Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Winter 2022 - Homework 12

Graphs

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form. **This homework is due at 11:59pm on Thursday, 19 May 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

`# YOUR CODE HERE`

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  `assert x == 2`

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

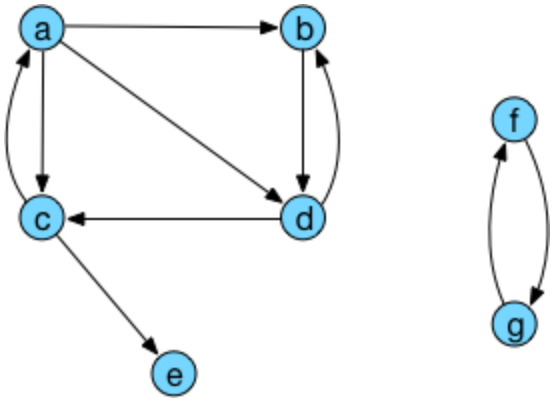
Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

A (directed) graph $G = (V, E)$ consists of a set of vertices (or nodes) V, and a set of edges $E \subseteq V \times V$.



An example of a graph with $V = \{a, b, c, d, e, f, g\}$ and
$E = \{(a, b), (a, c), (a, d), (b, d), (c, a), (c, e), (d, b), (d, c), (f, g), (g, f)\}$.

How should we represent a graph? A general principle of software development -- really, of life -- is: failing special reasons, always go for the simplest solution. So our first attempt consists in storing a graph exactly according to its definition: as a set of vertices and a set of edges.

In [1]:
```
class Graph(object):

    def __init__(self, vertices=None, edges=None):
        # We use set below, just in case somebody passes a list to the initializer.
        self.vertices = set(vertices or [])
        self.edges = set(edges or [])
```

In [2]:
```
g = Graph(vertices={'a', 'b', 'c', 'd', 'e', 'f', 'g'},
          edges={('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'd'),
                 ('c', 'a'), ('c', 'e'), ('d', 'b'), ('d', 'c'),
                 ('f', 'g'), ('g', 'f')})
```

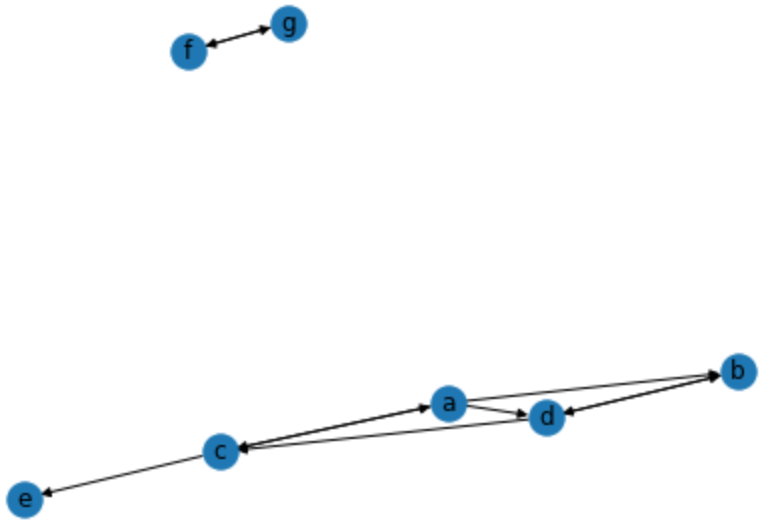
Great, but, how do we display graphs? And what can we do with them?

Let's first of all add a method .show() that will enable us to look at a graph; this uses the library networkx.

In [3]:
```
import networkx as nx # Library for displaying graphs.

class Graph(object):
```

```
    def __init__(self, vertices=None, edges=None):
        # We use set below, just in case somebody passes a list to the initializer.
        self.vertices = set(vertices or [])
        self.edges = set(edges or [])

    def show(self):
        g = nx.DiGraph()
        g.add_nodes_from(self.vertices)
        g.add_edges_from(self.edges)
        nx.draw(g, with_labels=True)
```

In [4]:
```
g = Graph(vertices={'a', 'b', 'c', 'd', 'e', 'f', 'g'},
          edges={('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'd'),
                 ('c', 'a'), ('c', 'e'), ('d', 'b'), ('d', 'c'),
                 ('f', 'g'), ('g', 'f')})
g.show()
```



Ok, this is not nearly as pretty as what we generated by hand, but it will have to do.

One-Step Reachability and Graph Representations

What are conceivable operations on graphs? There are some basic ones, such as adding a vertex and adding an edge. These are easily taken care of.

In [5]:
```
import networkx as nx # Library for displaying graphs.

class Graph(object):

    def __init__(self, vertices=None, edges=None):
        # We use set below, just in case somebody passes a list to the initializer.
        self.vertices = set(vertices or [])
        self.edges = set(edges or [])

    def show(self):
        g = nx.DiGraph()
```

```
        g.add_nodes_from(self.vertices)
        g.add_edges_from(self.edges)
        nx.draw(g, with_labels=True)

    def add_vertex(self, v):
        self.vertices.add(v)

    def add_edge(self, e):
        self.edges.add(e)
```

Further, a graph represents a set of connections between vertices, so a very elementary question to ask is the following: if we are at vertex v, can we get to another vertex u by following one or more edges?

As a first step towards the solution, we want to compute the set of vertices reachable from v in one step, by following one edge; we call these vertices the *successors* of v.

Writing a function g.successors(u) that returns the set of successors of u is simple enough. Note that the code directly mimicks the mathematical definition:

$$Successors(u) = \{v \in V \mid (u, v) \in E\} .$$

In [6]:
```
import networkx as nx # Library for displaying graphs.

class Graph(object):

    def __init__(self, vertices=None, edges=None):
        # We use set below, just in case somebody passes a list to the initializer.
        self.vertices = set(vertices or [])
        self.edges = set(edges or [])

    def show(self):
        g = nx.DiGraph()
        g.add_nodes_from(self.vertices)
        g.add_edges_from(self.edges)
        nx.draw(g, with_labels=True)

    def add_vertex(self, v):
        self.vertices.add(v)

    def add_edge(self, e):
        self.edges.add(e)

    def successors(self, u):
        """Returns the set of successors of vertex u"""
        return {v for v in self.vertices if (u, v) in self.edges}
```

In [7]:
```
g = Graph(vertices={'a', 'b', 'c', 'd', 'e', 'f', 'g'},
          edges={('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'd'),
                 ('c', 'a'), ('c', 'e'), ('d', 'b'), ('d', 'c'),
                 ('f', 'g'), ('g', 'f')})
g.successors('a')
```

Out[7]: {'b', 'c', 'd'}

But there's a rub. The method successors, as written, requires us to loop over the whole set of vertices. Because self.edges is a set, represented as a hash table, once we have a pair (u, v), checking

```
(v, u) in self.edges
```

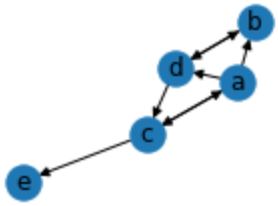
is efficient. But typically, graphs have a locality structure, so that each node is connected only to a small subset of the total vertices; having to loop over all vertices to find the successors of a vertex is a great waste. It is as if I asked you to what places you can get from San Francisco with a direct flight, and to answer, you started to rattle off all of the world's cities, from Aachen, Aalborg, Aarhus, ..., all the way to Zürich, Zuwarah, Zwolle, and for each city you checked if there's a flight from San Francisco to that city! Clearly not the best method.

Given that our main use for graphs is to answer reachability-type questions, a better idea is to store the edges via a dictionary that associates with each vertex the set of successors of the vertex. The vertices will simply be the keys of the dictionary.

In [8]:
```
import networkx as nx # Library for displaying graphs.

class Graph(object):

    def __init__(self, vertices=None, edges=None):
        self.s = {u: set() for u in vertices or []}
        for u, v in (edges or []):
            self.add_edge((u, v))

    def show(self):
        g = nx.DiGraph()
        g.add_nodes_from(self.s.keys())
        g.add_edges_from([(u, v) for u in self.s for v in self.s[u]])
        nx.draw(g, with_labels=True)

    def add_vertex(self, v):
        if v not in self.s:
            self.s[v] = set()

    def add_edge(self, e):
        u, v = e
        self.add_vertex(u)
        self.add_vertex(v)
        self.s[u].add(v)

    @property
    def vertices(self):
        return set(self.s.keys())

    def successors(self, u):
        """Returns the set of successors of vertex u"""
        return self.s[u]
```

In [9]:
```
g = Graph(vertices={'a', 'b', 'c', 'd', 'e', 'f', 'g'},
          edges={('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'd'),
                 ('c', 'a'), ('c', 'e'), ('d', 'b'), ('d', 'c'),
                 ('f', 'g'), ('g', 'f')})
g.show()
print(g.successors('a'))
```

```
{'b', 'd', 'c'}
```



Graph Reachability

We now come to one of the fundamental graph algorithms, in fact, perhaps *the* most fundamental algorithm for graphs: computing the set of vertices reachable from a given starting vertex. Exploring what is reachable from a graph vertex is a truly basic task, and variations on the algorithm can be used to answer related questions, such as whether a vertex is reachable from a given starting vertex.

The algorithm keeps two sets of vertices:

- The set of *open* vertices: these are the vertices that are known to be reachable, and whose successors have not yet been explored.
- The set of *closed* vertices: these are the vertices that are known to be reachable, and whose successors we have already explored.

Intially, the set of open vertices contains only the starting vertex, and the set of closed vertices is empty, as we have completed no exploration. Repeatedly, we pick an open vertex, we move it to the closed set, and we put all its successor vertices -- except those that are closed already -- in the open set. The algorithm continues until there are no more open vertices; at that point, the set of reachable vertices is equal to the closed vertices.

If there is *one* graph algorithm that you must learn by heart, and that you should be able to write even when you hang upside down from monkeybars, this is it.

Let us write the algorithm as a function first.

In [14]:
```
def reachable(g, v):
    """Given a graph g, and a starting vertex v, returns the set of states
    reachable from v in g."""
    vopen = {v}
    vclosed = set()
    while len(vopen) > 0:
        u = vopen.pop()
        vclosed.add(u)
        vopen.update(g.successors(u) - vclosed)
    return vclosed
```

In [16]:
```
print(reachable(g, 'a'))
print(reachable(g, 'g'))
```

```
{'c', 'e', 'b', 'd', 'a'}
{'g', 'f'}
```

To visualize the algorithm, let us write a version where at each iteration, open vertices are drawn in red and closed ones in green

In [12]:
```
def reachable(g, v):
    """Given a graph g, and a starting vertex v, returns the set of states
    reachable from v in g."""
    vopen = {v}
    vclosed = set()
    while len(vopen) > 0:
        u = vopen.pop()
        vclosed.add(u)
        vopen.update(g.successors(u) - vclosed)
    return vclosed
```

In [15]:
```
reachable(g, 'a')
```

Out[15]:
```
{'a', 'b', 'c', 'd', 'e'}
```

Breadth-First and Depth-First Search

Breadth First

In **breadth-first** search, we explore in concentric circles emanating from the starting point: first all vertices at distance 1, then all vertices at distance 2, and so on. In general, we explore all vertices at distance $\leq n$ before we explore vertices at distances $> n$.

To implement breadth-first search, we store the open vertices `vopen` as a list rather than a set. We then explore vertices in the order they have been added to `vopen`: this ensures that vertices closer to the search origin are explored earlier than farther-away vertices.

The difference in code between reachability search, and its specialized breadth-first version, is minimal.

In [17]:
```
def breath_first(g, v):
    """Given a graph g, and a starting vertex v, returns the set of states
    reachable from v in g."""
    # vopen is a FIFO: first in, first out. Like a normal queue.
    # we add elements from the end, and pop them from the beginning.
    vopen = [v]
    vclosed = set()
    while len(vopen) > 0:
        u = vopen.pop(0) # Pop from the beginning
        vclosed.add(u)
        # vopen.update(g.successors(u) - vclosed)
        for w in g.successors(u) - vclosed:
            if w not in vopen:
                vopen.append(w) # Add to the end
    return vclosed
```

In [18]:
```
gg = Graph(vertices={},
           edges={('a', 'b'), ('b', 'c'), ('c', 'd'),
                  ('a', 'u'), ('u', 'v'), ('v', 'w'), ('u', 'z')})
```

In [19]:
```
breath_first(gg, 'a')
```

Out[19]:
```
{'a', 'b', 'c', 'd', 'u', 'v', 'w', 'z'}
```

We see that we explore b and u before any of their successors are explored, and similarly, we explore z, v, and c before d or w.

Depth-First Search

In **depth-first** search, we follow a path as long as possible, and only when we come to an end do we explore other nodes. In depth-first search, the most recent visited vertex, the one added last to the list of open vertices, is the one that will be explored first.

The difference in code from breadth-first search is minimal. In breadth-first search, the vertex to be explored next is the *oldest* among the open ones:

```
u = vopen.pop(0)
```

In depth-first search, it will be the *newest* among the open ones:

```
u = vopen.pop()
```

That's the whole difference.

In [20]:
```
def depth_first(g, v):
    """Given a graph g, and a starting vertex v, returns the set of states
```

```
    reachable from v in g."""
    # vopen is a stack / LIFO: last in, first out. Like a stack.
    # we add elements from the end, and pop them from the end.
    vopen = [v]
    vclosed = set()
    while len(vopen) > 0:
        u = vopen.pop() # THIS is the difference: there is no 0 in the parentheses.
        vclosed.add(u)
        # vopen.update(g.successors(u) - vclosed)
        for w in g.successors(u) - vclosed:
            if w not in vopen:
                vopen.append(w)
    return vclosed
```

In [21]:
```
depth_first(gg, 'a')
```

Out[21]:
```
{'a', 'b', 'c', 'd', 'u', 'v', 'w', 'z'}
```

We see how in depth-first search we explore completely one side of the successors of a, consisting of u, v, w, z, before exploring the other side b, c, d.

Problem 1: Returning the edges

In our latest implementation, we do not have direct access to the edges of the graph. In other words, for a graph g, we cannot do:

```
for (u, v) in g.edges:
    ...
```

We ask you to write an iterator over edges, to make the above code work. The iterator should yield the edges of the graph, one by one.

In [32]:
```
### An iterator for the set of edges

def graph_edges(self):
    """Yields the edges of the graph, one by one.  Each edge is yielded as a
    pair of vertices (source, destination). """
    # YOUR CODE HERE
    for p in self.s:
      for p2 in self.s[p]:
        yield (p, p2)


Graph.edges = property(graph_edges)
```

In [34]:
```
### Here you can play with your code.
e = [(1, 2), (1, 3), (2, 3)]
g = Graph(vertices=[1, 2, 3], edges=e)
print(set(g.edges))
```

```
{(1, 2), (1, 3), (2, 3)}
```

Here are some tests.

In [35]:
```
### simple tests

e = [(1, 2), (1, 3), (2, 3)]
g = Graph(vertices=[1, 2, 3], edges=e)
assert set(g.edges) == set(e)

import types
# You need to build a generator, one of those things with the yield statement.
assert isinstance(g.edges, types.GeneratorType)
```

Here are some randomized test.

In [36]:
```
### 10 points: random tests

import random

for _ in range(10):
    num_vertices = random.randint(4, 10)
    num_edges = random.randint(1, num_vertices * num_vertices)
    vertices = random.sample(range(0, 1000), num_vertices)
    edges = {(random.choice(vertices), random.choice(vertices)) for _ in range(num_
    g = Graph(vertices=vertices, edges=edges)
    assert set(g.edges) == edges
```

Problem 2: Is a graph a tree?

A tree is a graph (V, E) with two special properties:

- Every vertex has at most one incoming edge.
- Either there are no vertices, or there is a vertex with no incoming edges, called the *root*, from which all other vertices are reachable.

If the second property does not hold, incidentally, the graph is called a *forest*.

Write an `is_tree` function such that `is_tree(g)` returns True if the graph `g` is a tree, False otherwise.

In [62]:
```
 ### Implementation of tree test

def is_tree(g):
    """Returns True iff the graph is a tree."""
    # YOUR CODE HERE
    branches = {key: 0 for key in g.s}
    for u, v in g.edges:
      branches[v] += 1
    roots = 0
    for key in branches:
      if branches[key] == 0:
        roots += 1
```

```
      elif branches[key] > 1:
        return False
    if g.s == {}:
      return True
    elif roots != 1:
      return False
    else:
      return True
```

In [64]:
```
### Here you can play with your code
g = Graph()
print(is_tree(g))
```

```
True
```

In [63]:
```
 ### 10 points: Tests for `is_tree`

g = Graph(vertices=[1, 2, 3], edges=[(1, 2), (1, 3)])
assert is_tree(g)
g = Graph(vertices=[1, 2, 3], edges=[(1, 2), (2, 3), (1, 3)])
assert not is_tree(g)
g = Graph(vertices=[1, 2, 3], edges=[(1, 3), (2, 3)])
assert not is_tree(g)
```

In [65]:
```
### 10 points: More tests for `is_tree`

g = Graph()
assert is_tree(g)
```

Problem 3: Reachability using either of two graphs

In this problem, you are given *two* graphs `g1`, `g2`, that share the same set of vertices. You have to write a function `can_reach(v, g1, g2, w)`, which returns `True` iff you can go from vertex `v` to vertex `w` using either edges of `g1` or `g2`. Note that to go from `v` to `w`, you can use one or more edges from `g1` and one or more edges of `g2`, mixed in any way you like. To solve the problem, you have to modify the reachability algorithms so that edges from either graph can be used.

Hint: Modify the reachability algorithm.

In [66]:
```
def can_reach(v, g1, g2, w):
    """Given two graphs g1, g2 that share the same vertices, and two verteces v, w,
    returns True if you can go from v to w using edges of either g1 or g2 (mixed an
    way you want) and False otherwise."""
    # YOUR CODE HERE
    vopen = {v}
    vclosed = set()
```

```
    new_edges = set.union(set(g1.edges), set(g2.edges))
    new_g = Graph(vertices = g1.vertices, edges = list(new_edges))
    while len(vopen) > 0:
      n = vopen.pop()
      vclosed.add(n)
      vopen.update(new_g.successors(n) - vclosed)
    print(vclosed)
    if w in vclosed: return True
    else: return False
```

In [67]:
```
### Here you can play with your code.
vertices = {1, 2, 3, 4, 5, 6, 7}
g1 = Graph(vertices=vertices, edges=[(1, 2), (3, 4)])
g2 = Graph(vertices=vertices, edges=[(2, 3), (4, 5), (6, 7)])
can_reach(1, g1, g2, 3)
```

```
{1, 2, 3, 4, 5}
```

Out[67]: True

In [68]:
```
### 10 points: simple tests for can_reach

vertices = {1, 2, 3, 4, 5, 6, 7}
g1 = Graph(vertices=vertices, edges=[(1, 2), (3, 4)])
g2 = Graph(vertices=vertices, edges=[(2, 3), (4, 5), (6, 7)])
assert can_reach(1, g1, g2, 2)
assert can_reach(1, g1, g2, 3)
assert can_reach(1, g1, g2, 4)
assert can_reach(1, g1, g2, 5)
assert not can_reach(1, g1, g2, 6)
assert not can_reach(1, g1, g2, 7)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

In [69]:
```
### 10 points: more advanced tests for can_reach

vertices = set(range(100))
# g1 edges go from n to 2n, g2 edges go from n to 3n.
g1 = Graph(vertices=vertices, edges=[(n, 2 * n) for n in range(100) if 2 * n < 100])
g2 = Graph(vertices=vertices, edges=[(n, 3 * n) for n in range(100) if 3 * n < 100])
assert can_reach(1, g1, g2, 6)
assert can_reach(1, g1, g2, 24)
assert can_reach(1, g1, g2, 32)
assert can_reach(1, g1, g2, 9)
assert not can_reach(1, g1, g2, 15)
assert not can_reach(1, g1, g2, 60)
assert can_reach(5, g1, g2, 15)
assert can_reach(5, g1, g2, 30)
```

```
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 64, 72, 81, 96}
{5, 40, 10, 45, 15, 80, 20, 90, 60, 30}
{5, 40, 10, 45, 15, 80, 20, 90, 60, 30}
```

In [69]: