15.4 Connectivity and Recursive Graph Traversal

Now that we have covered the basics of a Python representation of graphs, we are ready to perform some interesting graph computations! First, recall this graph method from Section 15.3:

```
class Graph:
    def adjacent(self, item1: Any, item2: Any) -> bool:
        """Return whether item1 and item2 are adjacent vertices in this
         graph.

        Return False if item1 or item2 do not appear as vertices in this
         graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return any(v2.item == item2 for v2 in v1.neighbours)
        else:
            # We didn't find an existing vertex for both items.
            return False
```

Our goal in this section will be to implement a generalization of this method that goes from checking whether two vertices are adjacent to whether two vertices are connected:[1]

[1] Recall that two vertices are connected when there exists a path between them.

```
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """Return whether item1 and item2 are connected vertices in this
         graph.

        Return False if item1 or item2 do not appear as vertices in this
         graph.

        >>> g = Graph()
        >>> g.add_vertex(1)
        >>> g.add_vertex(2)
        >>> g.add_vertex(3)
        >>> g.add_vertex(4)
        >>> g.add_edge(1, 2)
        >>> g.add_edge(2, 3)
        >>> g.connected(1, 3)
        True
        >>> g.connected(1, 4)
        False
        """
```

We can start implementing this method in the same fashion as `Graph.adjacent`, by finding the vertex corresponding to `item1`:

```python
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            ...
        else:
            return False
```

The problem, of course, is in the final else branch. We can't simply check whether `item2` is an immediate neighbour of `item1`, since there are many other possible cases for two vertices being connected: `item2` could equal `item1` itself, or it could be a neighbour, or the neighbour of a neighbour, or the neighbour of a neighbour of a neighbour, etc.

So you might be tempted to start writing code that tries to cover all these cases:

```python
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            if v1.item == item2:
                return True
            elif any(u.item == item2 for u in v1.neighbours):
                return True
            elif any(any(u1.item == item2 for u1 in u.neighbours)
                     for u in v1.neighbours):
                return True
            ...
        else:
            return False
```

You can see why this gets unmanageable very quickly, and fact cannot possibly cover all possibilities! Because we need to handle an arbitrary number of "neighbour" links, we can't write a fixed amount of code that enumerates the possibilities; instead, we either need to use a loop or recursion. We'll develop a recursive approach in this section, although it is possible to translate this approach into one that uses loops as well.

Recursively determining connectivity

Here is our "complete" implementation of `Graph.connected`. As we did in earlier chapters, we're going to move the trickiest part (filling in that else branch) into a helper method.

```
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2, set())  # Pass in an empty
         "visited" set
        else:
            return False
```

Note that our helper method, `check_connected` is called with `vertex` to the left of the dot—that means that it's a method of the `_Vertex` class, not `Graph`. Here is its initial specification:

```
class _Vertex:
    def check_connected(self, target_item: Any) -> bool:
        """Return whether this vertex is connected to a vertex corresponding
         to the target_item.
        """
```

Now let's recall our definition of connectedness: two vertices are connected when there exists a path between them. The challenge with this definition is that it only requires that a path exists, but we don't know how long it is. But there's another, equivalent way of defining connectedness *recursively*. Given two vertices v_1 and v_2, they are connected when:

- $v_1 = v_2$, or
- there exists a neighbour u of v_1 such that u and v_2 are connected.

This definition maps onto the recursive definition structure we've seen so far, with the first point being the base case and the second being the recursive part. However, this definition is different from all the previous recursive definitions in one important respect: it isn't exactly *structural*, since it doesn't break down the data type—a graph—into a smaller instance with the same structure. We'll come back to this point further down below.

But first, well, let's take this recursive definition and run with it! Even though we're still new to graphs, everything we've learned about translating recursive definitions into Python still applies, and our code actually turns out to be surprisingly simple:[2]

[2] This version uses a loop, but you could certainly write a version with the built-in `any` function as well.

```python
class _Vertex:
    def check_connected(self, target_item: Any) -> bool:
        """Return whether this vertex is connected to a vertex corresponding
         to the target_item.
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            for u in self.neighbours:
                if u.check_connected(target_item):
                    return True

            return False
```

Pretty simple! But unfortunately, our implementation has a flaw that leads to a new kind of error:

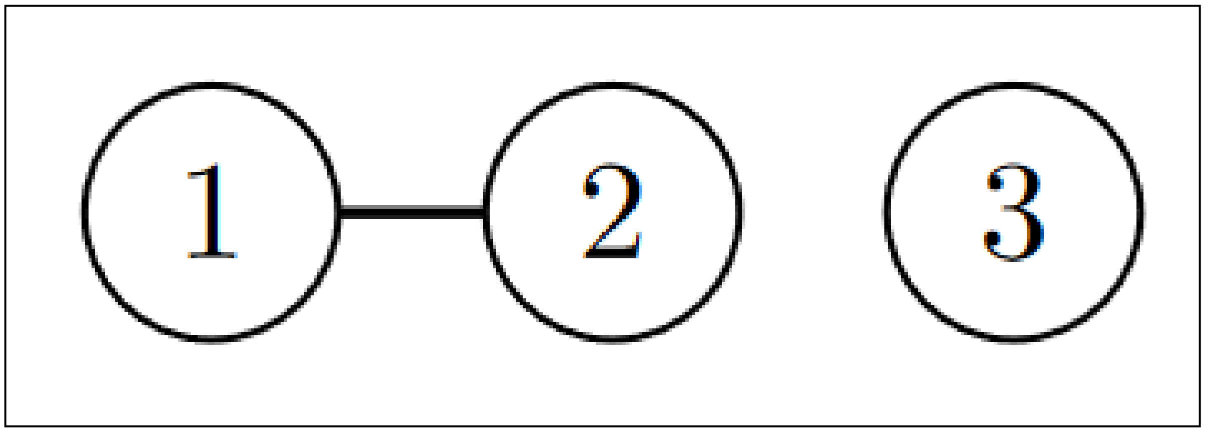
```
>>> g = Graph()
>>> g.add_vertex(1)
>>> g.add_vertex(2)
>>> g.add_edge(1, 2)
>>> g.connected(1, 3)  # Should return False!
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "path/to/graph.py", line 163, in connected
    return vertex.check_connected(item2)
  File "path/to/graph.py", line 41, in check_connected
    if u.check_connected(target_item):
  File "path/to/graph.py", line 41, in check_connected
    if u.check_connected(target_item):
  File "path/to/graph.py", line 41, in check_connected
    if u.check_connected(target_item):
  [Previous line repeated 985 more times]
  File "path/to/graph.py", line 36, in check_connected
    if self.item == target_item:
RecursionError: maximum recursion depth exceeded in comparison
```

What's going on?

`RecursionError` *and the danger of infinite recursion*

Our implementation for `_Vertex.check_connected` is simple, and exactly follows the recursive definition of connectedness we gave above. So what we're running into here is not an error in our logic, but a limitation of how we've expressed that logic in our code. To understand what's gone wrong, let's do a bit of manual tracing of our example above.

First, we have a graph `g` with three vertices, containing the items `1`, `2`, and `3`. In our initial call to `g.connected(1, 3)`, we first find the vertex corresponding to `1`, which we'll name v_1. Then our code calls `_Vertex.check_connected` on v_1 and `3`. This is where the problem begins.



1. For v_1, `self.item == 1`, which is not `3`. We enter the else branch, where the loop (`for u in self.neighbours`) executes. The only neighbour is the vertex containing `2`, which we'll name v_2. We make a recursive call to `_Vertex.check_connected` on v_2 and `3`.
2. For v_2, `self.item == 2`, which is not `3`. We enter the else branch, where the loop (`for u in self.neighbours`) executes. The only neighbour is v_1. We make a recursive call to `_Vertex.check_connected` on v_1 and `3`.
3. For v_1, `self.item == 1`, which is not `3`. We enter the else branch…

Hey, wait a second! We're back to the first step, repeating a call to `_Vertex.check_connected` with arguments v_1 and `3`. That's not good, because we know what happens in this case: we recurse on v_2.

This is an instance of **infinite recursion**, meaning we've expressed a recursive computation that does not stop by reaching a base case. In our example, the Python interpreter alternates between calling `_Vertex.check_connected` with v_1 and `3` and with v_2 and `3`, and never stops. From a technical point of view, each of these function calls adds a new *stack frame* onto the function call stack, which uses up more and more computer memory. To prevent your computer from running out of memory, the Python interpreter enforces a limit on the total number of stack frames that can be on the call stack at any one point in time—and when that limit is reached, the Python interpreter raises a `RecursionError`.

Fixing `_Vertex.check_connected`

Okay, so now that we understand what the problem is, how do we fix it? Intuitively, we need to prevent repeated recursive calls to the same node. Before changing our code, let's first revisit our recursive definition of connectedness.

Given two vertices v_1 and v_2, they are connected when:

- $v_1 = v_2$, or

- there exists a neighbour u of v_1 such that u and v_2 are connected.

Our problem is in the recursive step. When we say that "u and v_2 are connected", we're allowing for the possibility that these two vertices are connected by a path going through v_1. But that's not necessary: if v_1 and v_2 are connected, then we should be able to find a path between a neighbour u and v_2 that *doesn't* use v_1, and then add v_1 to the start of that path.

So we can modify this definition as follows. Given two vertices v_1 and v_2, they are connected when:

- $v_1 = v_2$, or
- there exists a neighbour u of v_1 such that u and v_2 are connected by a path that does not use v_1.

One way to visualize is that in the recursive step, we should be able to *remove v_1 from the graph* and still find a path between u and v_2. And this is how we can take our original recursive definition and impose some structure on it: we take the original graph and make it smaller by "removing" the vertex v_1, and then recursively checking for the connectivity of each neighbour u and v_2.

Adding a `visited` parameter

Now, let's take this idea and apply it to fix our code. It might be tempting to actually mutate our `Graph` object to remove the "v_1" vertex at each recursive call, but we don't actually want to mutate the original graph when we call `Graph.connected`. So instead, we'll use a common technique when traversing graphs: keep track of the items that have been already visited by our algorithm, so that we don't visit the same vertex more than once.

Here is our new `_Vertex.check_connected` specification:

```
class _Vertex:
    def check_connected(self, target_item: Any, visited: set[_Vertex]) ->
     bool:
        """Return whether this vertex is connected to a vertex corresponding
         to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
            - self not in visited
        """
```

And before getting to our updated implementation, let's update how we call this helper method in `Graph.connected`. When we make the initial call to `_Vertex.check_connected`, we haven't yet visited any vertices:

```
class Graph:
    def connected(self, item1: Any, item2: Any) -> bool:
        """..."""
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            return v1.check_connected(item2, set())  # Pass in an empty
         "visited" set
        else:
            return False
```

And finally, let's modify our implementation of `_Vertex.check_connected` to use this new parameter. We need to make two changes: first, we add `self` to `visited` before making any recursive calls (to indicate that the current `_Vertex` has been visited by our algorithm); second, when looping over `self.neighbours`, we only make recursive calls into nodes that have not yet been visited.

```
class _Vertex:
    def check_connected(self, target_item: Any, visited: set[_Vertex]) ->
         bool:
        """Return whether this vertex is connected to a vertex corresponding
         to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
            - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            new_visited = visited.union({self})  # Add self to the set of
         visited vertices
            for u in self.neighbours:
                if u not in new_visited:  # Only recurse on vertices that
         haven't been visited
                    if u.check_connected(target_item, new_visited):
                        return True

            return False
```

And with this version, we've eliminated our infinite recursion error:

```
>>> g = Graph()
>>> g.add_vertex(1)
>>> g.add_vertex(2)
>>> g.add_edge(1, 2)
>>> g.connected(1, 3)  # Should return False!
False
```

Bonus optimization

One last thing: you might have noticed that in our final implementation of `_Vertex.check_connected`, we created a new set of visited vertices (`new_visited`) rather than mutating `visited`, which would have been simpler and more efficient. Consider this alternate version:

```python
class _Vertex:
    def check_connected(self, target_item: Any, visited: set[_Vertex]) ->
         bool:
        """Return whether this vertex is connected to a vertex corresponding
         to the target_item,
        WITHOUT using any of the vertices in visited.

        Preconditions:
            - self not in visited
        """
        if self.item == target_item:
            # Our base case: the target_item is the current vertex
            return True
        else:
            visited.add(self)         # Add self to the set of visited
         vertices
            for u in self.neighbours:
                if u not in visited:  # Only recurse on vertices that haven't
         been visited
                    if u.check_connected(target_item, visited):
                        return True

            return False
```

This call uses our familiar `set.add` method, which is indeed constant time. However, there is an important difference with this version: now there is only one `visited` set object that gets shared and mutated across all recursive calls, including ones that return `False`.

So for example, suppose `self` has two neighbours `u0` and `u1`, and we first recursive on `u0`. If `u0` is not connected to the target vertex, then the `check_connected` call will return `False`, as expected. But `u0` would still be in `visited` after that recursive call ends, meaning the subsequent `check_connected` call with `u1` will never visit `u0`. This is somewhat surprising, but technically still correct: once we've established that `u0` is not connected to the target item, we *shouldn't* recurse on it ever again.

But even though this implementation is correct, and much more efficient than the previous one, it does come with an important warning we want you to remember for the future. Whenever you use recursion with a mutable argument, be very careful when choosing whether to mutate that argument or create a modified copy—if you choose to mutate the argument, know that all recursive calls will mutate it as well!