15.6 Computing Spanning Trees

In the last section, we learned about cycles and trees, two graph concepts that are deeply entwined with graph connectivity. We saw that given a graph G, if $G = (V, E)$ is a connected graph, then $|E| \geq |V| - 1$. We said that we could take any connected graph G and repeatedly remove edges that are part of a cycle until we end up with a *tree*—a connected graph with no cycles.

One computational question that arises naturally from this analysis is how can we design and implement an algorithm that finds such a tree. In this section, we'll tie together both theoretical and computational threads in this chapter to develop an algorithm that accomplishes this efficiently.

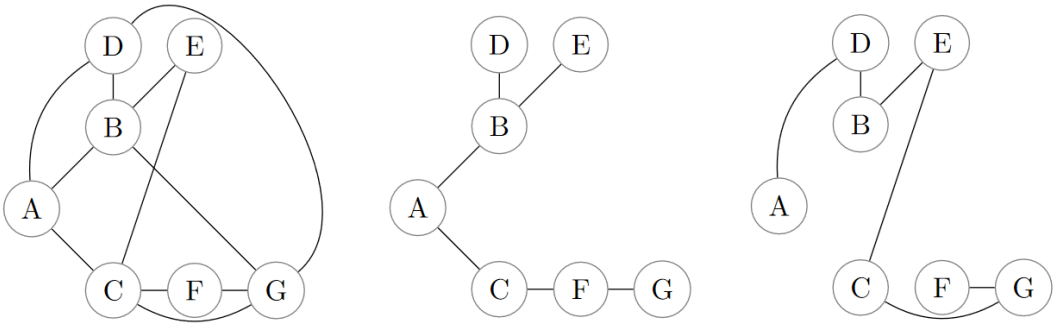
Spanning trees and the problem specification

First, a definition.

Definition. Let $G = (V, E)$ be a *connected* graph. Let $G' = (V, E')$ be another graph with the same vertex set as G, and where $E' \subseteq E$, i.e., its edges are a subset of the edges of G. We say that G' is a **spanning tree** of G when G' is a tree.

Every connected graph has at least one spanning tree, but can generally have more than one.[1] For example, the diagrams below show a graph G, and two possible spanning trees.

[1] In fact, it is possible to prove that a connected graph G has a *unique* spanning tree if and only if G is itself a tree.



Our goal is to implement a `Graph` method that computes a spanning tree for the graph. To keep things simple, we'll focus here on just returning a *subset of the edges*, although we could modify our approach to return a brand-new `Graph` object as well.

```
class Graph:
    def spanning_tree(self) -> list[set]:
        """Return a subset of the edges of this graph that form a spanning
         tree.

        The edges are returned as a list of sets, where each set contains the
         two
        ITEMS corresponding to an edge. Each returned edge is in this graph
        (i.e., this function doesn't create new edges!).

        Preconditions:
            - this graph is connected
        """
```

A brute force approach

The argument we gave at the end of the last section translates naturally into an algorithm for computing a spanning tree of a grpah. Given a connected graph, we can start with all of its edges and then try finding a cycle. If we find a cycle, remove one of its edges, and repeat until there are no more cycles. Here is some pseudocode for this approach:

```
class Graph:
    def spanning_tree(self) -> list[set]:
        edges_so_far = all edges in self
        while edges_so_far contains a cycle:
            remove an edge in the cycle from edges_so_far

        return edges_so_far
```

While this approach is correct, it's quite inefficient, as finding a cycle (the while loop condition) is relatively complex. It turns out that we can do better by taking a similar approach to the `Graph.connected` and `_Vertex.check_connected` methods from Section 15.4.

A first step: printing out connected vertices

To start, recall our `_Vertex.check_connected` method:

```
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
```

We used this method to check whether a given vertex was connected to some target vertex. This implementation uses a loop with an early return, stopping if one of the recursive calls finds the target vertex. However, suppose we drop the early return, or are searching for a vertex that `self` isn't connected to. In this case, we expect that this method will recurse on *every vertex that `self` is connected to.*

This means that we can take this same code structure and modify it to, for example, print out all of the vertices that `self` is connected to:

```
class _Vertex:
    def print_all_connected(self, visited: set[_Vertex]) -> None:
        """Print all items that this vertex is connected to, WITHOUT using
         any of the vertices
        in visited.

        Preconditions:
            - self not in visited
        """
        print(self.item)

        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't
         been visited
                u.print_all_connected(visited)
```

Here's an example of this method in action:

```
>>> g = Graph()
>>> for i in range(0, 5):
...     g.add_vertex(i)
>>> g.add_edge(0, 1)
>>> g.add_edge(0, 3)
>>> g.add_edge(1, 2)
>>> g.add_edge(1, 4)
>>> g.add_edge(2, 4)
```

```
>>> g._vertices[0].print_all_connected(set())
0
1
4
2
3
```

Okay, we can see all of the items printed out, so that's a start. But how do we turn this into a spanning tree? Let's make one modification to our code to make it easier to see what's going on. We'll use the same technique that we saw when printing out a `Tree` in Section 13.2 and add some indentation to our output.

```
class Graph:
    def print_all_connected_indented(self, visited: set[_Vertex], d: int) ->
         None:
        """Print all items that this vertex is connected to, WITHOUT using
         any of the vertices
        in visited.

        Print the items with indentation level d.

        Preconditions:
            - self not in visited
            - d >= 0
        """
        print('  ' * d + str(self.item))

        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't
         been visited
                u.print_all_connected_indented(visited, d + 1)
```

By increasing this depth parameter `d`, we can now see the recursive call structure that this method traces:

```
>>> # Using same graph g as above
>>> g._vertices[0].print_all_connected_indented(set(), 0)
0
  3
  1
    4
      2
```

This output looks an awful lot like the `Tree` outputs we saw in Section 13.2. So this is our key insight: the recursive call structure of `_Vertex.check_connected` forms a tree that spans

all of the vertices that the starting vertex is connected to.

A spanning tree algorithm

We can now use this insight to compute a spanning tree of a connected graph. Essentially, what we need to do is convert our recursive traversal algorithm from one that prints items to one that returns a list of edges.

There are two key changes we'll need to make:

1. Each recursive call will now return a list of edges, so we'll need an accumulator to keep track of them.
2. To "handle the root", we'll also need to add edges between `self` and each vertex where we make a recursive call.

```
class _Vertex:
    def spanning_tree(self, visited: set[_Vertex]) -> list[set]:
        """Return a list of edges that form a spanning tree of all vertices
         that are
        connected to this vertex WITHOUT using any of the vertices in
         visited.

        The edges are returned as a list of sets, where each set contains the
         two
        ITEMS corresponding to an edge.

        Preconditions:
            - self not in visited
        """
        edges_so_far = []

        visited.add(self)
        for u in self.neighbours:
            if u not in visited:  # Only recurse on vertices that haven't
         been visited
                edges_so_far.append({self.item, u.item})
                edges_so_far.extend(u.spanning_tree(visited))

        return edges_so_far
```

And we can call this method as follows:

```
>>> # Using same graph g as above
>>> g._vertices[0].spanning_tree(set())
[{0, 3}, {0, 1}, {1, 4}, {2, 4}]
```

Finally, we can use this method as a helper to implement our original `Graph.spanning_tree` method. But how will we choose a starting vertex? *We can choose any starting vertex we want*, because as long as we assume our graph is connected, any vertex we pick will be connected to every other vertex.

```
class Graph:
    def spanning_tree(self) -> list[set]:
        """Return a subset of the edges of this graph that form a spanning
         tree.

        The edges are returned as a list of sets, where each set contains the
         two
        ITEMS corresponding to an edge. Each returned edge is in this graph
        (i.e., this function doesn't create new edges!).

        Preconditions:
            - this graph is connected
        """
        # Pick a vertex to start
        all_vertices = list(self._vertices.values())
        start_vertex = all_vertices[0]

        # Use our helper _Vertex method!
        return start_vertex.spanning_tree(set())
```

Looking ahead

One thing you might have noticed with our spanning tree algorithm is that different starting vertices result in different spanning trees, because of the different recursive call structures that occur. This is perfectly normal—our function specification doesn't specify *which* spanning tree gets returned, only that one is returned.

In this course, we don't really care about being able to predict what spanning tree gets returned, as this requires tracing into each recursive call.[2] However, in future courses you'll learn more about different kinds of spanning trees and algorithms for computing them, and be able to better predict exactly which spanning tree a given algorithm will return, corresponding to the order in which the vertices are visited.

[2] As we've said many times in earlier chapters, this type of tracing is time-consuming and error-prone!