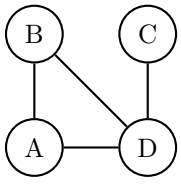
1 Breadth-first Search

Lab Objective: *Shortest path problems are an important part of graph theory and network analysis. Applications include finding the fastest way to drive between two points on a map, network routing, genealogy, automated circuit layout, and a variety of other important problems. In this lab we represent graphs as adjacency dictionaries, implement a shortest path algorithm based on a breadth-first search, and use the NetworkX package to solve a shortest path problem on a large network of movies and actors.*

Adjacency Dictionaries

Computers can represent mathematical graphs in various ways. Graphs with very specific structures are often stored with specialized data structures, such as binary search trees. More general graphs without structural constraints are usually represented with an *adjacency matrix*, where each row and column of the matrix corresponds to a node in the graph, and the entries indicate connections between nodes. Adjacency matrices are usually implemented in a sparse matrix format since only the entries corresponding to node connections are nonzero.

Another common graph data structure is an *adjacency dictionary*, a dictionary with a key for each node in the graph. The dictionary values are the set of nodes connected to the key node. Adjacency dictionaries automatically gain the advantages of a sparse matrix format since they only store information on the actual node connections (the nonzero entries of the adjacency matrix).



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right] \end{array}$$

$$\begin{aligned} &\{\text{A}: \{\text{B, D}\},\\ &\ \text{B}: \{\text{A, D}\},\\ &\ \text{C}: \{\text{D}\},\\ &\ \text{D}: \{\text{A, B, D}\}\} \end{aligned}$$

Figure 1.1: A simple unweighted graph (left), its adjacency matrix (middle), and its adjacency dictionary (right). The graph is undirected, so the adjacency matrix is symmetric. Note that the adjacency dictionary also encodes this behavior: since A and B are connected, B is in the set of values corresponding to the key A, and A is in the set of values corresponding to the key B.

Hash-based Data Structures

A Python `set` is an unordered data type with no repeated elements. The set class is implemented as a *hash table*, meaning it uses *hash values*—integers that uniquely identify an object—to organize its elements. Roughly speaking, in order to access, add, or remove an object `x` to a set, Python computes the hash value of `x`, and that value indicates where `x` is (or should be) in memory. In other words, there is only one place in memory that `x` could be; if it isn't in that place, it isn't in the set. This implementation results in $O(1)$ lookup, insertion, and removal operations, an enormous improvement over the $O(n)$ search time for lists and the $O(\log n)$ search time for sorted structures like binary search trees. It is also why set elements are unique.

Method	Description
`add()`	Add an element to the set. This has no effect if the element is already present.
`remove()`	Remove an element from the set, raising a `KeyError` if it is not a member of the set.
`discard()`	Remove an element from the set without raising an exception if it is not a member of the set.
`pop()`	Remove and return an arbitrary set element.
`union()`	Return all elements that are in either set as a new set.
`intersection()`	Return all elements that are in both sets as a new set.
`update()`	Add all elements of another set in-place.

Table 1.1: Basic methods of the `set` class.

```
# Initialize a set. Note that repeats are not added.
>>> animals = {"cow", "cat", "dog", "mouse", "cow"}
>>> print(animals)
{'cow', 'dog', 'mouse', 'cat'}

>>> animals.add("horse")     # Add an object to the set.
>>> "horse" in animals
True
>>> animals.remove("emu")    # Attempt to delete an object from the set,
KeyError: 'emu'              # resulting in an exception.
>>> animals.pop()            # Delete and return a random object from the set.
'mouse'
>>> print(animals)
{'cat', 'horse', 'dog', 'cow'}

# Add all of the elements of another set to this one.
>>> animals.update({"dog", "velociraptor"})
>>> print(animals)
{'velociraptor', 'cat', 'horse', 'dog', 'cow'}

# Intersect this set with another one.
>>> animals.intersection({"cat", "cow", "cheetah"})
{'cat', 'cow'}
```

Sets are extremely fast, but they do not support indexing because the elements are unordered. A Python `dict`, on the other hand, is a hash-based data structure that stores key-value pairs: the keys of a dictionary act like a set (unique and unordered, with $O(1)$ lookup), but each key corresponds to another object, called its value. The keys index the dictionary and allow $O(1)$ lookup of the values.

Method	Description
`keys()`	Return a set-like iterator for the dictionary's keys.
`values()`	Return a set-like iterator for the dictionary's values.
`items()`	Return an iterator for the dictionary's key-value pairs.
`pop()`	Remove a specified key and return the corresponding value, raising a `KeyError` if the key is not a member of the dictionary.
`update()`	Add or overwrite key-value pairs in-place with those from another dictionary.

Table 1.2: Basic methods of the `dict` class.

```
# Initialize a dictionary.
>>> grades = {"business": "A", "math": "A+", "visual arts": "B"}
>>> grades["math"]
'A+'                              # The key "math" maps to the value "A+".

# Add a "science" key with corresponding value "A".
>>> grades["science"] = "A"

# Remove the "business" key.
>>> grades.pop("business")
'A'
>>> print(grades)
{'math': 'A+', 'visual arts': 'B', 'science': 'A'}

# Display the keys, values, and items.
>>> list(grades.keys()), list(grades.values())
(['math', 'visual arts', 'science'], ['A+', 'B', 'A'])
>>> for key, value in grades.items():
...   print(key, "=>", value)
...
math => A+
visual arts => B
science => A

# Add key-value pairs from another dictionary.
>>> grades.update({"cooking":"A+", "math": "C"})
>>> print(grades)
{'math': 'C', 'visual arts': 'B', 'science': 'A', 'cooking': 'A+'}
```

Dictionaries are ideal for storing values that need to be accessed often and for representing one-to-one or one-to-many relationships. Thus, the `dict` class is a natural choice for implementing adjacency dictionaries. For example, the following code defines the adjacency dictionary for the graph in Figure 1.1. Note that the dictionary values are sets.

```
>>> adjacency = {'A': {'B', 'D'},
                 'B': {'A', 'D'},
                 'C': {'D'},
                 'D': {'A', 'B', 'C'}}

# The nodes of the graph are the dictionary keys.
>>> set(adjacency.keys())
{'B', 'D', 'A', 'C'}

# The values are the nodes that the key node is adjacent to.
>>> adjacency['A']
{'B', 'D'}                  # A is adjacent to B and D.
>>> 'C' in adjacency['B']
False                       # B and C are not adjacent.
>>> 'C' in adjacency['D']
True                        # C and D are adjacent.
```

Achtung!

Elements of a `set` and keys of a `dict` must be *hashable*. Mutable objects—lists, sets and dictionaries—are not hashable, so they are not allowed as set elements or dictionary keys. Thus, in order to represent a graph with an adjacency dictionary, each of the node labels should a string, a number, or some other hashable type.

Problem 1. Consider the following `Graph` class.

```
class Graph:
    """A graph object, stored as an adjacency dictionary. Each node in the
    graph is a key in the dictionary. The value of each key is a set of
    the corresponding node's neighbors.

    Attributes:
        d (dict): the adjacency dictionary of the graph.
    """
    def __init__(self, adjacency={}):
        """Store the adjacency dictionary as a class attribute"""
        self.d = dict(adjacency)

    def __str__(self):
        """String representation: a view of the adjacency dictionary."""
        return str(self.d)
```

Add the following methods to this class.

1. `add_node()`: Add a node (with no initial edges) if it is not already present. (Hint: use `set()` to create an empty set.)
2. `add_edge()`: Add an edge between two nodes. Add the nodes to the graph if they are not already present.
3. `remove_node()`: Remove a node, including all edges adjacent to it. This method should raise a `KeyError` if the node is not in the graph.
4. `remove_edge()`: Remove the edge between two nodes. This method should raise a `KeyError` if either node is not in the graph, or if there is no edge between the nodes.

Breadth-first Search

Many common problems that arise in graph theory require finding the shortest path between two nodes in a graph. For some highly structured graphs, such as binary search trees, this is a fairly straightforward problem (in the case of a tree, the shortest path is also the only path). Finding a path between nodes in a graph of arbitrary structure, however, requires a careful and methodical approach. The two most common graph search algorithms are *depth-first search* (DFS) and *breadth-first search* (BFS). The breadth-first strategy is almost always better at finding shortest paths than the depth-first strategy,[1] though a DFS can be useful for path problems in certain graphs.

To traverse a graph with a BFS, choose a node to start at, called the *source* node. First, visit each of the source node's neighbors. Next, visit each of the source node's neighbors' neighbors. Then visit each of their neighbors, continuing the process until all nodes have been visited. This strategy explores all of the nodes closest to the source node before incrementally moving "deeper" (further from the source node) into the tree.

The implementation of a BFS requires the following data structures to keep track of which nodes have already been visited and the order in which to visit nodes in future steps.

- A list V: The nodes that have been **visited**, in visitation order.
- A **queue** Q: The nodes to be visited, in the order that they were discovered. Recall that a *queue* is a limited-access list where data is inserted to one end, but removed from the other (first-in, first-out).
- A set M: The nodes that have been visited, or that are **marked** to be visited. This is the union of the nodes in V and Q.

To begin the search, add the source node to Q and M. Then, until Q is empty, repeat the following:

1. Pop a node off of Q; call it the *current* node.
2. "Visit" the current node by appending it to V.
3. Add the neighbors of the current node that are not in M to Q and M.

The "that are not in M" clause of step 3 prevents nodes from being added to Q more than once. Note that step 3 could be replaced with "Add the neighbors of the current node that are not in $V \cup Q$ to Q." However, lookup in M (a set) is much faster than lookup in V and Q (arrays or linked lists), so including M greatly speeds up the algorithm.

[1] See `https://xkcd.com/761/`.

Note

The first-in, first-out (FIFO) structure of Q enforces the "breadth-first" nature of the BFS: nodes that are marked first are visited first. Using a a last-in, first-out (LIFO) stack for Q changes the search to a DFS: the next node to visit is the one that was marked last.

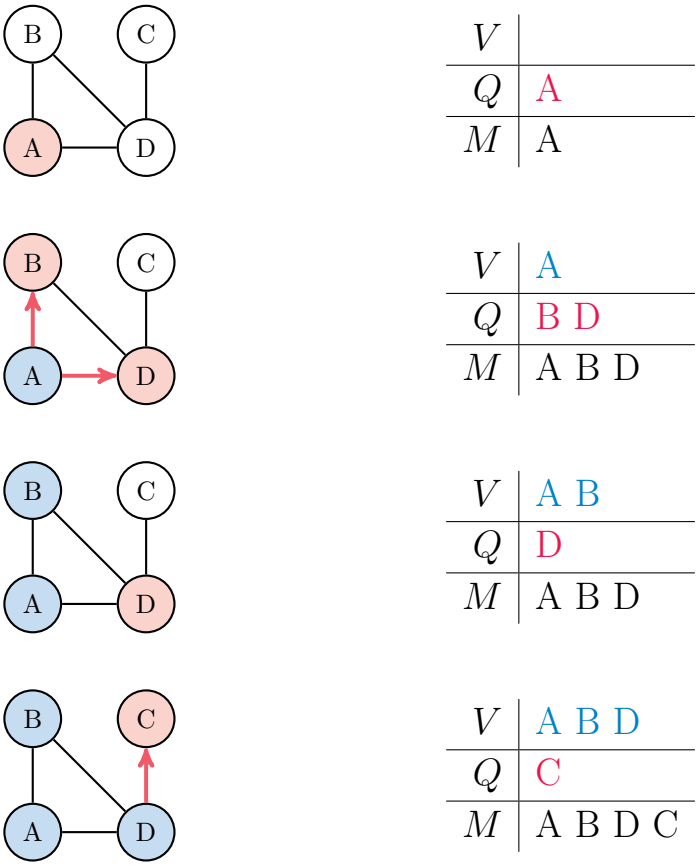


Figure 1.2: To start a BFS from node A to node C, put A in the visit queue Q and mark it by adding it to the set M. Pop A off the queue and "visit" it by adding A to the visited list V and the neighboring nodes B and D to Q. Then visit B, but do not add anything to Q because all of the neighbors of B are already marked. Finally, visit D, at which point the target node C is located because it is adjacent to D.

Problem 2. Write a method for the `Graph` class that accepts a source node. Traverse the graph with a breadth-first search until all nodes have been visited. Return the list of nodes in the order that they were visited. If the source node is not in the graph, raise a `KeyError`.
(Hint: for Q, use a `deque` from the `collections` module, and make sure that nodes are added to one end but popped off of the other.)

Shortest Paths via BFS

Consider the problem of locating a path between two nodes with a BFS. The nodes that are directly connected to the source node are all visited before any other nodes; more generally, the nodes that are n nodes away from the source node are all visited before nodes that are $n+1$ or more nodes from the source point. Therefore, the search path taken to discover to the target with a BFS must be the shortest path from the source node to the target node.

Examine again the graph in Figures 1.1 and 1.2. The shortest path from A to C starts at A, goes to D, and ends at C. During a BFS originating at A, D is placed on the visit queue because it is one of A's neighbors, and C is placed on the queue because it is one of D's neighbors. Given that A was the node that visited D, and that D was the node that visited C, the shortest path from A to C can be constructed by stepping backward along the search path.

To implement this idea, initialize a dictionary before starting the BFS. When a node is marked and added to the visit queue, add a key-value pair mapping the **visited** node to the **visiting** node (for example, B $\mapsto$ A means B was marked while visiting A). When the target node is found, step through the dictionary until arriving at the source node, recording each step.

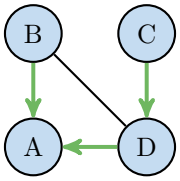


Figure 1.3: In the BFS from Figure 1.2, nodes B and D were marked while visiting node A, and node C was marked while visiting node D (this is same as reversing the red arrows in Figure 1.2). Thus the "visit path" from C to A is C $\to$ D $\to$ A, so the shortest path from A to C is [A, D, C].

Problem 3. Add a method to the `Graph` class that accepts source and target nodes. Begin a BFS at the source node and proceed until the target is found. Return a list containing the node values in the shortest path from the source to the target (including the endpoints). If either of the input nodes are not in the graph, raise a `KeyError`.

Shortest Paths via NetworkX

NetworkX is a Python package for creating, manipulating, and exploring graphs. Its `Graph` object represents a graph with an adjacency dictionary, similar to the class from Problems 1–3, and has many methods for interpreting information about the graph and its structure. As before, the nodes must be hashable (a number, string, or another immutable object).

Method	Description
`add_node()`	Add a single node to the graph.
`add_nodes_from()`	Add a list of nodes to the graph.
`add_edge()`	Add an edge between two nodes.
`add_edges_from()`	Add a list of edges to the graph.

Table 1.3: Methods of the `nx.Graph` class for adding nodes and edges.

```
>>> import networkx as nx

# Initialize a NetworkX graph from an adjacency dictionary.
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

>>> print(G.nodes())            # Print the nodes.
['A', 'B', 'C', 'D']
>>> print(G.edges())            # Print the edges as tuples.
[('A', 'D'), ('A', 'B'), ('B', 'D'), ('C', 'D')]

>>> G.add_node('E')             # Add a new node.
>>> G.add_edge('A', 'F')        # Add an edge, which also adds a new node 'F'.
>>> G.add_edges_from([('A', 'C'), ('F', 'G')])  # Add several edges at once.

>>> set(G['A'])                 # Get the set of nodes neighboring node 'A'.
{'B', 'C', 'D', 'F'}
```

The Kevin Bacon Problem

The vintage parlor game *Six Degrees of Kevin Bacon* is played by naming an actor, then trying to find the shortest chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Pulp Fiction (1994)* with Frank Whaley, who was in *JFK (1991)* with Kevin Bacon. In other words, the goal of the game is to solve a shortest path problem on a graph that connects actors to the movies that they have been in.

Problem 4. The file `movie_data.txt` contains IMDb data for about 137,000 movies. Each line of the file represents one movie: the title is listed first, then the cast members, with entries separated by a `/` character. For example, the line for *The Dark Knight (2008)* starts with

```
The Dark Knight (2008)/Christian Bale/Heath Ledger/Aaron Eckhart/...
```

Any `/` characters in movie titles have been replaced with the vertical pipe character `|` (for example, `Frost|Nixon (2008)`).

Write a class whose constructor accepts the name of a file to read. Initialize a set for movie titles, a set for actor names, and an empty NetworkX `Graph`, and store them as attributes. Read the file line by line, adding the title to the set of movies and the cast members to the set of actors. Add an edge to the graph between the movie and each cast member.
(Hint: Use the `split()` method for strings to parse each line.)

It should take no more than 20 seconds to construct the entire graph. Check that there are 137,018 movies and 930,717 actors. Compare parts of your graph to Figure 1.4.

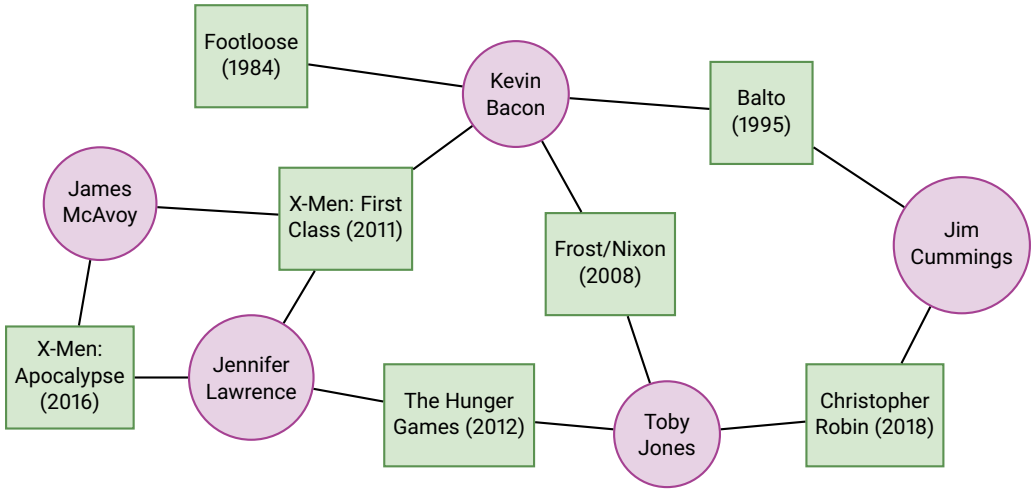


Figure 1.4: A subset of the graph in `movie_data.txt`. Each of these actors have a Bacon number of 1 because they have all been in a movie with Kevin Bacon. Every actor in *The Hunger Games* has a Bacon number of at most 2 because of the paths through Jennifer Lawrence or Toby Jones.

Note

The movie/actor graph of Problem 4 and Figure 1.4 has an interesting property: actors are only directly connected to movies, and movies are only directly connected to actors. This kind of graph is called *bipartite* because there are two types of nodes, and no node has an edge connecting it to another node of its type.

Achtung!

NetworkX `Graph` objects can be visualized with `nx.draw()` (followed by `plt.show()`). However, this visualization tool is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research. Because of the size of the dataset, **do not** attempt to visualize the graph in Problem 4 with `nx.draw()`.

The Six Degrees of Kevin Bacon game poses an interesting question: can any actor be linked to Kevin Bacon, and if so, in how many steps? The game hypothesizes, "Yes, within 6 steps" (hence the title). More precisely, let the *Bacon number* of an actor be the number of steps from that actor to Kevin Bacon, only counting actors. For example, since Samuel L. Jackson was in a film with Frank Whaley, who was in a film with Kevin Bacon, Samuel L. Jackson has a Bacon number of 2. Actors who have been in a movie with Kevin Bacon have a Bacon number of 1, and actors with no path to Kevin Bacon have a Bacon number of ∞. The game asserts that the largest Bacon number is 6.

NetworkX is equipped with a variety of graph analysis tools, including a few for computing paths between nodes (and, therefore, Bacon numbers). To compute a shortest path between nodes u and v, `nx.shortest_path()` starts one BFS from u and another from v, switching off between the two searches until they both discover a common node. This approach is called a *bidirectional BFS* and is typically faster than a regular, one-sided BFS.

Function	Description
`has_path()`	Return `True` if there is a path between two specified nodes.
`shortest_path()`	Return **one** shortest path between nodes.
`shortest_path_length()`	Return the length of the shortest path between nodes.
`all_shortest_paths()`	Yield **all** shortest paths between nodes.

Table 1.4: NetworkX functions for path problems. Each accepts a `Graph`, then a pair of nodes.

```
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

# Compute the shortest path between 'A' and 'D'.
>>> nx.has_path(G, 'A', 'C')
True
>>> nx.shortest_path(G, 'A', 'C')
['A', 'D', 'C']
>>> nx.shortest_path_length(G, 'A', 'C')
2

# Compute all possible shortest paths between two nodes.
>>> G.add_edge('B', 'C')
>>> list(nx.all_shortest_paths(G, 'A', 'C'))
[['A', 'D', 'C'], ['A', 'B', 'C']]

# When the second node is omitted from these functions, the shortest paths
# from the given node to EVERY node are computed and returned as a dictionary.
>>> nx.shortest_path(G, 'A')
{'A': ['A'], 'D': ['A', 'D'], 'B': ['A', 'B'], 'C': ['A', 'D', 'C']}
>>> nx.shortest_path_length(G, 'A')     # Path lengths are defined by the
{'A': 0, 'D': 1, 'B': 1, 'C': 2}        #  number of edges, not nodes.
```

Problem 5. Write a method for your class from Problem 4 that accepts two actors' names. Use NetworkX to compute the shortest path between the actors and the degrees of separation between the two actors (if one of the actors is `"Kevin Bacon"`, this is the Bacon number of the other actor). Note that this number is different than the number of entries in the actual shortest path list, since the movies are just intermediate steps between actors and are not counted when calculating the degress of separation.

The idea of a Bacon number provides a few ways to analyze the connectivity of the Hollywood network. For example, the distribution of all Bacon numbers describes how close Kevin Bacon is to actually knowing all of the actors in Hollywood Someone with a lower average number—for instance, the average *Jackson number*, for Samuel L. Jackson—is, on average, "more connected with Hollywood" than Kevin Bacon. The actor with the lowest average number is sometimes called *the center of the Hollywood universe.*

Problem 6. Write a method for your class from Problem 4 that accepts one actor's name. Calculate the shortest path lengths of every actor in the collection to the specified actor (not including movies). Use `plt.hist()` to plot the distribution of path lengths and return the average path length.
(Hint: Use a NetworkX function to compute all path lengths simultaneously; this is significantly faster than calling your method from Problem 5 repeatedly. Also, use the keyword argument `bins=[i-.5 for i in range(8)]` in `plt.hist()` to get the histogram bins to correspond to integers nicely.)

As an aside, the prolific Paul Erdős is the Kevin Bacon equivalent in the mathematical community. Someone with an *Erdős number* of 2 co-authored a paper with someone who co-authored a paper with Paul Erdős. Having an Erdős number of 1 or 2 is considered quite an achievement (see `https://xkcd.com/599/`).

Additional Material

Other Hash-based Structures

The standard library has a few specialized alternatives to regular sets and dictionaries.

- `frozenset`: an immutable version of the usual set class. Frozen sets cannot be altered after creation and therefore lack methods like `add()`, `pop()`, and `remove()`, but they can be placed in other sets and used as dictionary keys.
- `collections.defaultdict`: a dictionary with default values. For instance, `defaultdict(set)` creates a dictionary that automatically uses an empty set as the value whenever a non-present key is used for indexing. See `https://docs.python.org/3/library/collections.html` for examples.
- `collections.OrderedDict`: a dictionary that remembers insertion order. For example, the `popitem()` method returns the most recently added key-value pair.

Depth-first Search

A *depth-first search* (DFS) takes the opposite approach of a BFS. Instead of checking all neighbors of a single node before moving on, it checks the first neighbor, then their first neighbor, then their first neighbor, and so on until reaching a leaf node. The algorithm then backtracks to the previous node and checks its second neighbor. While a DFS is rarely useful for finding shortest paths, it is a common strategy for solving recursively structured problems, such as mazes or Sudoku puzzles.

Consider adding a keyword argument to your method from Problem 2 that specifies whether to use a BFS (the default) or a DFS. To change from a BFS to a DFS, change the visit queue Q to a stack. You may be able to implement the change in a single line of code.

The Center of the Hollywood Universe

Computing the center of the universe in a graph amounts to solving Problem 6 for every node in the graph. This is computationally expensive, but since each average number is independent of the others, the problem is a good candidate for *parallel programming*, which divides the computational workload between multiple processes. Even with parallelism, however, computing the center of the Hollywood universe may require significant computational time and resources.

Shortest Paths on Weighted Graphs

The graphs presented in this lab are *unweighted*, meaning all edges have the same importance. A *weighted graph* assigns a weight to each edge, which can usually be thought of as the distance between the two adjacent nodes. The shortest path problem becomes much more complicated on weighted graphs, and requires additions to the plain BFS. The standard approach is *Dijkstra's algorithm*, which is implemented as `nx.dijkstra_path()`. Another approach, the *Bellman-Ford algorithm*, is implemented as `nx.bellman_ford_path()`.