CSC111 Lecture 14: Representing Graphs in Python

Hisbaan Noorani

March 3, 2021

Contents

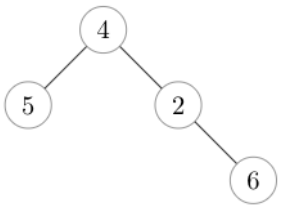
For your reference, here are the `_Vertex` and `Graph` classes we introduced in lecture.

```python
from __future__ import annotations
from typing import Any


class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.
    """
    item: Any
    neighbours: set[_Vertex]

    def __init__(self, item: Any, neighbours: set[_Vertex]) -> None:
        """Initialize a new vertex with the given item and neighbours."""
        self.item = item
        self.neighbours = neighbours


class Graph:
    """A graph.
    """
    # Private Instance Attributes:
    #     - _vertices:
    #         A collection of the vertices contained in this graph.
    #         Maps item to _Vertex object.
    _vertices: dict[Any, _Vertex]

```

```
    def __init__(self) -> None:
        """Initialize an empty graph (no vertices or edges)."""
        self._vertices = {}

    def add_vertex(self, item: Any) -> None:
        """Add a vertex with the given item to this graph.

        The new vertex is not adjacent to any other vertices.

        Preconditions:
            - item not in self._vertices
        """
        self._vertices[item] = _Vertex(item, set())

    def add_edge(self, item1: Any, item2: Any) -> None:
        """Add an edge between the two vertices with the given items in this graph.

        Raise a ValueError if item1 or item2 do not appear as vertices in this graph.

        Preconditions:
            - item1 != item2
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Add the new edge
            v1.neighbours.add(v2)
            v2.neighbours.add(v1)
        else:
            # We didn't find an existing vertex for both items.
            raise ValueError
```

1 Exercise 1: Reviewing the `Graph` implementation

1. Consider the following graph:



Write the Python code that we could use to represent this graph. We've started by creat-

ing an empty `Graph` for you, which you should mutate with calls to `Graph.add_vertex` and `Graph.add_edge`:

```
>>> graph = Graph()
>>> graph.add_vertex(2)
>>> graph.add_vertex(4)
>>> graph.add_vertex(5)
>>> graph.add_vertex(6)
>>> graph.add_edge(2, 4)
>>> graph.add_edge(2, 6)
>>> graph.add_edge(4, 5)
```

2. Complete the following function, which creates and returns a graph of `n` vertices where every vertex is adjacent to every other vertex.

```
def complete_graph(n: int) -> Graph:
    """Return a graph of n vertices where all pairs of vertices are adjacent.

    The vertex items are the numbers 0 through n - 1, inclusive.

    Preconditions:
        - n >= 0
    """
    graph_so_far = Graph()

    for i in range(0, n):
        graph_so_far.add_vertex(i)

        for j in range(0, i):
            graph_so_far.add_edge(i, j)

    return graph_so_far
```

3. Finally, add two representation invariants to the `_Vertex` class to represent the following restrictions on edges in a graph:

   - a vertex cannot be a neighbour of itself
   - edges are symmetric: for any vertex v, all of its neighbours have v in their `neighbours` set

```
class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.

```

```
    Representation Invariants:
        - self not in self.neighbours
        - all(self in n.neighbours for n in self.neighbours)
    """
    item: Any
    neighbours: set[_Vertex]
```

2 Exercise 2: Writing `Graph` functions

In this exercise, you'll get some practice writing some methods to operate on our new `_Vertex` and `Graph` data types.

1. Implement the `Graph` method below.

```
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]
            v2 = self._vertices[item2]

            # Return whether the two vertices have an edge
            # This will handle them both being in the graph
            # but not being neighbours
            return v1 in v2.neighbours and v2 in v1.neighbours

        # Return False when they are not in the graph
        return False
```

2. Implement the `Graph` method below.

   Hint: use the statement from Lecture 13 Exercises, Part 2 Q4

```
class Graph:
    def num_edges(self) -> int:
        """Return the number of edges in this graph."""
        total_degree = 0

        for item in self._vertices:
            total_degree += len(self._vertices[item].neighbours)

        # Bonus comprehension implementation!
        # total_degree = sum(len(self._vertices[item].neighbours)
        #                    for item in self._vertices)
```

```
12
13          return total_degree // 2
14
15          # You could even do this all in a oneliner:
16          return sum(len(self._vertices[item].neighbours)
17                     for item in self._vertices) // 2
```

3 Additional exercises

1. Write a `Graph` method that returns the maximum degree of a vertex in the graph (assuming the graph has at least one vertex).

2. Write a `Graph` method that returns a list of all edges (represented as `sets` of items) in the graph. Don't worry about order in the list.

3. (*harder*) A **triangle** in a graph is a set of three vertices that are all adjacent to each other. Write a `Graph` method that returns a list of all triangles in the graph.