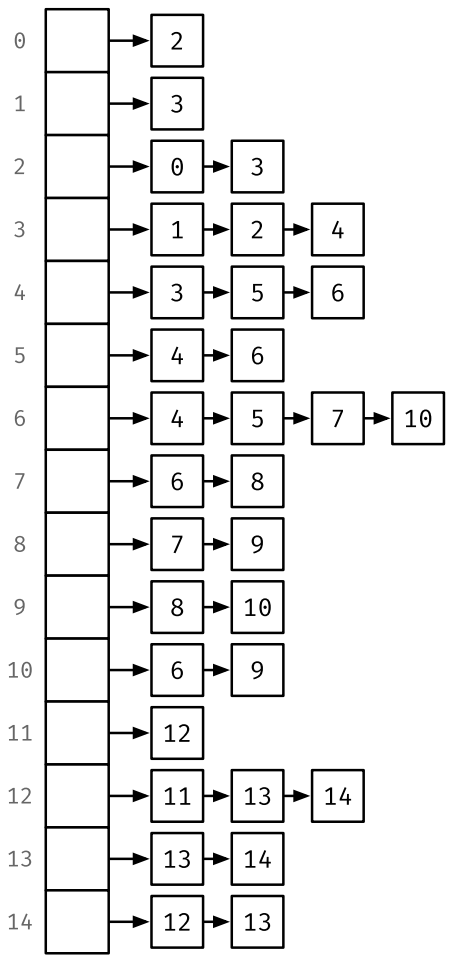
```
2.182
```

# Adjancency list and matrix

- An adjacency list represent the edges as a list of lists
  - For each vertex, store the vertices it is adjacent to
- An adjacency matrix represents the edges as elements in a matrix
  - If there is an edge between `v` and `w` then `adj_m[v][w] == true`
  - The boolean can be replaced by an int to represent a weight

# Adjancency list



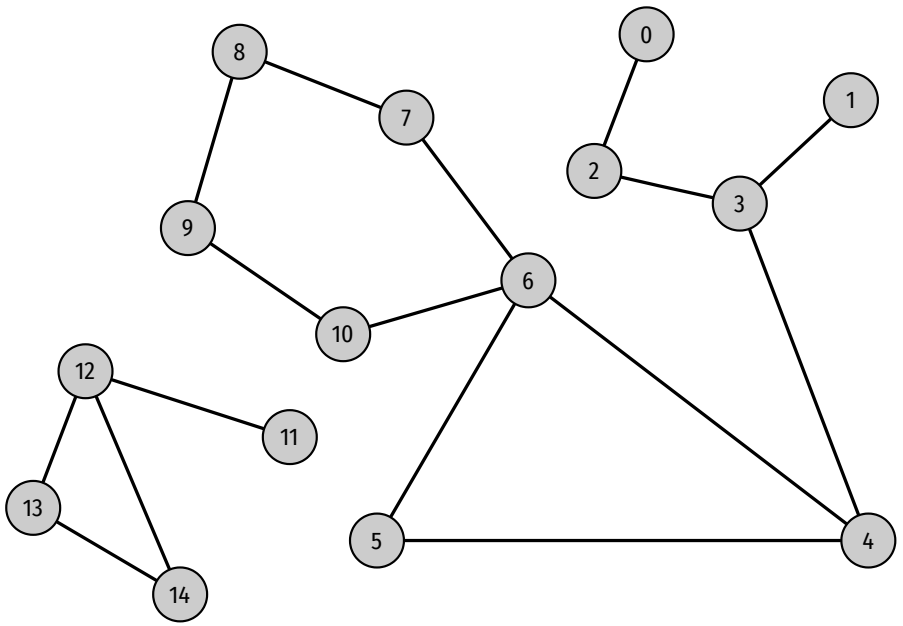



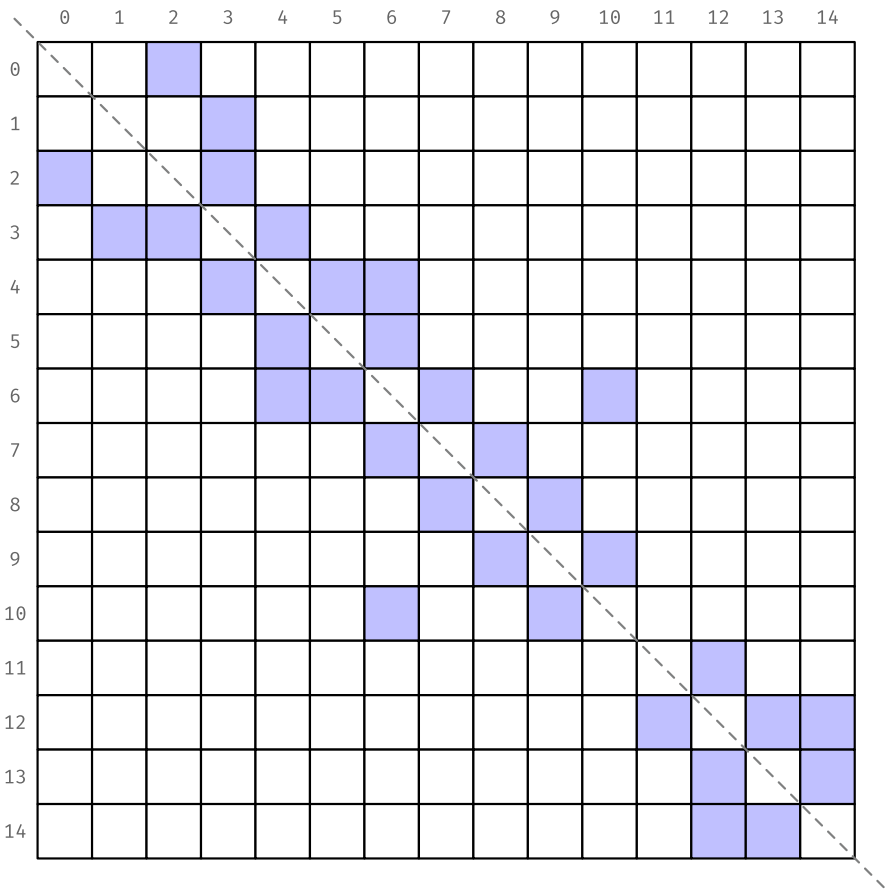
# Adjancency list

```
 1  v = 11
 2  adj_l = [[] for i in range(v)]
 3
 4  adj_l[3].append(4)
 5  adj_l[4].append(3)
 6  adj_l[4].append(5)
 7  adj_l[5].append(4)
 8  adj_l[4].append(6)
 9  adj_l[6].append(4)
10
11  print(adj_l[4])
```

```
[3, 5, 6]
```

# Adjancency matrix



# Adjancency matrix

```
 1  import numpy as np
 2
 3  v = 11
 4  adj_m = np.zeros((v, v), dtype=bool)
 5
 6  adj_m[3, 4] = True
 7  adj_m[4, 3] = True
 8
 9  adj_m[4, 5] = True
10  adj_m[5, 4] = True
11
12  adj_m[4, 6] = True
13  adj_m[6, 4] = True
14
15  print(np.flatnonzero(adj_m[4,:] == True))
```

```
[3 5 6]
```

# Which one should we use?

| Representation | Space | Add edge | Has edge | Adj |
|---|---|---|---|---|
| List of edges | $E$ | 1 | $E$ | $E$ |
| Adjacency matrix | $V^2$ | 1 | 1 | $V$ |
| Adjacency list | $E + V$ | 1 | $\text{degree}(v)$ | $\text{degree}(v)$ |

# Which one should we use?

- Depends (as always)
- Some algorithms work better with a specific representation
- So, often useful to have operations to convert between them
- But, in practice, adjacency list is often preferable
  - Real-world graphs are often sparse

# Reimplementing with adjacency list

```
 1  class Graph:
 2    def __init__(self, v:int) -> None:
 3      self.al = [[] for _ in range(v)]
 4
 5    @property
 6    def V(self) -> int:
 7      return len(self.al)
 8
 9    @property
10    def E(self) -> int:
11      return sum([len(l) for l in self.al]) // 2
```

# Reimplementing with adjacency list

```
1  from fastcore.basics import patch
2
3  @patch
4  def add_edge(self:Graph, v:int, w:int) -> None:
5    if v < len(self.al) and w < len(self.al):
6      self.al[v].append(w)
7      self.al[w].append(v)
```

# Reimplementing with adjacency list

```
1 @patch
2 def adj(self:Graph, v:int) -> list[int]:
3   if v < len(self.al):
4     return self.al[v]
```

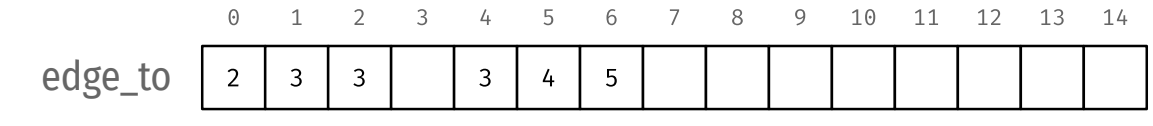
# Just checking...

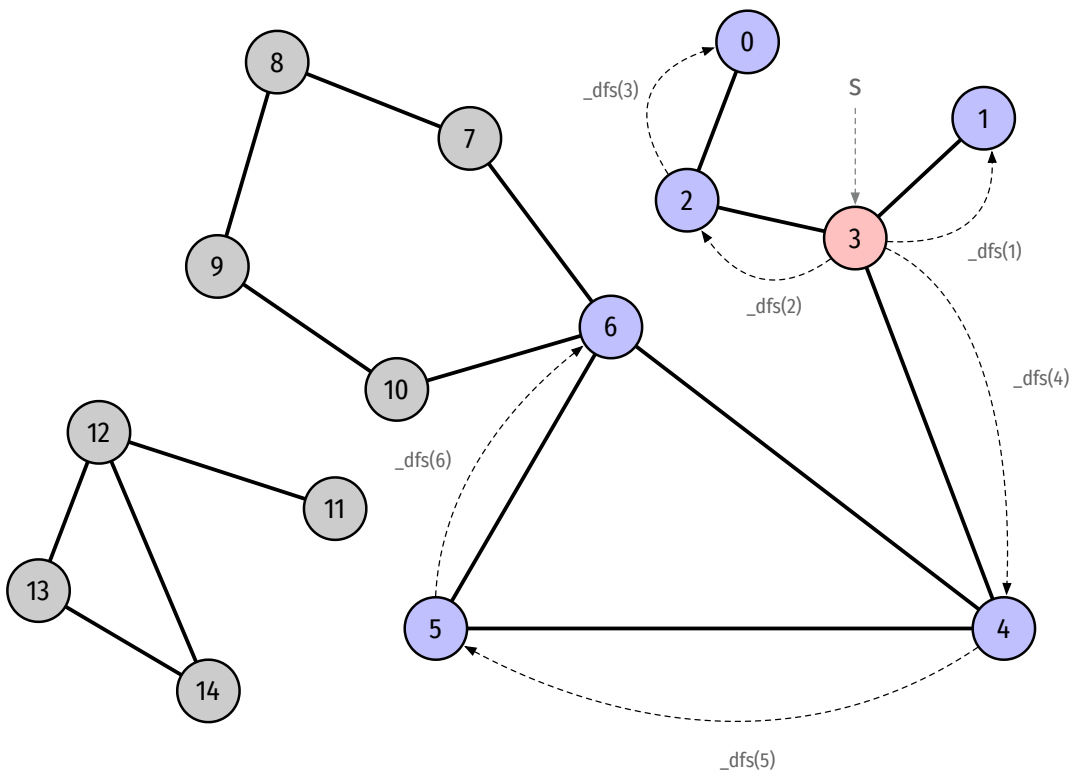
```
 1  g = Graph(15)
 2  g.add_edge(0, 2)
 3  g.add_edge(2, 3)
 4  g.add_edge(1, 3)
 5  g.add_edge(3, 4)
 6  g.add_edge(4, 5)
 7  g.add_edge(4, 6)
 8  g.add_edge(5, 6)
 9  g.add_edge(6, 7)
10  g.add_edge(7, 8)
11  g.add_edge(8, 9)
12  g.add_edge(9, 10)
13  g.add_edge(10, 6)
14
15  g.add_edge(11, 12)
16  g.add_edge(12, 13)
17  g.add_edge(12, 14)
18  g.add_edge(13, 14)
19
20  assert g.E == 16
21  assert g.V == 15
```

# Depth-first search (dfs)

- Find all vertices connected to a given source vertex
- Find a path between two vertices
- The algorithm visits a vertex
  - Marks it as visited
  - Vistits all unmarked vertices adjacent to it
- Systematic search through a graph

# Depth-first search



# Implementation

```
1  class DFSPaths:
2    def __init__(self, g:Graph, s:int) -> None:
3      self.G = g
4      self.s = s
5      self.marked = np.zeros(self.G.V, dtype=bool)
6      self.edge_to = np.zeros(self.G.V, dtype=int)
7      self._dfs(s)
```

# Implementation

```
1  @patch
2  def _dfs(self:DFSPaths, v:int) -> None:
3    self.marked[v] = True
4    for w in self.G.adj(v):
5      if not self.marked[w]:
6        self._dfs(w)
7        self.edge_to[w] = v
```

# Implementation

```
1  @patch
2  def has_path_to(self:DFSPaths, v:int) -> bool:
3    return self.marked[v]
```

# Implementation

```
@patch
def path_to(self:DFSPaths, v:int) -> list[int]:
  if not self.has_path_to(v):
    return None

  x, path = v, []
  while x != self.s:
    path.insert(0, x)
    x = self.edge_to[x]
  path.insert(0, self.s)
  return path
```

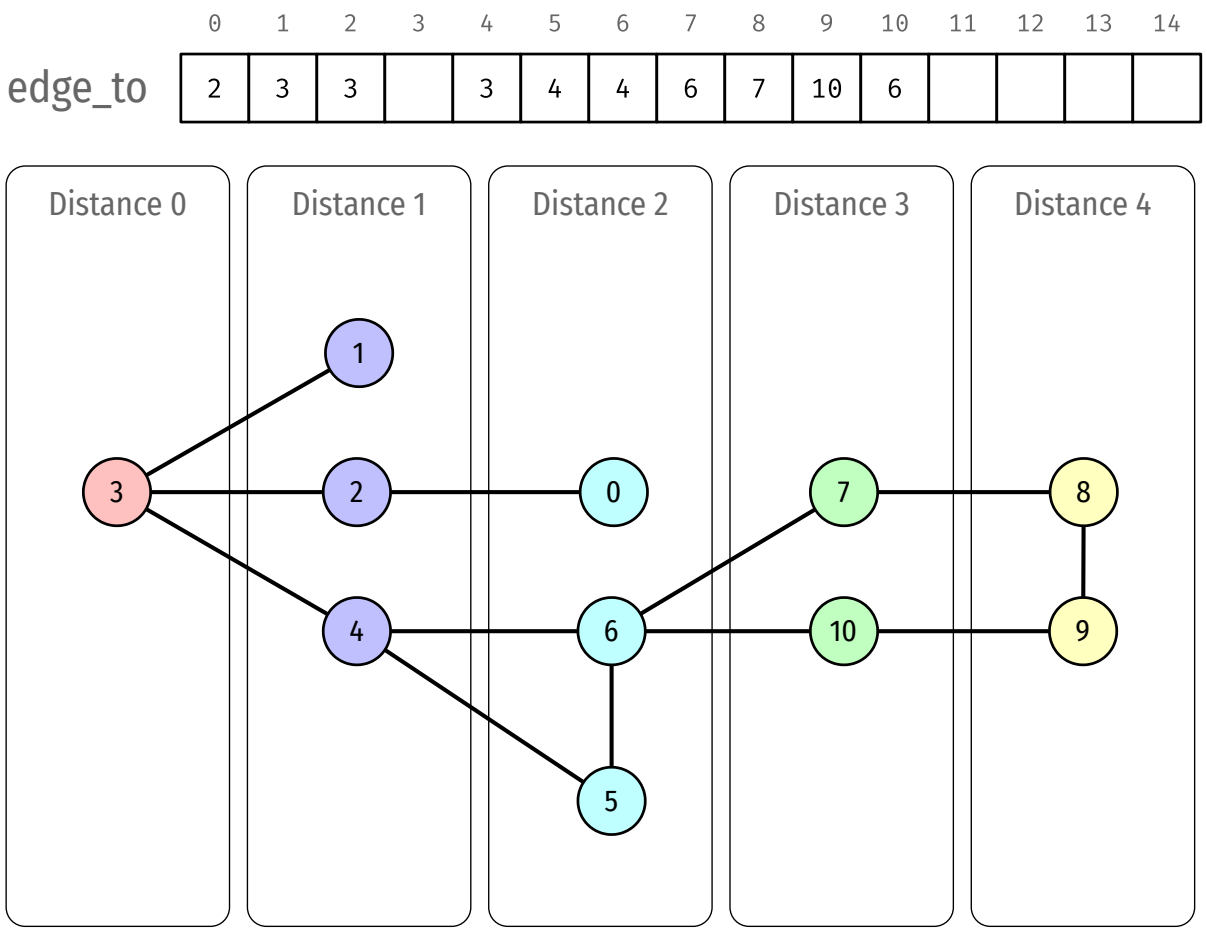
# Testing it

```
1 dfs = DFSPaths(g, 3)
2 assert dfs.has_path_to(5) == True
3 assert dfs.has_path_to(11) == False
4 assert dfs.path_to(8) == [3, 4, 5, 6, 7, 8]
```

# Breadth-first search (bfs)

- Bfs visits vertices “distance by distance”
- Starts with all adjacant vertices at distance 1, then distance 2, …
- Dfs explores as far along as path as possible before backtracking

# Bfs


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
edge_to
2 3 3 3 4 4 6 7 10 6
Distance 0
Distance 1
Distance 2
Distance 3
Distance 4
1
3
2
0
7
8
4
6
10
9
5

# Implementation

```
1  class BFSPaths:
2    def __init__(self, g:Graph, s:int) -> None:
3      self.G = g
4      self.s = s
5      self.marked = np.zeros(self.G.V, dtype=bool)
6      self.edge_to = np.zeros(self.G.V, dtype=int)
7      self._bfs(s)
```

# Implementation

```
 1  @patch
 2  def _bfs(self:BFSPaths, v:int) -> None:
 3    q = []
 4    q.insert(0, v)
 5    self.marked[v] = True
 6    while q:
 7      vv = q.pop()
 8      for w in self.G.adj(vv):
 9        if not self.marked[w]:
10          q.insert(0, w)
11          self.marked[w] = True
12          self.edge_to[w] = vv
```

# Implementation

```
1  @patch
2  def has_path_to(self:BFSPaths, v:int) -> bool:
3    return self.marked[v]
```

# Implementation

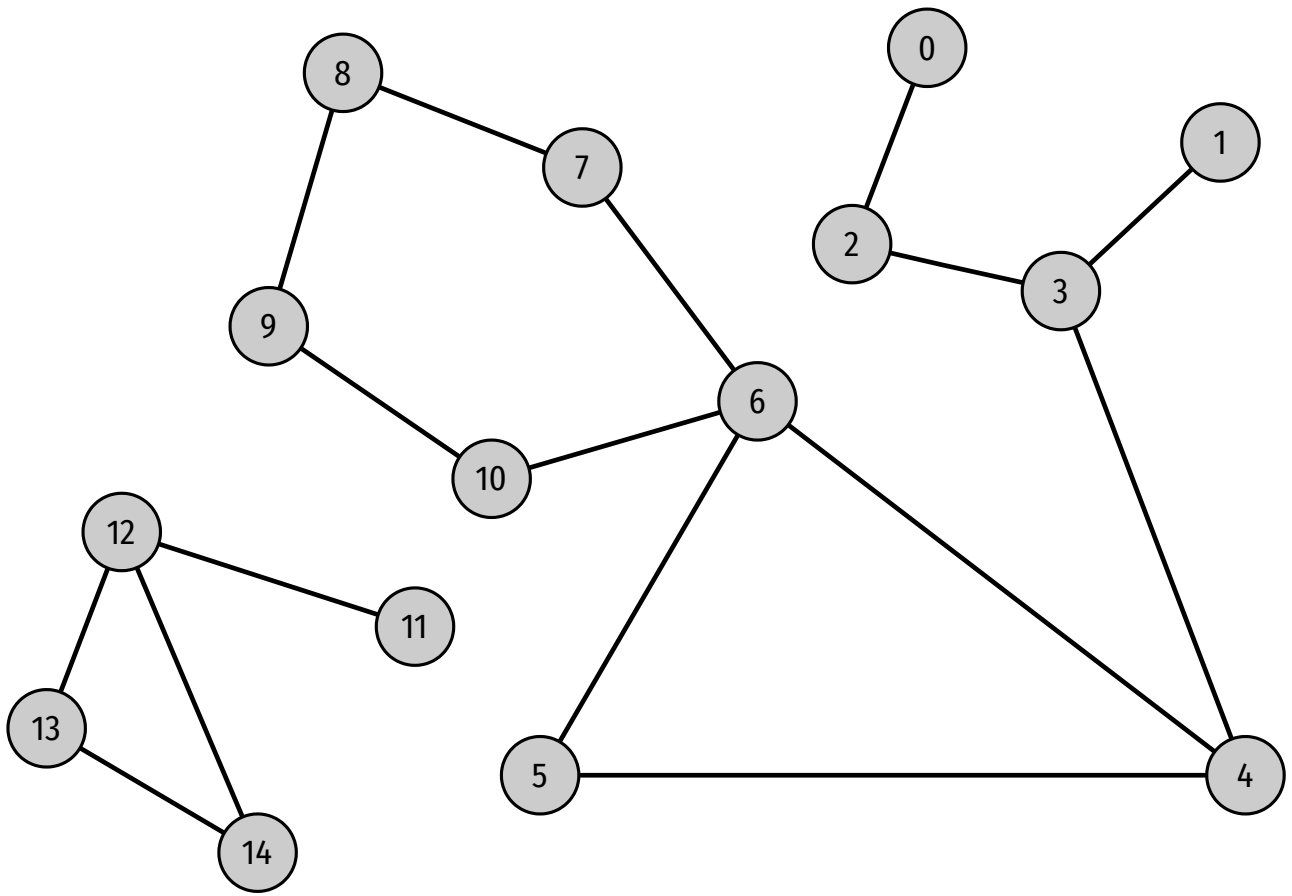
```
@patch
def path_to(self:BFSPaths, v:int) -> list[int]:
  if not self.has_path_to(v):
    return None

  x, path = v, []
  while x != self.s:
    path.insert(0, x)
    x = self.edge_to[x]
  path.insert(0, self.s)
  return path
```

# Testing it

```
1 bfs = BFSPaths(g, 3)
2 assert bfs.has_path_to(5) == True
3 assert bfs.has_path_to(11) == False
4 print(bfs.path_to(8))
5 print(bfs.edge_to)
```

```
[3, 4, 6, 7, 8]
[ 2  3  3  0  3  4  4  6  7 10  6  0  0  0  0]
```

# Testing it

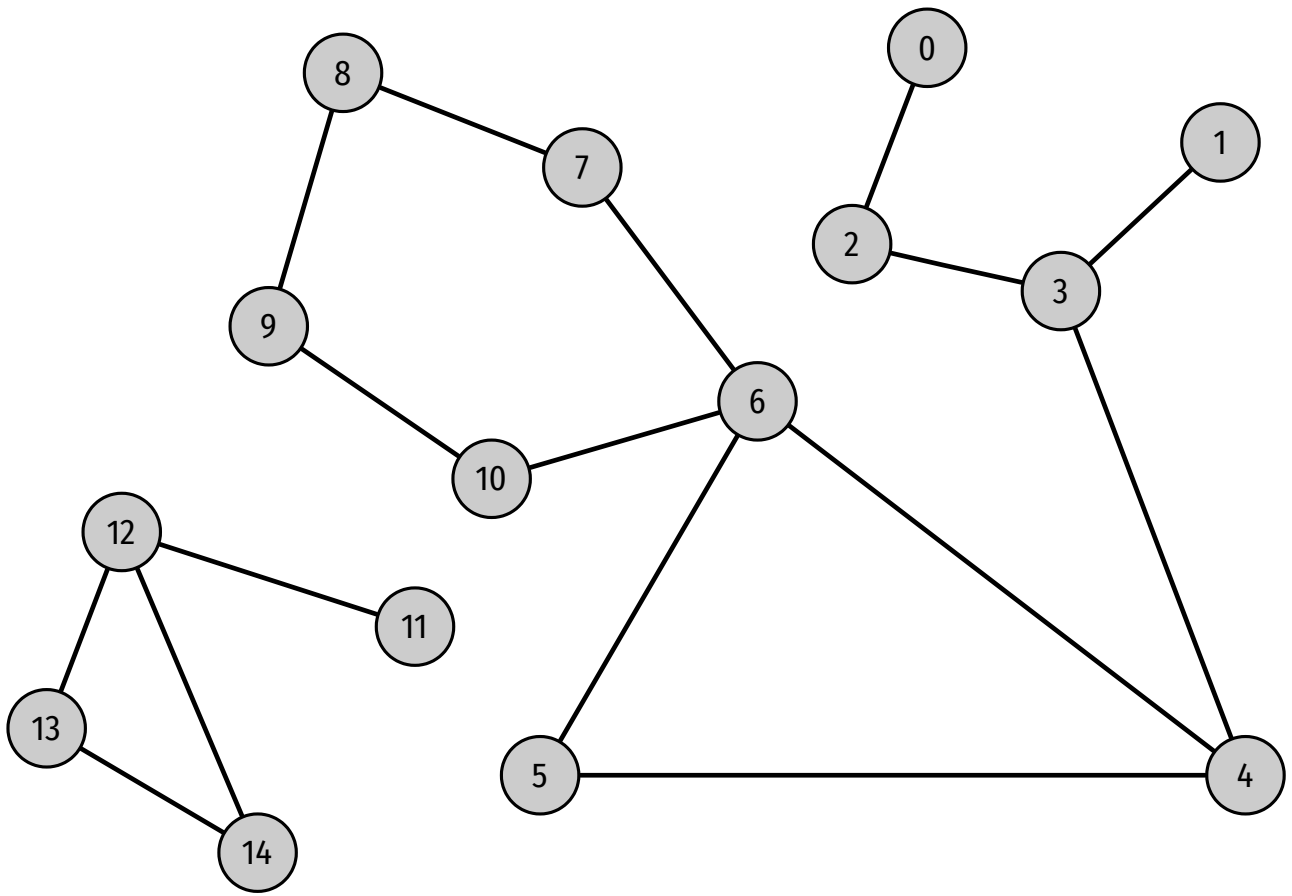


# Testing it

```
1 print(dfs.path_to(8))
2 print(bfs.path_to(8))
```

```
[3, 4, 5, 6, 7, 8]
[3, 4, 6, 7, 8]
```

# Connected components



# Connected components

- Two vertices, *v* and *w* are connected if there is a path between them
- A connected component is a maximal set of connected vertices
- Similar to Union-Find, but we cannot use the same algorithms
- We use dfs here (but bfs also works)

# Implementation

```
1  class CC:
2    def __init__(self, G:Graph) -> None:
3      self.marked = np.zeros(G.V, dtype=bool)
4      self.ids = np.zeros(G.V, dtype=int)
5      self.cnt = 0
6      for v in range(G.V):
7        if not self.marked[v]:
8          self._dfs(G, v)
9          self.cnt += 1
```

# Implementation

```
1  @patch
2  def _dfs(self:CC, G:Graph, v:int) -> None:
3    self.marked[v] = True
4    self.ids[v] = self.cnt
5    for w in G.adj(v):
6      if not self.marked[w]:
7        self._dfs(G, w)
```

# Implementation

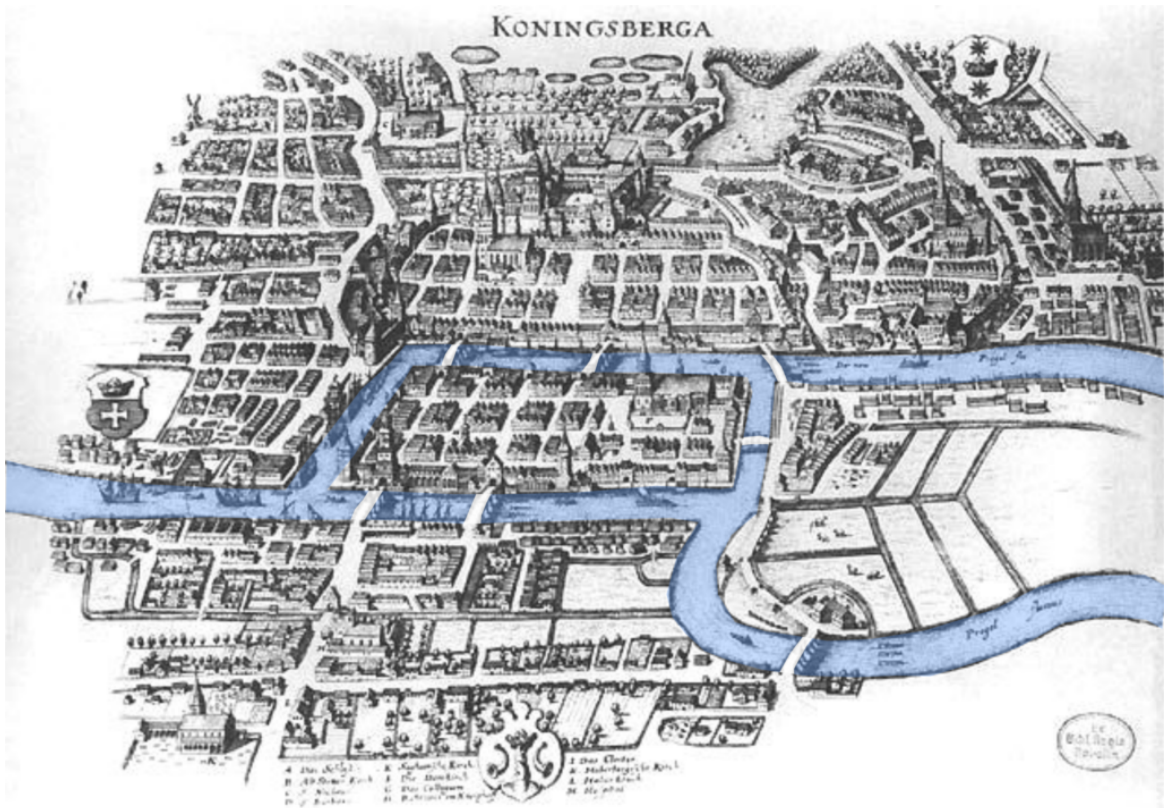
```
 1  @patch(as_prop=True)
 2  def count(self:CC) -> int:
 3    return self.cnt
 4
 5  @patch
 6  def comp_id(self:CC, v:int) -> int:
 7    return self.ids[v]
 8
 9  @patch
10  def get_comp(self:CC, c:int) -> list[int]:
11    return [v for v,cnt in enumerate(self.ids) if cnt == c]
```

# Testing it

```
1 cc = CC(g)
2 assert cc.count == 2
3 assert cc.comp_id(1) == cc.comp_id(9)
4 assert cc.comp_id(14) != cc.comp_id(3)
5 print(cc.get_comp(cc.comp_id(14)))
```

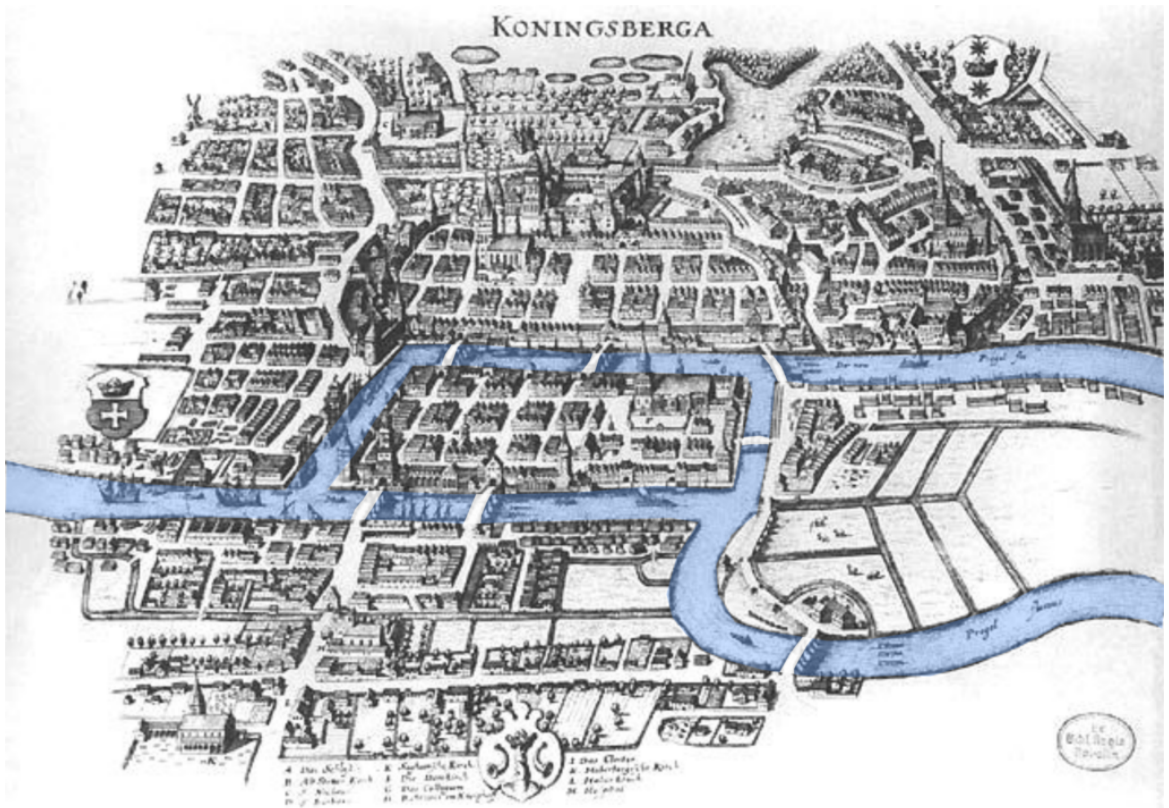
```
[11, 12, 13, 14]
```

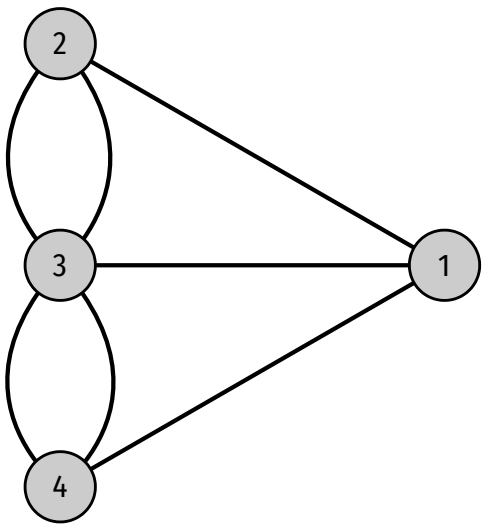
# Fun with graphs (seven bridges)




Can we find a path that crosses each bridge once and returns us to our starting point?

# Fun with graphs (seven bridges)



# Eulerian path

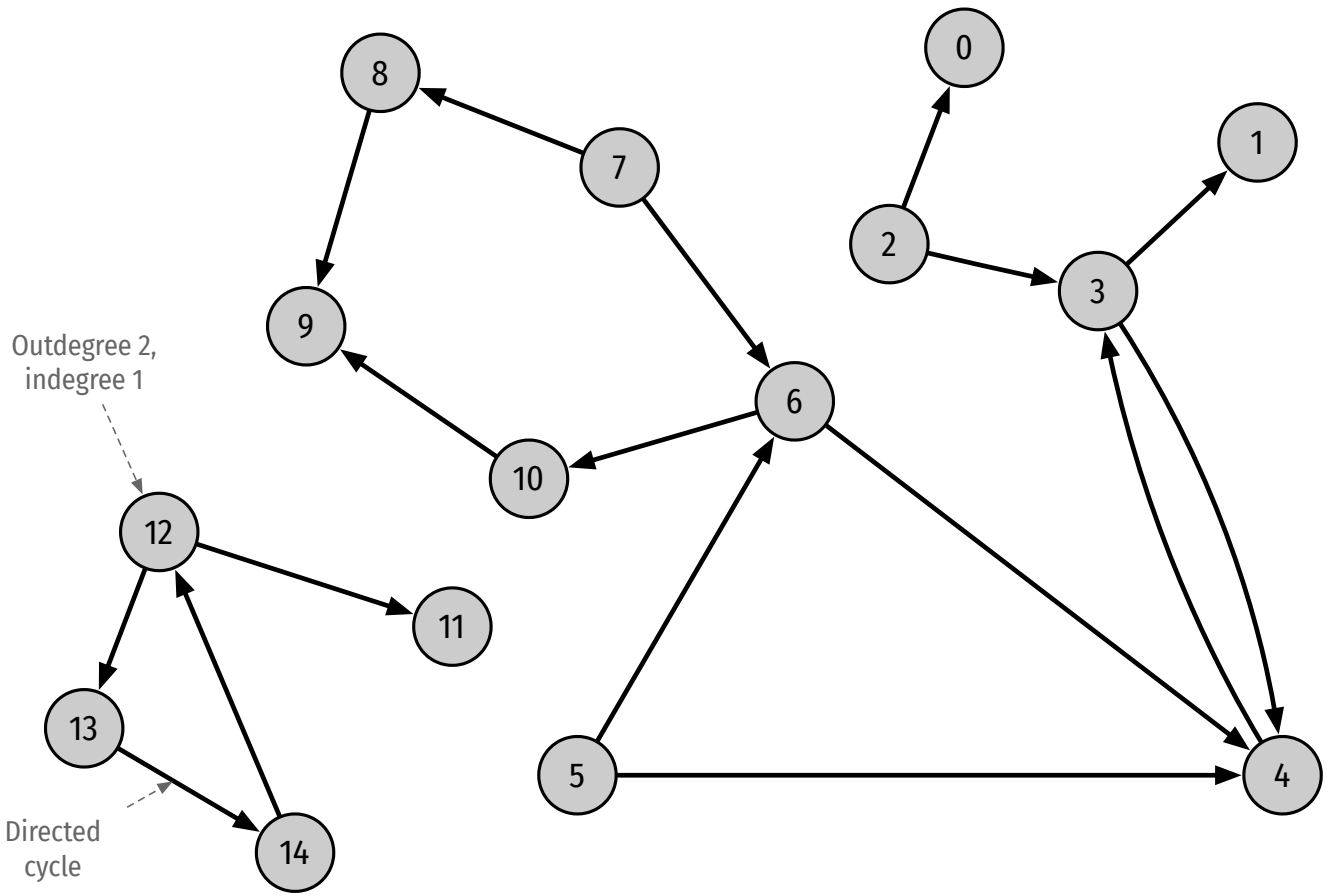
- Find a general cycle that uses every edge exactly once
- Such as cycle is a Eulerian path
- Requires every vertex to have an even degree
- Easy to solve, $O(E)$

# From edges to vertices

- What if we want to find a cycle that visits each vertex exactly once?
- Hamiltonian path
- Intractable (no efficient algorithm exists)

# Directed Graphs

# Directed graph



# Directed graphs

- Use when direction matters
  - One-way streets
  - Hyperlinks
  - Legal move in game
  - Control flow in programs
  - ...

# Implementation and API

- Similar to undirected graphs
- But, now only add edge in the right direction
- Some new operations make sense, e.g., reverse

# Implementation and API

```
 1  class DiGraph:
 2    def __init__(self, v:int) -> None:
 3      self.al = [[] for _ in range(v)]
 4
 5    @property
 6    def V(self) -> int:
 7      return len(self.al)
 8
 9    @property
10    def E(self) -> int:
11      return sum([len(l) for l in self.al])
```

# Reimplementing with adjacency list

```
 1  from fastcore.basics import patch
 2
 3  @patch
 4  def add_edge(self:DiGraph, src:int, dst:int) -> None:
 5    if src < len(self.al) and dst < len(self.al):
 6      self.al[src].append(dst)
 7
 8  @patch
 9  def adj(self:DiGraph, v:int) -> list[int]:
10    if v < len(self.al):
11      return self.al[v]
```

# Testing it

```
 1  dg = DiGraph(15)
 2  dg.add_edge(2, 0)
 3  dg.add_edge(2, 3)
 4  dg.add_edge(3, 1)
 5  dg.add_edge(3, 4)
 6  dg.add_edge(4, 3)
 7  dg.add_edge(5, 4)
 8  dg.add_edge(5, 6)
 9  dg.add_edge(6, 4)
10  dg.add_edge(6, 10)
11  dg.add_edge(7, 6)
12  dg.add_edge(7, 8)
13  dg.add_edge(8, 9)
14  dg.add_edge(10, 9)
15
16  dg.add_edge(12, 11)
17  dg.add_edge(12, 13)
18  dg.add_edge(13, 14)
19  dg.add_edge(14, 12)
20
21  assert dg.E == 17
```

# Searching in directed graphs

- Same as undirected
- Every undirected graph is a directed graph
- With edges in both directs

# Testing it

```
1 dfs = DFSPaths(dg, 6)
2 assert dfs.has_path_to(1)
3 assert not dfs.has_path_to(7)
4 print(dfs.path_to(1))
5 print(dfs.path_to(9))
```

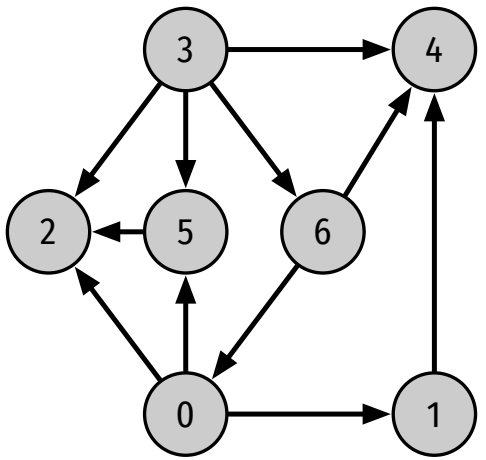
```
[6, 4, 3, 1]
[6, 10, 9]
```

# Topological sorting

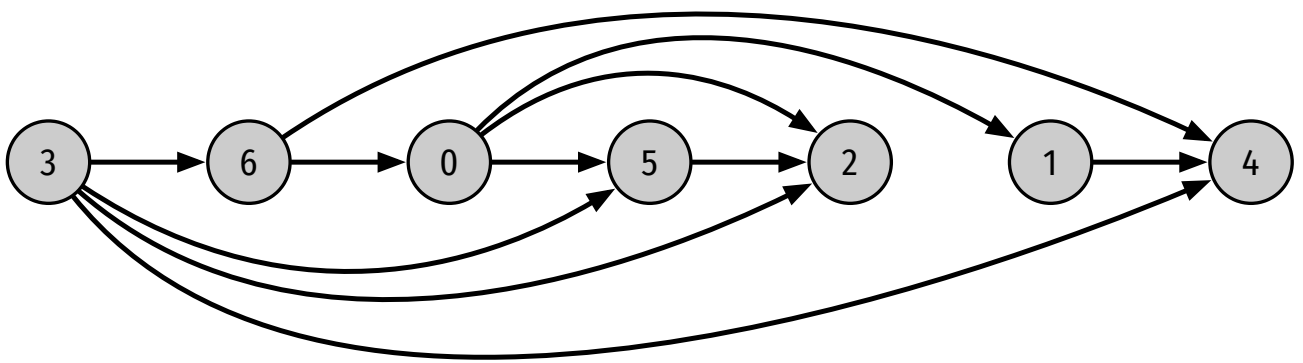
- Problem: Given a set of tasks and precedence constraints, in which order should the tasks be scheduled?
- We can solve this with a directed graph
- Vertices are tasks, edges give precendence constaints

# Example

Graph with precedence constraints




Feasible schedule



# Topological sorting

- Graph must be directed acyclic (DAG)
- We start with a DFS to get a reverse postorder
- Which we then reverse
- Which is the topological order

# Implementing

```
1  class DFO:
2    def __init__(self, G:DiGraph) -> None:
3      self.marked = np.zeros(G.V, dtype=bool)
4      self.post = []
5      for v in range(G.V):
6        if not self.marked[v]:
7          self._dfs(G, v)
```

# Implementing

```
1  @patch
2  def _dfs(self:DFO, G:DiGraph, v:int) -> None:
3    self.marked[v] = True
4    for w in G.adj(v):
5      if not self.marked[w]:
6        self._dfs(G, w)
7    self.post.append(v)
```

# Implementing

```
1  @patch
2  def reverse_post(self:DFO) -> list[int]:
3    return self.post[::-1]
```

# Testing it

```
 1  dg = DiGraph(7)
 2  dg.add_edge(0, 1)
 3  dg.add_edge(0, 2)
 4  dg.add_edge(0, 5)
 5  dg.add_edge(1, 4)
 6  dg.add_edge(3, 2)
 7  dg.add_edge(3, 4)
 8  dg.add_edge(3, 5)
 9  dg.add_edge(3, 6)
10  dg.add_edge(5, 2)
11  dg.add_edge(6, 0)
12  dg.add_edge(6, 4)
```

# Testing it

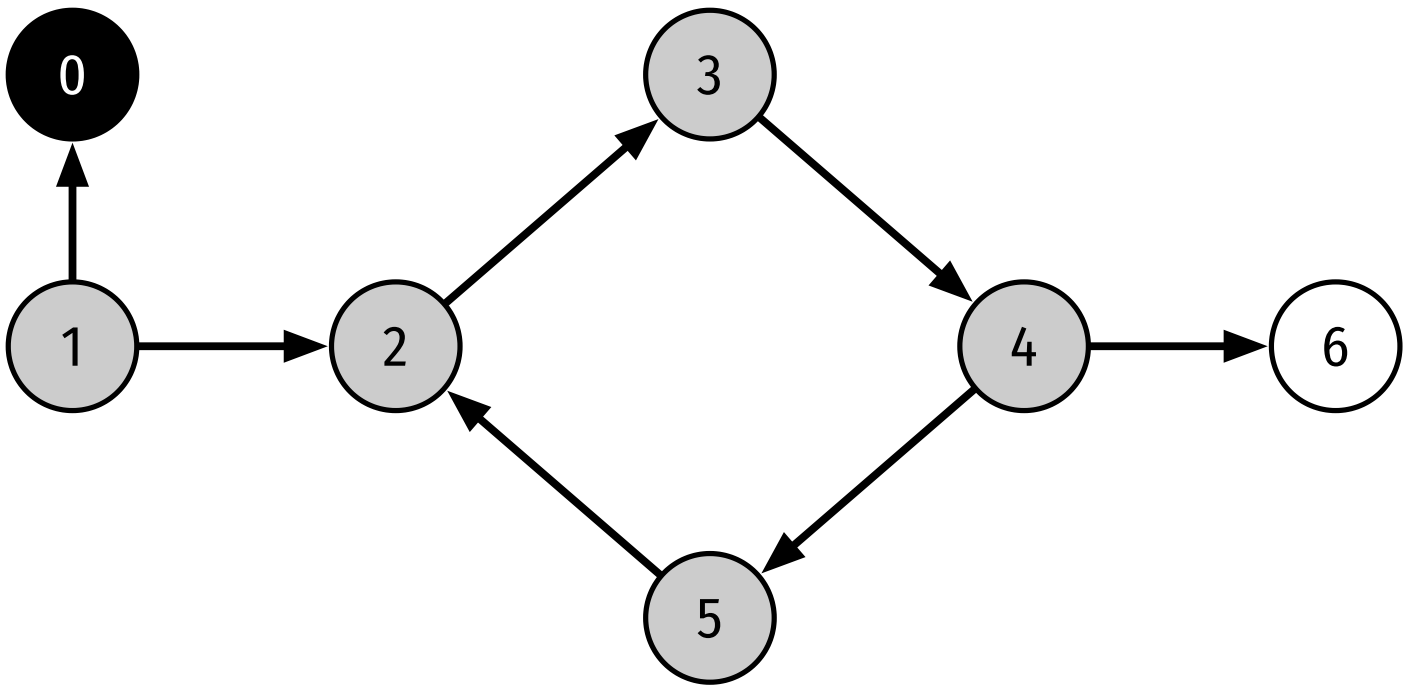
```
1 dfo = DFO(dg)
2 print(dfo.reverse_post())
```

```
[3, 6, 0, 5, 2, 1, 4]
```

# Finding cycles

- Another application of search
- But we need to extend the marked array to now contain colors
- We map false to white and true to black
- And add gray if something is being processed
- If we find a gray node in the adjacency list, then we have a cycle

# Finding cycles



# Implementation

```
1  from enum import IntEnum
2  class Color(IntEnum):
3    WHITE = 0
4    GRAY = 1
5    BLACK = 2
```

# Implementation

```
1 class CF:
2   def __init__(self, G:DiGraph):
3     self.color = np.zeros(G.V, dtype=int)
4     self.G = G
```

# Implementation

```
1  @patch
2  def has_cycle(self:CF) -> bool:
3    for v in range(self.G.V):
4      if self.color[v] == Color.WHITE:
5        if self._dfs(v):
6          return True
7    return False
```

# Implementation

```
@patch
def _dfs(self:CF, v:int) -> bool:
  self.color[v] = Color.GRAY
  for w in self.G.adj(v):
    if self.color[w] == Color.GRAY:
        return True
    if self.color[w] == Color.WHITE and self._dfs(w):
        return True
  self.color[v] = Color.BLACK
  return False
```

# Testing it

```
1  g = Graph(3)
2  g.add_edge(0, 1)
3  g.add_edge(1, 2)
4  g.add_edge(2, 0)
5
6  cf = CF(g)
7  print(cf.has_cycle())
```

```
True
```

# Testing it

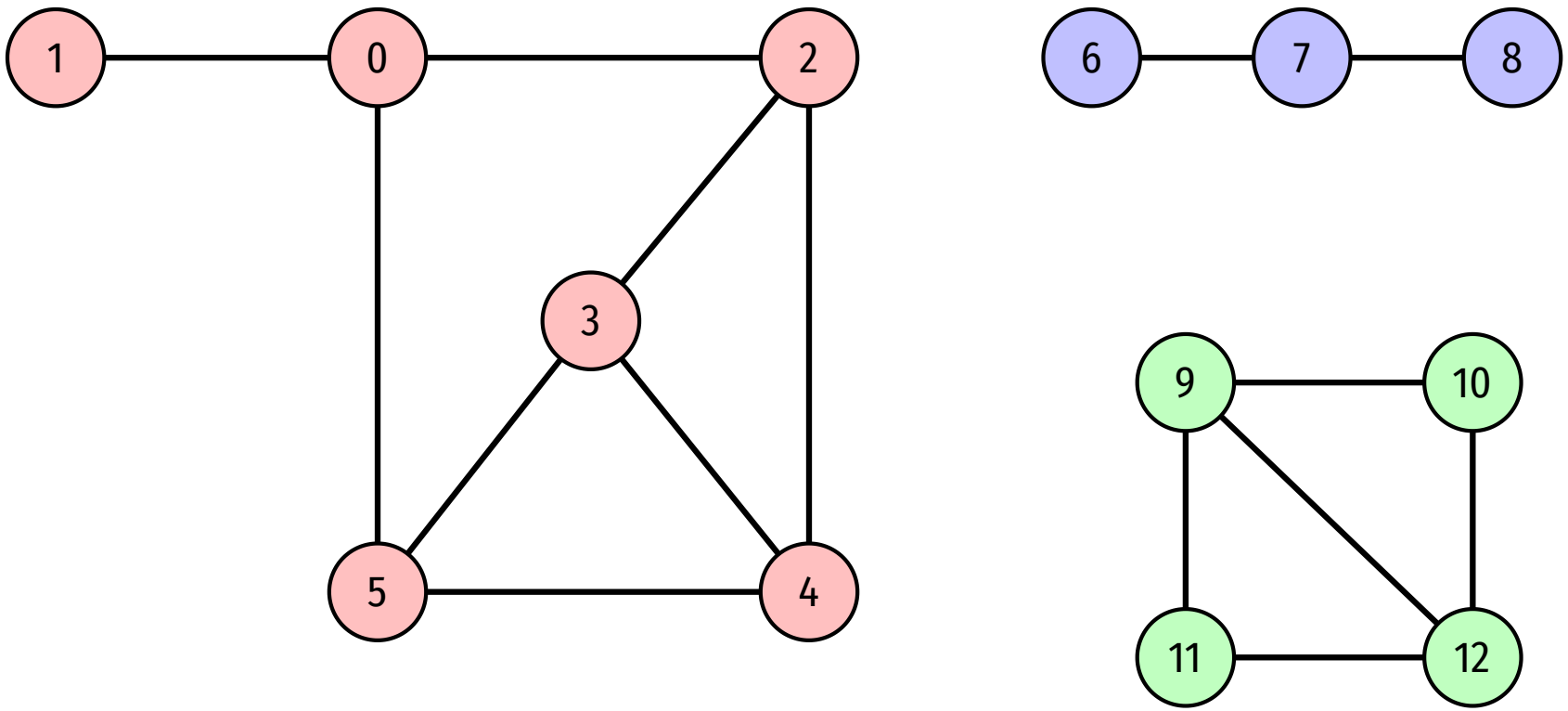
```
1 cf = CF(dg)
2 print(cf.has_cycle())
3
4 dg.add_edge(2, 0)
5 cf = CF(dg)
6 print(cf.has_cycle())
```

```
False
True
```

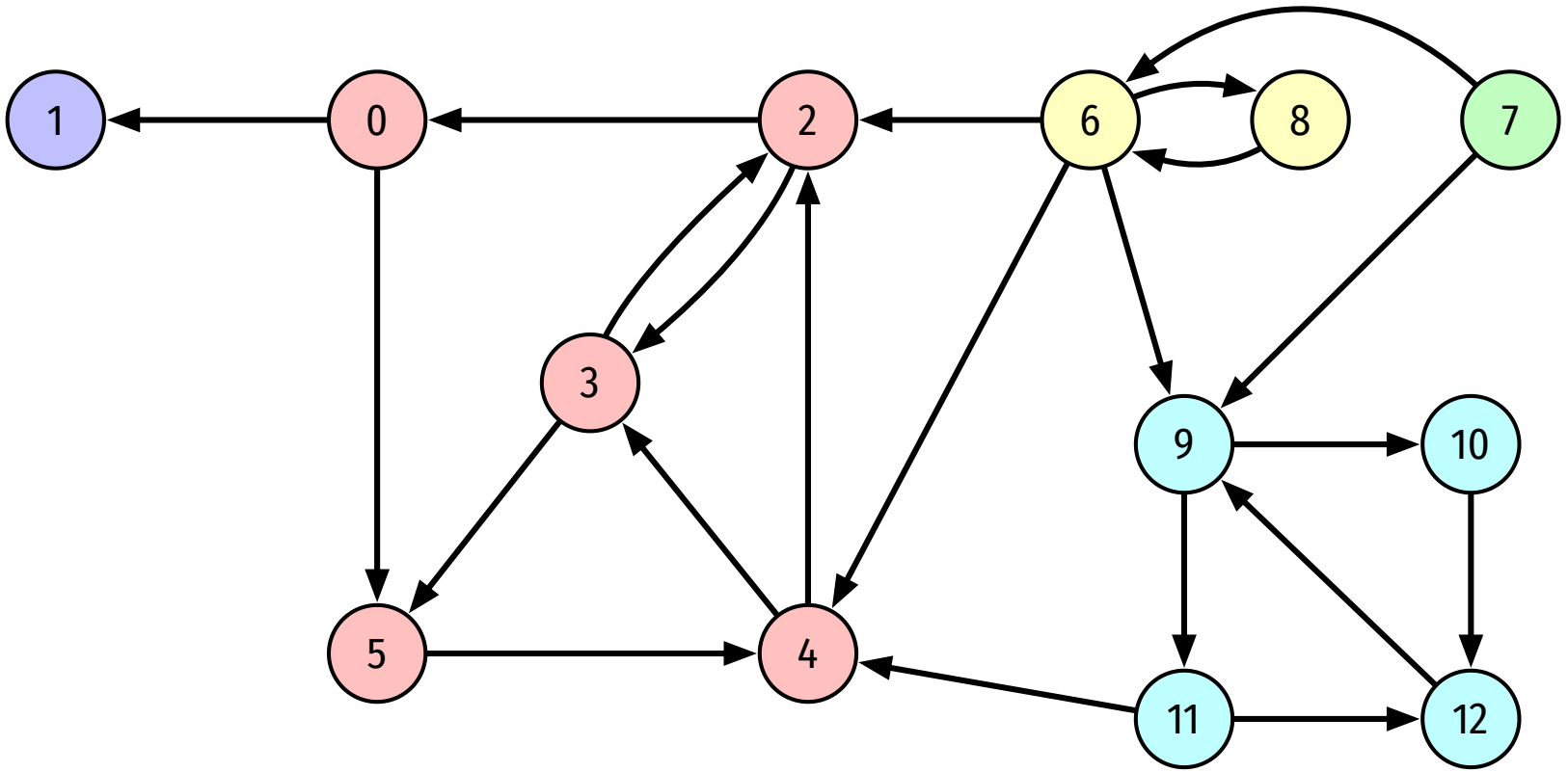
# Strongly-connected components

- Two vertices, *v* and *w* are strongly connected if there are directed paths from *v* to *w* and *w* to *v*
- A strong component is a maximal subset of strongly-connected vertices

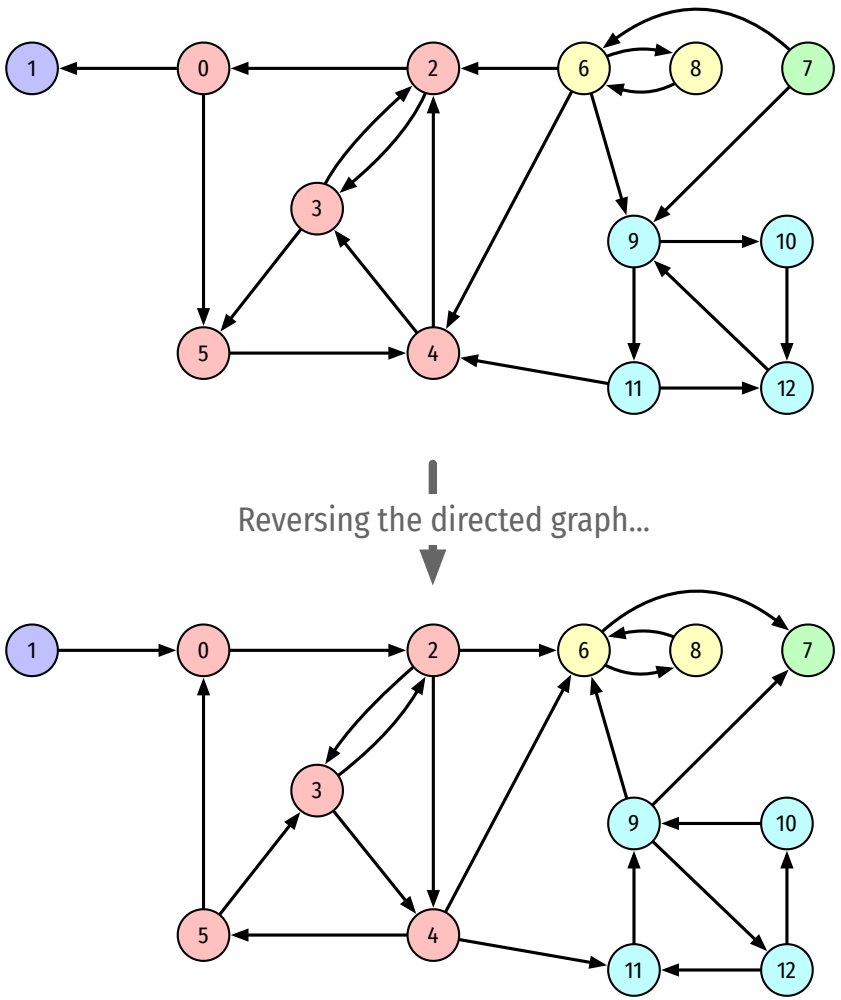
# Remember connected components



# Strongly connected components



# Reversing a graph



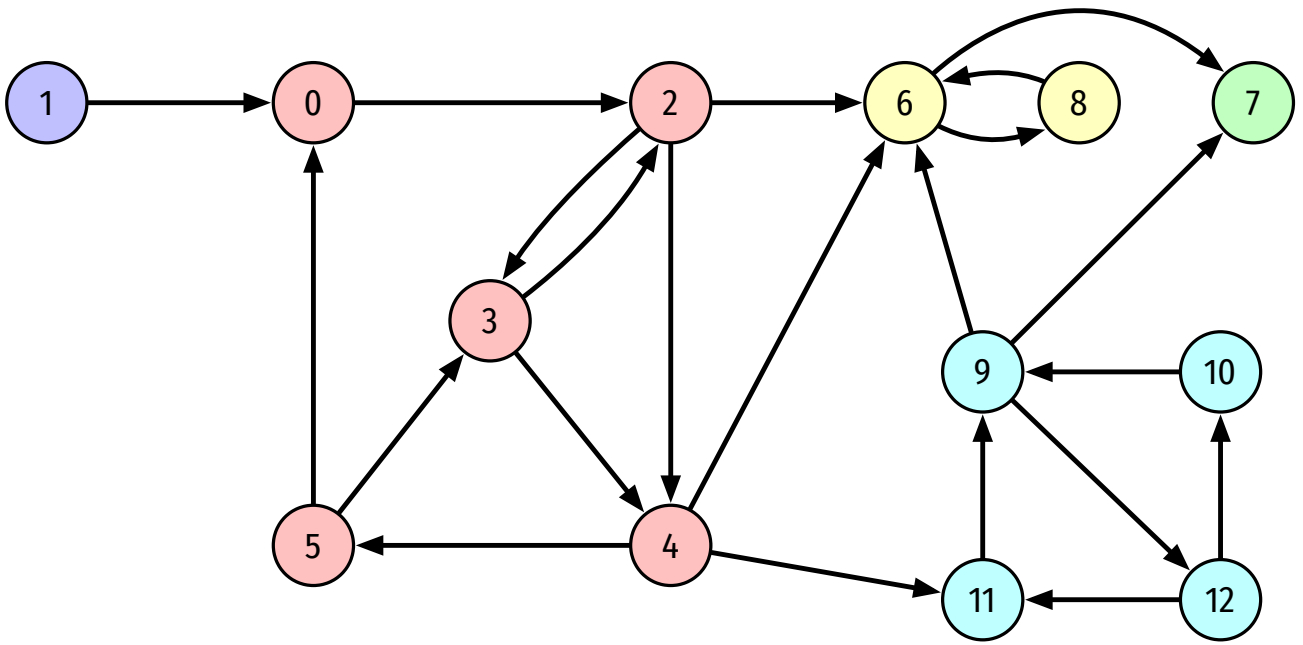
# Reverse G ($G^R$)

```
1  @patch
2  def reverse(self:DiGraph):
3    tmp = DiGraph(self.V)
4    for v in range(self.V):
5      for w in self.adj(v):
6        tmp.add_edge(w, v)
7    return tmp
```

# Observeration and idea

- The strong components are the same in $G$ and $G^R$
- Idea for the algorithm
  1. Compute the topological order on $G^R$
  2. Run dfs on $G$ on nodes in the order from step 1
- Kosaraju-Sharir

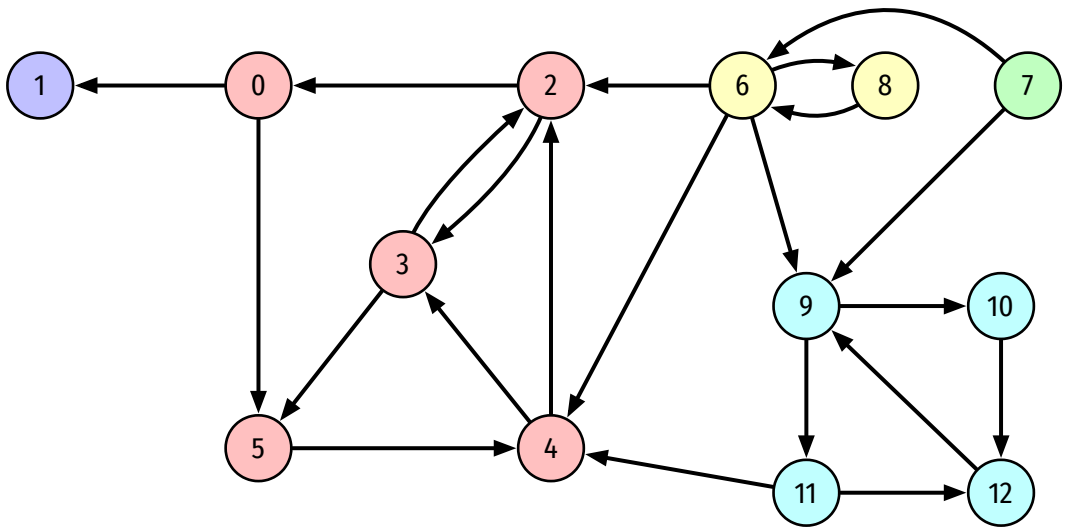
# Compute reverse postorder




Reverse postorder of $G^R$

# Run DFS on the reverse postorder

Reverse postorder of $G^R$


DFS visit order

# Strongly connected components

```
 1  class SCC:
 2    def __init__(self, G:DiGraph) -> None:
 3      self.marked = np.zeros(G.V, dtype=bool)
 4      self.ids = np.zeros(G.V, dtype=int)
 5      self.cnt = 0
 6      dfo = DFO(G.reverse())
 7      for v in dfo.reverse_post():
 8        if not self.marked[v]:
 9          self._dfs(G, v)
10          self.cnt += 1
```

# Strongly connected components

```
1  @patch
2  def _dfs(self:SCC, G:DiGraph, v:int) -> None:
3    self.marked[v] = True
4    self.ids[v] = self.cnt
5    for w in G.adj(v):
6      if not self.marked[w]:
7        self._dfs(G, w)
```

# Strongly connected components

```
 1  @patch
 2  def connected(self:SCC, G:DiGraph, v:int, w:int) -> bool:
 3    return self.ids[v] == self.ids[w]
 4
 5  @patch
 6  def components(self:SCC):
 7    tmp = {}
 8    for i,c in enumerate(self.ids):
 9      if c not in tmp:
10        tmp[c] = []
11      tmp[c].append(i)
12    return [v for k,v in tmp.items()]
```

# Testing it

```
 1  dg = DiGraph(13)
 2  el = [(0, 1), (0, 5), (2, 0), (2, 3), (3, 5), \
 3        (3, 2), (4, 2), (4, 3), (5, 4), (6, 2), \
 4        (6, 4), (6, 8), (6, 9), (7, 6), (7, 9), \
 5        (8, 6), (9, 10), (9, 11), (10, 12), \
 6        (11, 12), (11, 4), (12, 9)]
 7
 8  for s, d in el:
 9    dg.add_edge(s, d)
10
11  scc = SCC(dg)
12  print(scc.components())
```

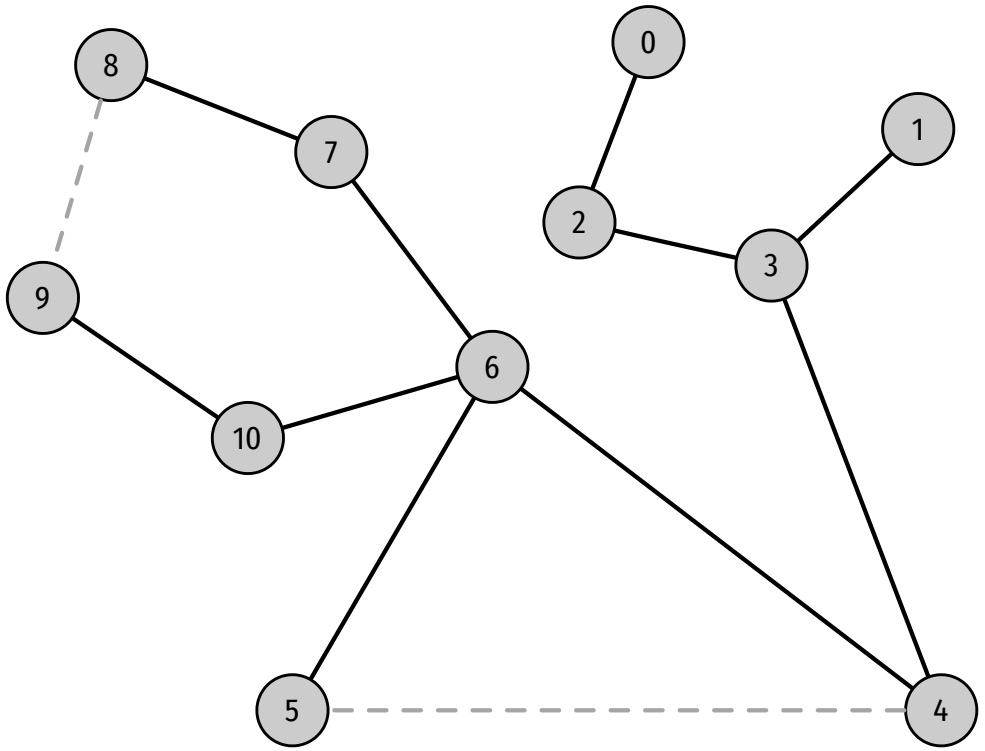
```
[[0, 2, 3, 4, 5], [1], [6, 8], [7], [9, 10, 11, 12]]
```

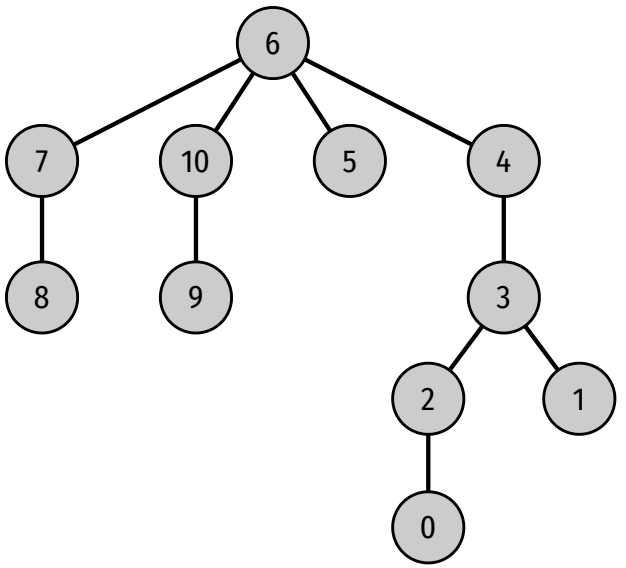
# Spanning trees

# Spanning trees

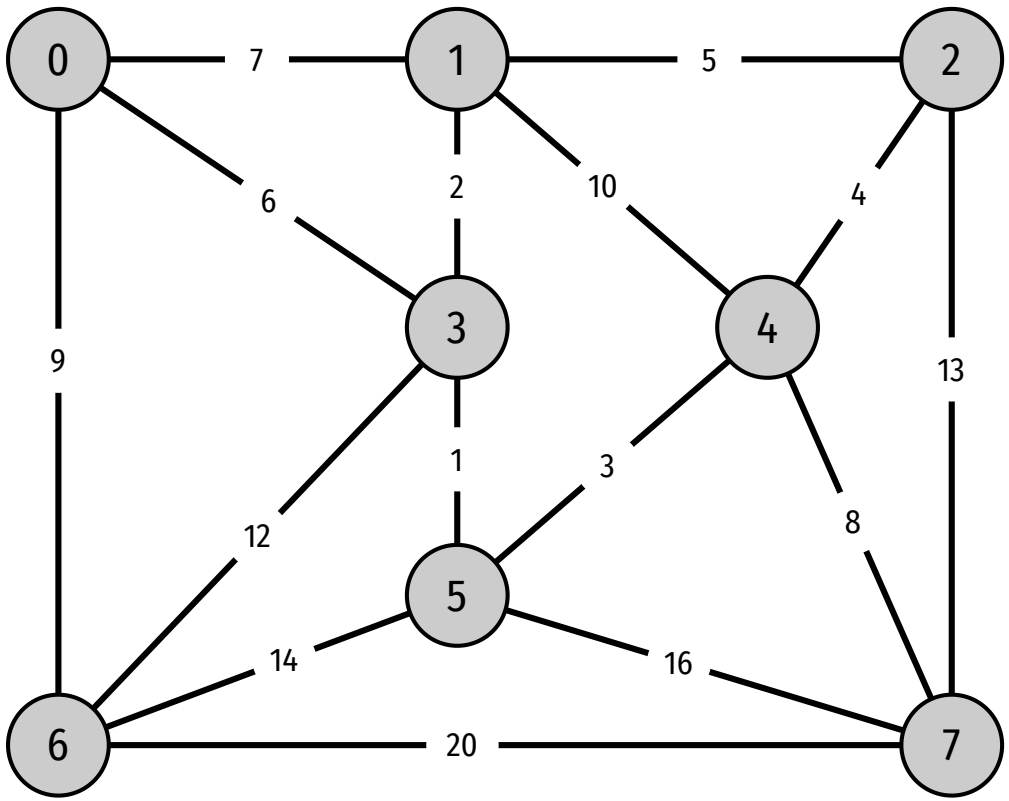
- A spanning tree *T* is a subgraph of an undirected graph *G* which contains all the vertices of *G*
  - Tree, connected and acyclic
  - Spanning, includes all vertices
- *G* must be connected
- There can exist multiple spanning trees
- Applications in path finding

# Spanning tree



# Edge weights



# Adding weigths to our graph

```
 1  class Edge:
 2    def __init__(self, v:int, w:int, weight:float) -> None:
 3      self.v = v
 4      self.w = w
 5      self.weight = weight
 6
 7    def either(self) -> int:
 8      return self.v
 9
10    def other(self, v:int) -> int:
11      return self.w if v == self.v else self.v
```

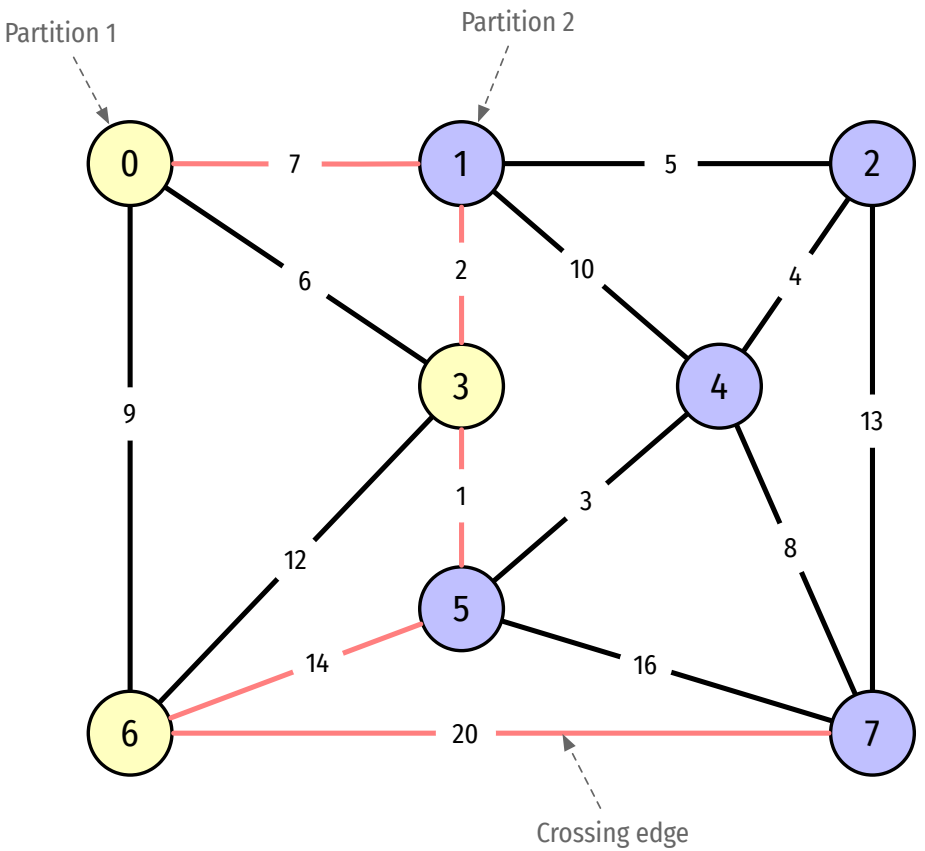
# Minimum spanning trees (MST)

- Assume a connected, undirected graph with positive edge weights
- A minimum spanning tree is the spanning tree with the lowest sum of edge weights

# Cut

- A cut is a way to partition the graph into two sets of vertices
- An edge that crosses the cut connects a vertex in one with with a vertex in the other

# Cut


Partition 1
Partition 2
0
1
2
3
4
5
6
7
7
5
6
2
10
4
9
13
1
3
12
8
14
16
20
Crossing edge

# Minimum spanning trees

- Given any cut, the crossing edge of minimum weight must be in the minimum spanning tree
- So, to compute minimum spanning tree, find a cut that does not have a crossing edge in the current tree
- Add the minimum weight crossing edge to the tree
- Repeat until $V$-1 edges are in the tree

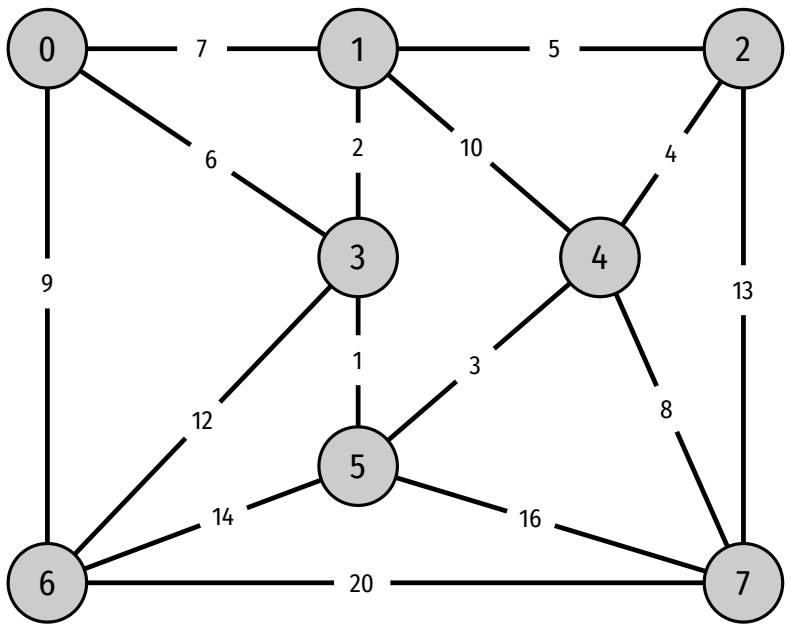
# Greedy algorithm

- A problem-solving strategy
- Pick the best choice at each step
- Can work to find the global optimum

# Kruskal's algorithm

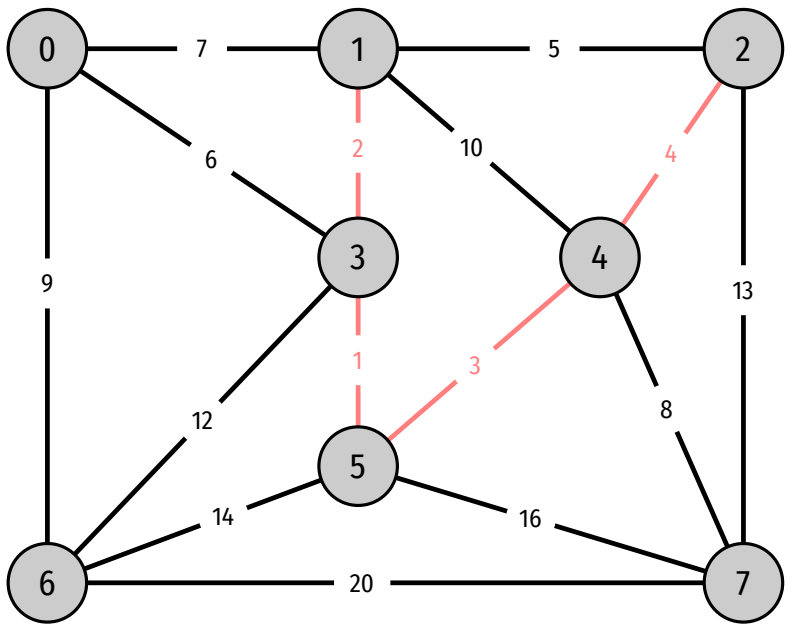
- Sort edges by weight (ascending order)
- Add next edge to the tree if it does not create a cycle
- Easy, except, how do we know if it creates a cycle?

# Example



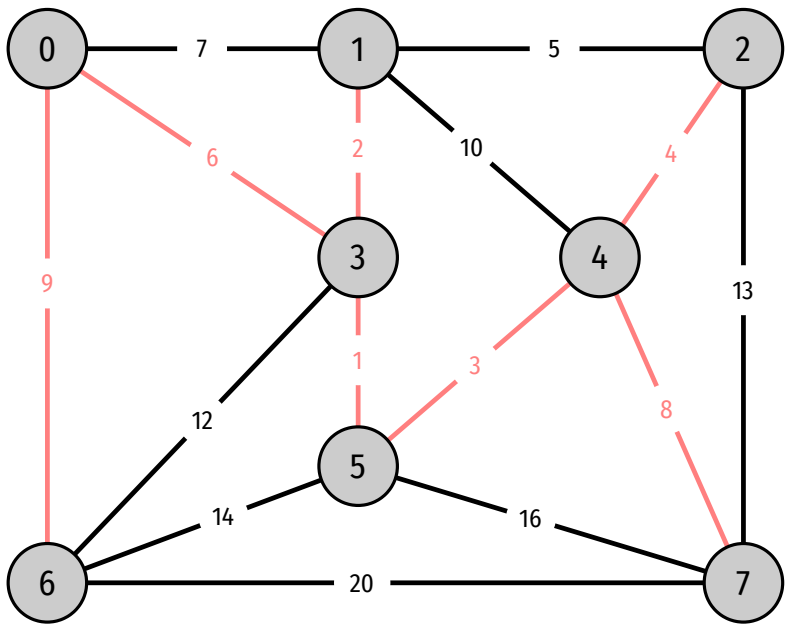

```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Example



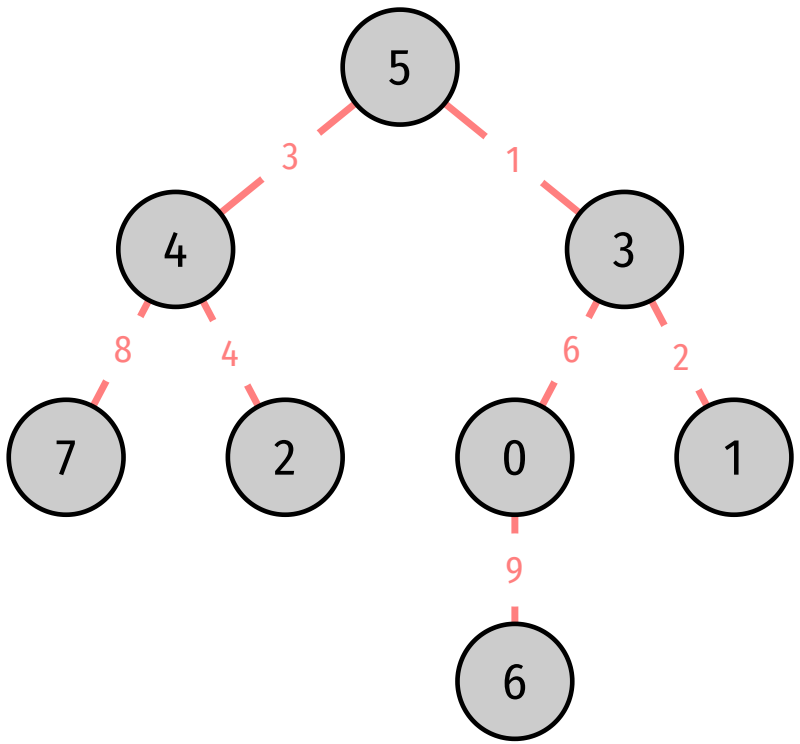

```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Example

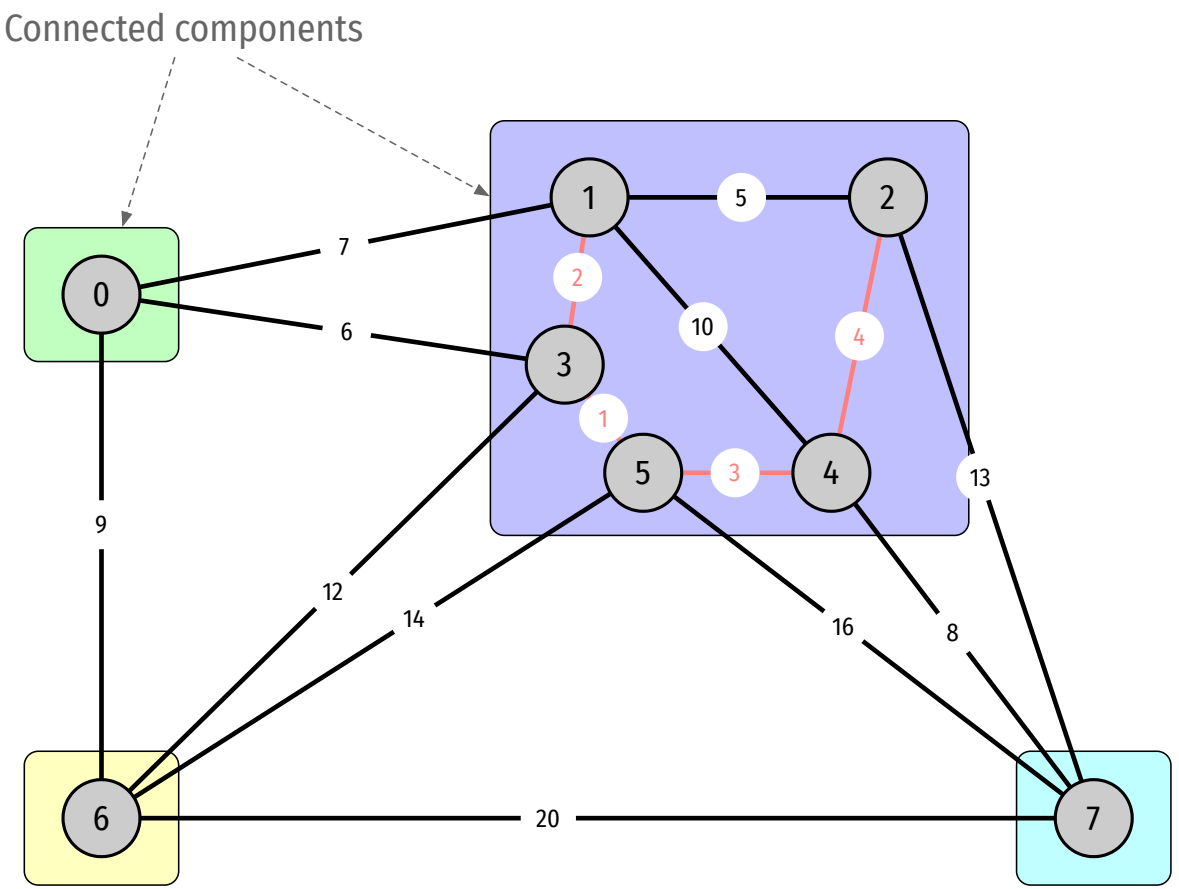



```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Example



# Cycles?



# Union find (Again)

```
 1  class UF:
 2    def __init__(self, N:int) -> None:
 3      self.d = list(range(N))
 4      self.sz = [1]*N
 5
 6    def connected(self, a:int, b:int) -> bool:
 7      return self.root(a) == self.root(b)
 8
 9    def root(self, a:int) -> int:
10      while a != self.d[a]:
11        self.d[a] = self.d[self.d[a]]
12        a = self.d[a]
13      return a
```

# Union find (Again)

```
 1  @patch
 2  def union(self:UF, a:int, b:int) -> None:
 3    ra = self.root(a)
 4    rb = self.root(b)
 5
 6    if self.sz[ra] < self.sz[rb]:
 7      self.d[ra] = rb
 8      self.sz[rb] += self.sz[ra]
 9    else:
10      self.d[rb] = ra
11      self.sz[ra] += self.sz[rb]
```

# Make edges sortable

```
1  @patch
2  def __lt__(self:Edge, other:Edge) -> bool:
3      return self.weight < other.weight
4
5  @patch
6  def __repr__(self:Edge) -> str:
7      return f'Edge({self.v}, {self.w}, {self.weight})'
```

# Really?

```
1  e1 = Edge(2, 3, 9)
2  e2 = Edge(1, 2, 5)
3  e3 = Edge(4, 5, 6)
4
5  print(sorted([e1, e2, e3]))
```

```
[Edge(1, 2, 5), Edge(4, 5, 6), Edge(2, 3, 9)]
```

# Implementation

```
 1  class EWGraph:
 2    def __init__(self, v:int) -> None:
 3      self.v = v
 4      self.al = [[] for _ in range(v)]
 5
 6    @property
 7    def V(self) -> int:
 8      return self.v
 9
10    @property
11    def E(self) -> int:
12      return sum([len(v) for v in self.al]) // 2
```

# Implementation

```
1  @patch
2  def add_edge(self:EWGraph, e:Edge) -> None:
3    v = e.either()
4    w = e.other(v)
5    self.al[v].append(e)
6    self.al[w].append(e)
```

# Implementation

```
1  @patch
2  def adj(self:EWGraph, v:int) -> list[Edge]:
3    return self.al[v]
4
5  @patch
6  def edges(self:EWGraph) -> list[Edge]:
7    return [v for l in self.al for v in l]
```

# Kruskal's algorithm

```
 1  from heapq import heappush, heappop, heapify
 2
 3  class Kruskal:
 4    def __init__(self, G:EWGraph) -> None:
 5      self.mst = []
 6      self.pq = []
 7      self.uf = UF(G.V)
 8      for e in G.edges():
 9        heappush(self.pq, e)
10
11      while self.pq and len(self.mst) < G.V - 1:
12        e = heappop(self.pq)
13        v = e.either()
14        w = e.other(v)
15        if not self.uf.connected(v, w):
16          self.uf.union(v, w)
17          self.mst.append(e)
```

# Kruskal's algorithm

```
1  @patch
2  def edges(self:Kruskal) -> list[Edge]:
3    return self.mst
4
5  @patch(as_prop=True)
6  def weight(self:Kruskal) -> float:
7    return sum([e.weight for e in self.mst])
```

# Testing it

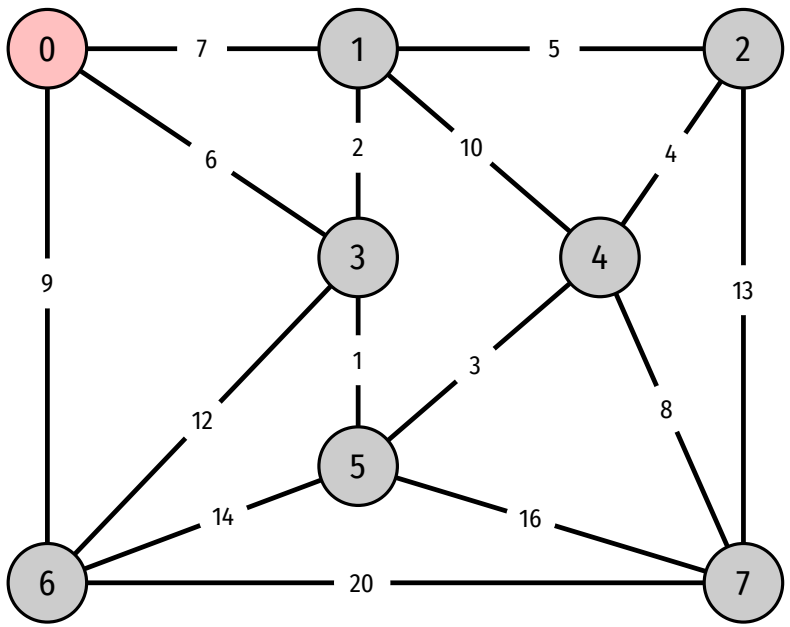
```
 1  g = EWGraph(8)
 2  el = [(0, 1, 7), (0, 3, 6), (0, 6, 9), (1, 2, 5), (1, 3, 2
 3        (1, 4, 10), (2, 4, 4), (2, 7, 13), (3, 5, 1),(3, 6,
 4        (4, 5, 3), (4, 7, 8),(5, 6, 14), (5, 7, 16), (6, 7,
 5
 6  for e in el:
 7    g.add_edge(Edge(*e))
 8
 9  mst = Kruskal(g)
10  print(mst.edges())
11  print(mst.weight)
```

```
[Edge(3, 5, 1), Edge(1, 3, 2), Edge(4, 5, 3), Edge(2, 4, 4),
Edge(0, 3, 6), Edge(4, 7, 8), Edge(0, 6, 9)]
33
```

# Prim's algorithm

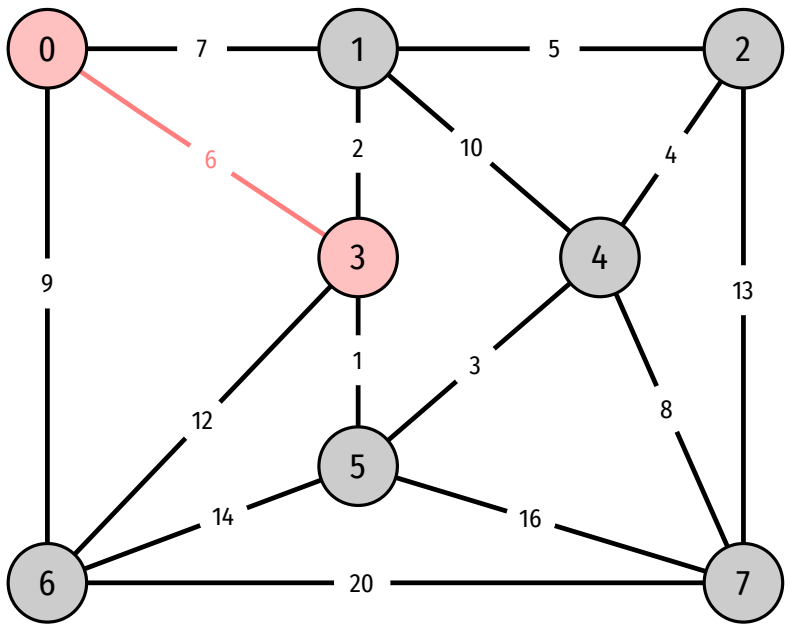
- Start with vertex 0
- Grow the tree, *T*, greedily
  - Add the edge with minimum weight and exactly one endpoint in *T*
  - Repeat until *V-1* edges

# Example



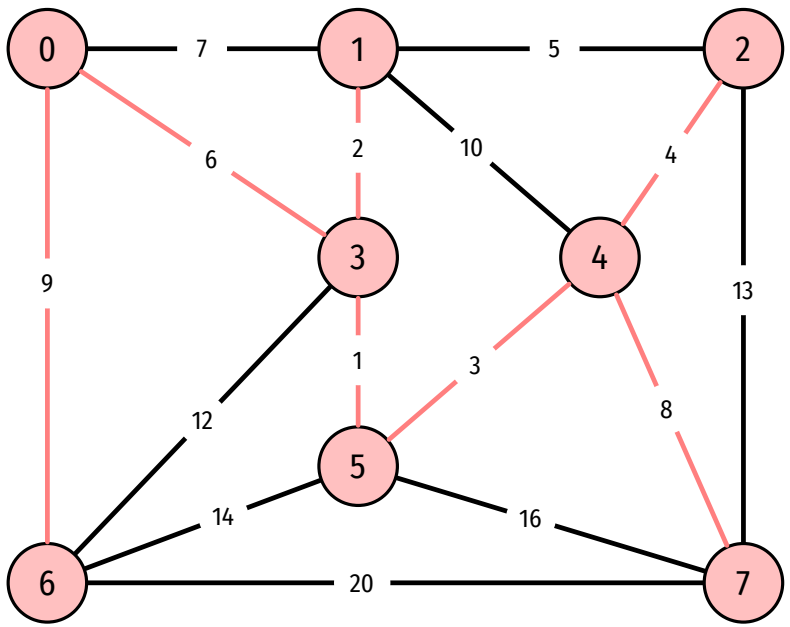

```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Example




```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Example




```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

# Prim's algorithm

```
 1  class Prim:
 2    def __init__(self, G:EWGraph) -> None:
 3      self.mst = []
 4      self.pq = []
 5      self.marked = np.zeros(G.V, dtype=bool)
 6
 7      self._visit(G, 0)
 8
 9      while self.pq and len(self.mst) < G.V - 1:
10        e = heappop(self.pq)
11        v = e.either()
12        w = e.other(v)
13        if self.marked[v] and self.marked[w]:
14          continue
15        self.mst.append(e)
16        if not self.marked[v]:
17          self._visit(G, v)
18        if not self.marked[w]:
19          self._visit(G, w)
```

# Prim's algorithm

```
1 @patch
2 def _visit(self:Prim, G:EWGraph, v:int) -> None:
3   self.marked[v] = True
4   for e in G.adj(v):
5     if not self.marked[e.other(v)]:
6       heappush(self.pq, e)
```

# Prim's algorithm

```
1  @patch
2  def edges(self:Prim) -> list[Edge]:
3    return self.mst
4
5  @patch(as_prop=True)
6  def weight(self:Prim) -> float:
7    return sum([e.weight for e in self.mst])
```

# Testing it

```
1 mstp = Prim(g)
2
3 assert sorted(mst.edges()) == sorted(mstp.edges())
4 assert mst.weight == mstp.weight
```

# Difference?

- With binary heap, Prim is $O(E \log V)$ and Kruskal $O(E \log E)$
- So, Kruskal is better for sparse graphs, Prim is better for dense graphs

# Shortest paths

# Shortest path

- Given a directed graph with (edge) weights, find the shortest part from $s$ to $t$
- Above “source-sink”, can also be
  - Single source to every vertex
  - Between all pairs of vertices
- Cycles and (negative) weights can make it more difficult

# Directed EW Graph

```
 1  class DiEdge:
 2    def __init__(self, v:int, w:int, weight:float) -> None:
 3      self.v = v
 4      self.w = w
 5      self.wght = weight
 6
 7    @property
 8    def src(self) -> int:
 9      return self.v
10
11    @property
12    def dst(self) -> int:
13      return self.w
```

# Directed EW Graph

```
 1  @patch(as_prop=True)
 2  def weight(self:DiEdge) -> float:
 3    return self.wght
 4
 5  @patch
 6  def __lt__(self:DiEdge, other:DiEdge) -> bool:
 7      return self.weight < other.weight
 8
 9  @patch
10  def __repr__(self:DiEdge) -> str:
11      return f'DiEdge({self.v}, {self.w}, {self.weight})'
```

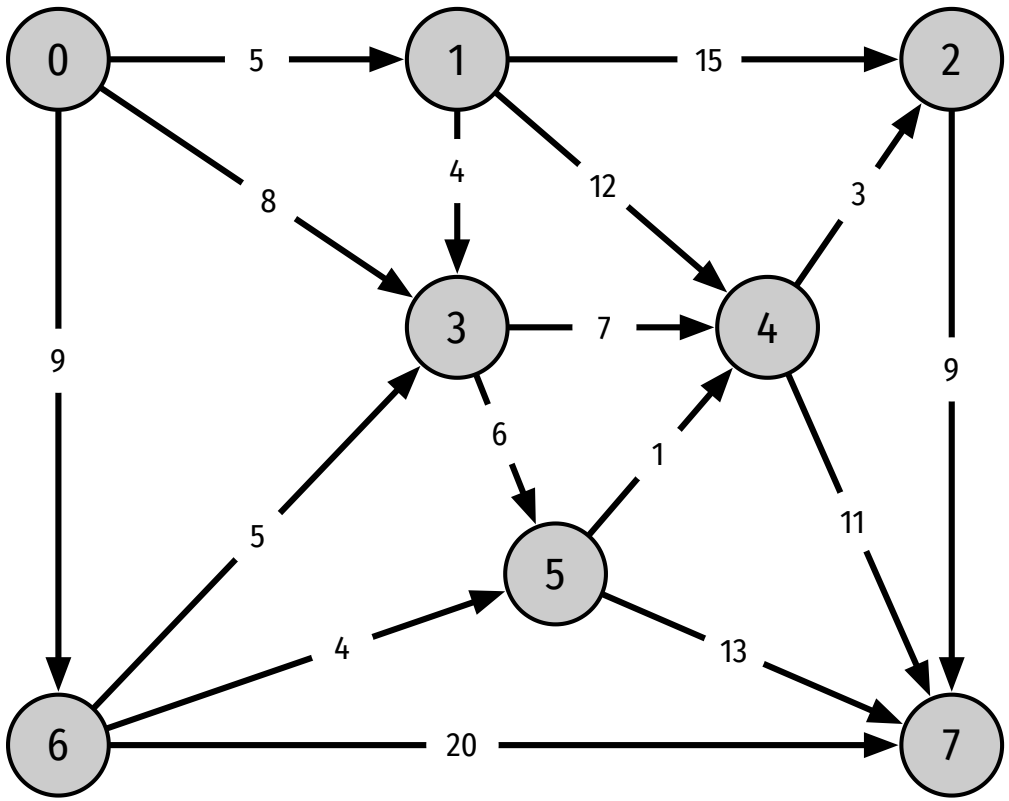
# Directed EW Graph

```
 1  class EWDiGraph:
 2    def __init__(self, v:int) -> None:
 3      self.v = v
 4      self.al = [[] for _ in range(v)]
 5
 6    @property
 7    def V(self) -> int:
 8      return self.v
 9
10    @property
11    def E(self) -> int:
12      return sum([len(v) for v in self.al])
```

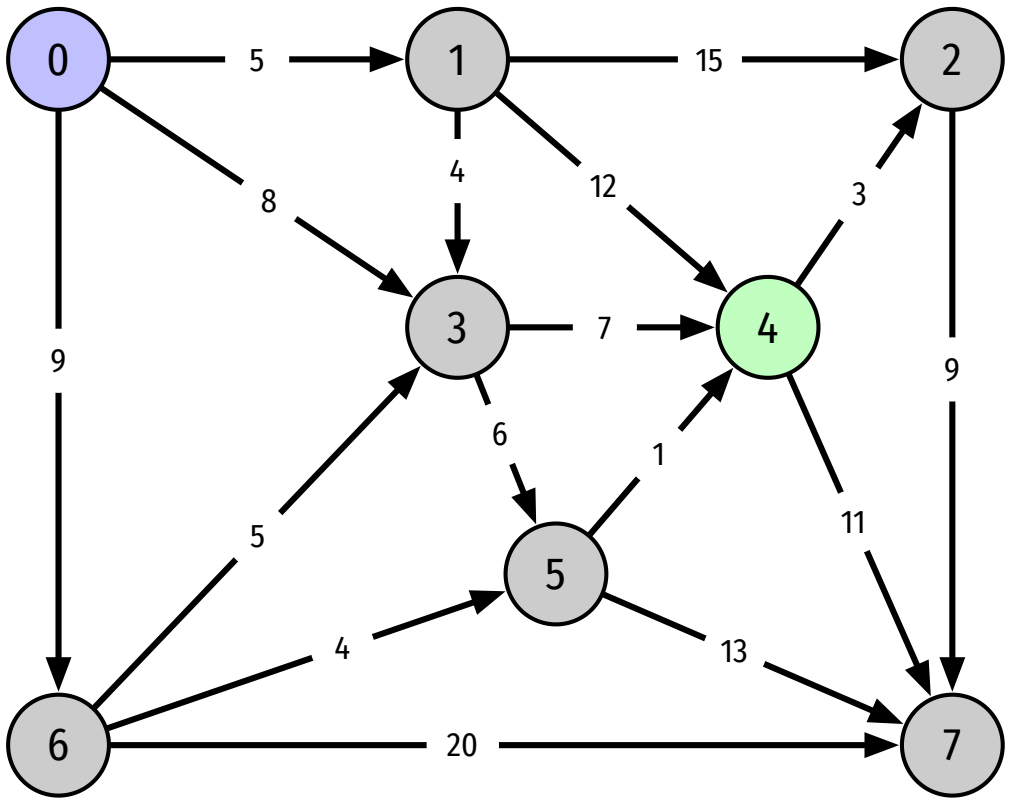
# Directed EW Graph

```
 1  @patch
 2  def add_edge(self:EWDiGraph, e:DiEdge) -> None:
 3    self.al[e.src].append(e)
 4
 5  @patch
 6  def adj(self:EWDiGraph, v:int) -> list[DiEdge]:
 7    return self.al[v]
 8
 9  @patch
10  def edges(self:EWDiGraph) -> list[DiEdge]:
11    return [v for l in self.al for v in l]
```

# Example



# Shortest path from 0 to 4

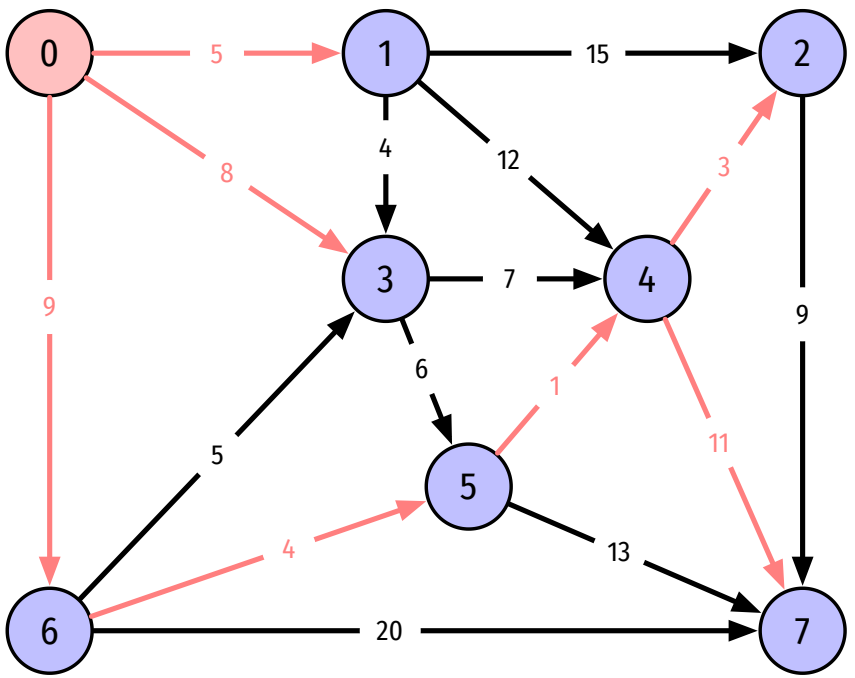


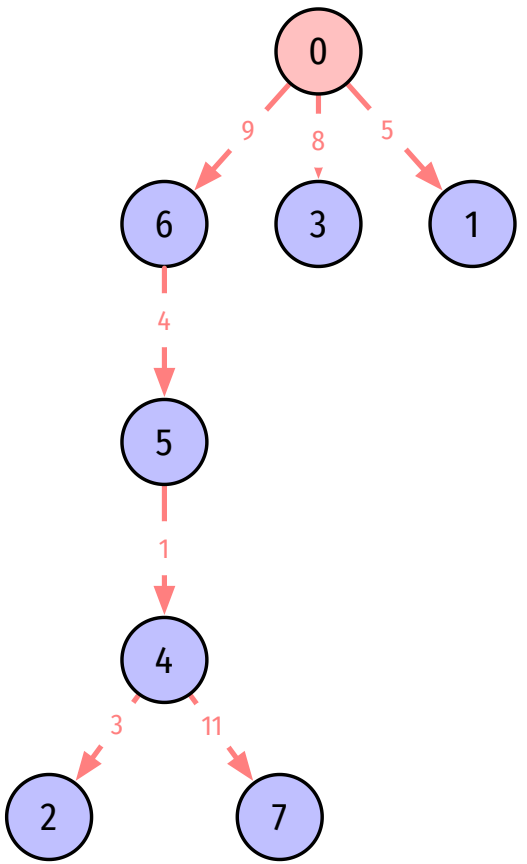

```
14, <0,6,5,4>
15, <0,3,4>
15, <0,3,5,4>
16, <0,1,3,4>
17, <0,1,4>
21, <0,6,3,4>
21, <0,6,3,5,4>
```

# How can we compute?

- Single source shortest paths
- Shortest path from $s$ to every other vertex
- Becomes a (shortest path) tree

# Shortest path tree



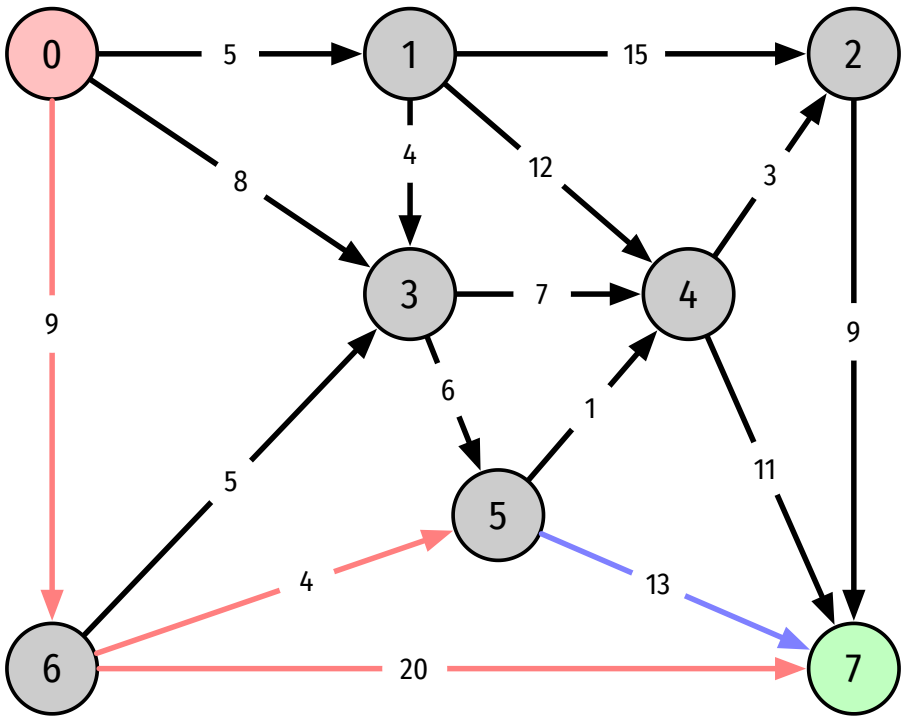
# How can we compute?

- We represent every edge in the tree using parent-link
  - `dist_to[v]` is the lenght of the shortest path from $s$ to $v$
  - `edge_to[v]` is the last edge of the shortest path from $s$ to $v$

# How can we compute?

- We compute by relaxing edges
- Assume we relax the edge *v*, *w*
  - `dist_to[v]` is the shortest *known* path from *s* to *v*
  - `dist_to[w]` is the shortest *known* path from *s* to *w*
  - `edge_to[w]` is the last edge of the shortest *known* path from *s* to *w*
- If *v*, *w* gives a shortest path to *w* through *v*
  - update `dist_to[w]` and `edge_to[w]`

# Relaxing an edge




```
edge_to[5] = <6,5,4>
edge_to[6] = <0,6,9>
edge_to[7] = <6,7,20>


dist_to[5] = 13
dist_to[6] = 9
dist_to[7] = 29


dist_to[5] + 13 < dist_to[7], so:
dist_to[7] = dist_to[5] + 13
edge_to[7] = <5,7,13>
```

# Relaxing an edge

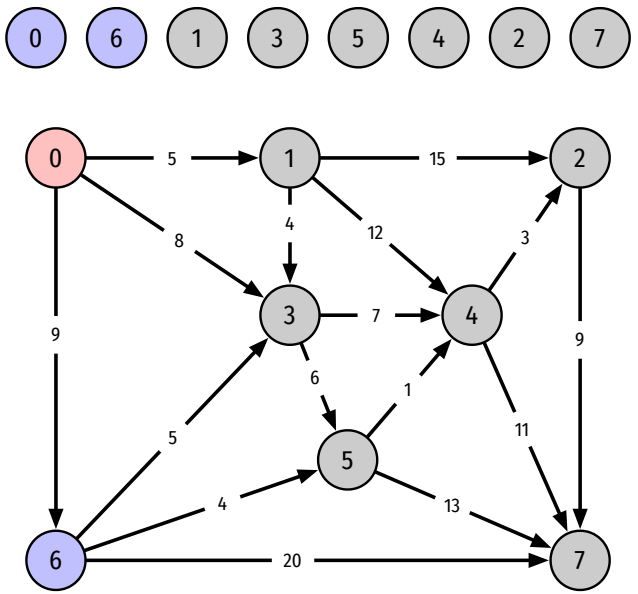
```
1 def relax(e:DiEdge) -> None:
2   if dist_to[e.dst] > dist_to[e.src] + e.weight:
3     dist_to[e.dst] = dist_to[e.src] + e.weight
4     edge_to[e.dst] = e
```

# Starting simple

- In which order should we relax edges?
- Assume
  - No cycles
  - So, Edge-weighted Directed Acyclic Graphs (DAGs)
- We can use topological order to compute shortest paths

# Starting simple

Reverse postorder







```
dist_to[0] = 0
dist_to[1] = 5
dist_to[2] = inf
dist_to[3] = 8
dist_to[4] = inf
dist_to[5] = inf
dist_to[6] = 9
dist_to[7] = inf

edge_to[0] = None
edge_to[1] = <0,1,5>
edge_to[2] = None
edge_to[3] = <0,3,8>
edge_to[4] = None
edge_to[5] = None
edge_to[6] = <0,6,9>
edge_to[7] = None
```

Relaxing edges from vertex 6

```
dist_to[7] > dist_to[6] + 20 ?
dist_to[7] = dist_to[6] + 20
edge_to[7] = <6,7,20>

dist_to[5] > dist_to[6] + 4 ?
dist_to[5] = dist_to[6] + 4
edge_to[5] = <6,5,4>

dist_to[3] > dist_to[6] + 5 ?
```

# Fixing the DFO

```
1  @patch
2  def _dfs(self:DFO, G:DiGraph, v:int) -> None:
3    self.marked[v] = True
4    for e in G.adj(v):
5      if not self.marked[e.dst]:
6        self._dfs(G, e.dst)
7    self.post.append(v)
```

# Implementation

```
 1  class DAGSP:
 2    def __init__(self, G:EWDiGraph, s:int) -> None:
 3      self.edge_to = [None] * G.V
 4      self.dist_to = np.full(G.V, np.inf, dtype=np.double)
 5      self.dist_to[s] = 0.0
 6
 7      dfo = DFO(G)
 8      for v in dfo.reverse_post():
 9        for e in G.adj(v):
10          self._relax(e)
```

# Implementation

```
1  @patch
2  def _relax(self:DAGSP, e:DiEdge) -> None:
3    if self.dist_to[e.dst] > self.dist_to[e.src] + e.weight:
4      self.dist_to[e.dst] = self.dist_to[e.src] + e.weight
5      self.edge_to[e.dst] = e
```

# Example

```
 1  dg = EWDiGraph(8)
 2  el = [(0,1,5), (0,3,8), (0,6,9), (1,2,15), (1,3,4), (1,4,1
 3        (2,7,9), (3,4,7), (3,5,6), (4,2,3), (4,7,11), (5,4,1
 4        (5,7,13), (6,3,5), (6,5,4), (6,7,20)]
 5
 6  for e in el:
 7    dg.add_edge(DiEdge(*e))
 8
 9  sp = DAGSP(dg, 0)
10  assert sp.dist_to[4] == 14.0
11  print(sp.dist_to)
```

```
[ 0.  5. 17.  8. 14. 13.  9. 25.]
```

# Getting the path

```
 1  @patch
 2  def has_path_to(self:DAGSP, v:int) -> bool:
 3    return self.dist_to[v] < np.inf
 4
 5  @patch
 6  def path_to(self:DAGSP, v:int) -> list[DiEdge]:
 7    if not self.has_path_to(v):
 8      return None
 9
10    x, path = self.edge_to[v], []
11    while x != None:
12      path.insert(0, x)
13      x = self.edge_to[x.src]
14    return path
```

# Example

```
1 print([(e.src,e.dst) for e in sp.path_to(4)])
```

```
[(0, 6), (6, 5), (5, 4)]
```

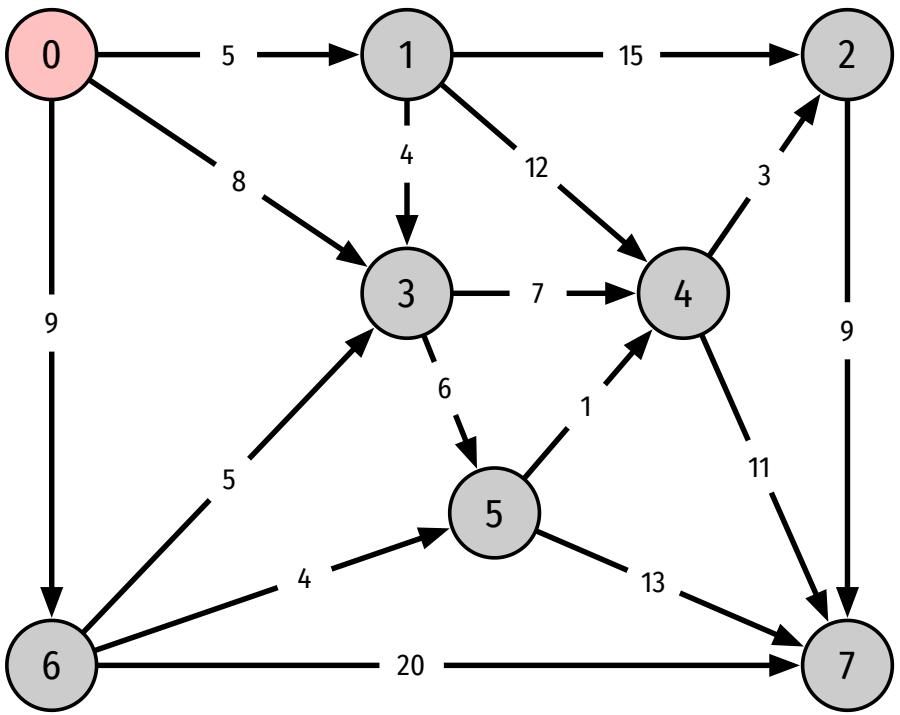
# Cycles

- The simple algorithm
  - No cycles, but positive and negative weights
- What if we have cycles?
  - Dijkstra
- But no negative weights

# Dijkstra's algorithm

- Consider vertices in inceasing distance from the source
- Add vertex to the shortest path tree and relax all outgoing edges from that vertex
- Prim's

# Dijkstra's algorithm




Edge relax order

```
0: <0,1,5> (=0)
<0,3,8> (=0)
<0,6,9> (=0)
1: <1,2,15> (=5)
<1,3,4> (=5)
<1,4,12> (=5)
3: <3,4,7> (=8)
<3,5,6> (=8)
6: <6,3,5> (=9)
<6,5,4> (=9)
<6,7,20> (=9)
5: <5,4,1> (=13)
<5,7,13> (=13)
4: <4,2,3> (=14)
                    <4,7,11> (=
2: <2,7,9> (=17)
7: (=25)
```

# Dijkstra's algorithm

```
 1  class DijkstraSP:
 2    def __init__(self, G:EWDiGraph, s:int) -> None:
 3      self.edge_to = [None] * G.V
 4      self.dist_to = np.full(G.V, np.inf, dtype=np.double)
 5      self.dist_to[s] = s
 6      self.pq = []
 7
 8      heappush(self.pq, (0.0, s))
 9      while self.pq:
10        dst, v = heappop(self.pq)
11        for w in G.adj(v):
12          self._relax(w)
```

# Dijkstra's algorithm

```
 1  @patch
 2  def _relax(self:DijkstraSP, e:DiEdge) -> None:
 3    if self.dist_to[e.dst] > self.dist_to[e.src] + e.weight:
 4      self.dist_to[e.dst] = self.dist_to[e.src] + e.weight
 5      self.edge_to[e.dst] = e
 6      for ix, x in enumerate(self.pq):
 7        if x[1] == e.dst:
 8          self.pq[ix] = (self.dist_to[e.dst], e.dst)
 9          heapify(self.pq)
10          break
11      else:
12        heappush(self.pq, (self.dist_to[e.dst], e.dst))
```

# Testing it

```
1  spd = DijkstraSP(dg, 0)
2
3  assert np.all(sp.dist_to == spd.dist_to)
```

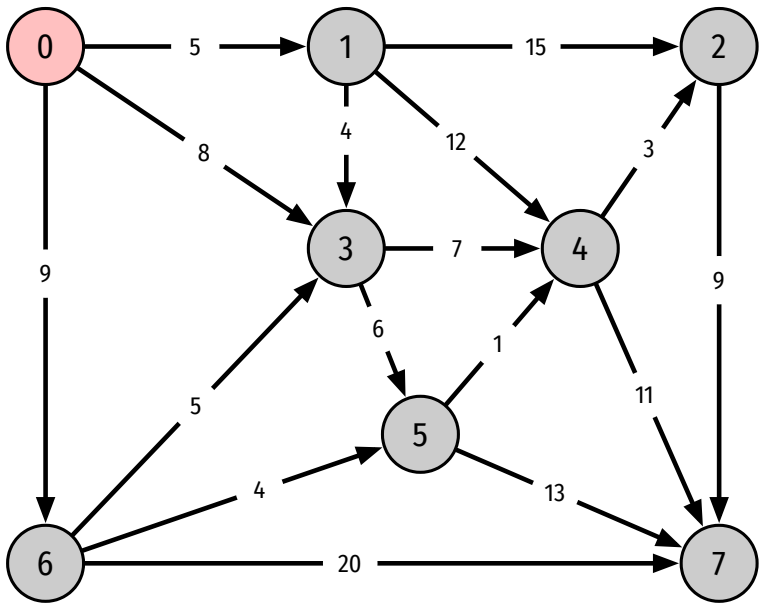
# Bellman-Ford

- What if we have negative weights?
- Bellman-Ford
  - But still no negative cycles

# Bellman-Ford

- Relax all *E* edges *V* times
- Note: if `dist_to[v]` does not change in pass *i*, no need to relax any outgoing edges from *v* in pass *i+1*
- We can use a queue
  - But we cannot have several copies of the same vertex in the queue at the same time!

# Bellman-Ford




Edge relax order

```
0: <0,1,5> -> 1:5
<0,3,8> -> 3:8
<0,6,9> -> 6:9
1: <1,2,15> -> 2:20
<1,3,4>
<1,4,12> -> 4:17
3: <3,4,7> -> 4:15
<3,5,6> -> 5:14
6: <6,3,5>
<6,5,4> -> 5:13
<6,7,20> -> 7:29
2: <2,7,9>
4: <4,2,3> -> 2:18
                    <4,7,11> ->
5: <5,4,1> -> 4:14
<5,7,13>
7:
2: <2,7,9>
4: <4,2,3> -> 2:17
                  <4,7,11> ->
2: <2,7,9>
7:
```

# Bellman-Ford

```
 1  class BFSP:
 2    def __init__(self, G:EWDiGraph, s:int) -> None:
 3      self.dist_to = np.full(G.V, np.inf, dtype=np.double)
 4      self.dist_to[s] = 0.0
 5      self.edge_to = [None] * G.V
 6      self.onqueue = np.zeros(G.V, dtype=bool)
 7      self.onqueue[s] = True
 8      self.queue = [s]
 9
10      while self.queue:
11        v = self.queue.pop()
12        self.onqueue[v] = False
13        self._relax(G, v)
```

# Bellman-Ford

```
 1  @patch
 2  def _relax(self:BFSP, G:EWDiGraph, v:int) -> None:
 3    for e in G.adj(v):
 4      w = e.dst
 5      if self.dist_to[w] > self.dist_to[v] + e.weight:
 6        self.dist_to[w] = self.dist_to[v] + e.weight
 7        self.edge_to[w] = e
 8        if not self.onqueue[w]:
 9          self.queue.insert(0, w)
10          self.onqueue[w] = True
```

# Testing it

```
1  spbf = BFSP(dg, 0)
2
3  assert np.all(sp.dist_to == spbf.dist_to)
```