Kevin Bacon problem

The starter code given to you has the following functions:

- `getMapAtoM`: reads the given file and creates a mapping from actors to their movies (which are sets)

- `createActorGraph`: creates an undirected graph where the vertices are actors; two actors are connected (have an edge between them) if they have a movie in common;

Note: this function uses as input the mapping created by `getMapAtoM`.

You are provided the first function and a skeleton of the second. Your task is to complete the second method (using set intersection). Next, create a function `KBNcompute` that takes this graph and an actor, and returns that actor's Kevin Bacon number:

```
mapAtoM = getMapAtoM()
print (mapAtoM['Rock, Chris']) # the set of Chris Rock's movies
G = createActorGraph(mapAtoM)
print ('Bacon, Kevin' in G.neighbors('Costner, Kevin')) # should print True
k = KBNcompute(G, 'Coestner, Kevin') # should return 1
```

Note: For any actor X, their Kevin Bacon (KB) number is the length of the path between that actor and 'Bacon, Kevin' (or infinity, if there is no path). For Kevin Bacon, the KB-number is 0.

Note: `KBNcompute(G, A)` should make use of the path-finding method from step 4 of part 2 of lab 12 (`findPath` method), so all it does is find the path between `A` and `'Bacon, Kevin'` and return its length!

In the `findPath` method of the earlier part of the lab, you were asked to use `dfs`, which doesn't always give you the shortest path. In order to find the Bacon number, you need to find the shortest path, which is given by `bfs`. Replace the call to `dfs` by a call to `bfs`. The pseudo code for bfs is as follows:

```
def bfs(G, origin):
    tree[origin] = None
    queue = [origin]
    while queue is not empty
        curnode = remove first in queue
        for x in neighbors (curnode):
            if x is not in tree:
                    tree[x] = curnode
                    add x to queue
    return tree

# tree is the mapping for each node to the previous one in the path to origin
```

Example for bfs:

```
G = SimpleGraph({'A','B','C','D'}, {('A','B'),('B','C'),('A','C'),('C','D')})

print(G.findPath('A', 'C'))
## findPath with bfs returns ['A', 'C'], the shortest path
## findPath with dfs returns ['A', 'B', 'C']
```

Now let's talk about something called a LABELLED graph. It has the additional attribute `_M`, which is a mapping from edges to labels. Labels can be numbers, or strings, or anything else. Add this map to your `SimpleGraph` class (in addition to `vertices` and `edges`), and change the `addEdge` method to take an optional third argument, which is a label (whose DEFAULT value is `None`). When label is not `None`, `addEdge` should add this label to the mapping:

```
    if label != None:
       self._M((u,v)) = label
```

Note that for an undirected graph, both `(u,v)` and `(v,u)` get mapped to the same label.

Next, change the `createActorGraph` function so for each edge, it stores the size of the intersection between the movie sets as the label. For example, if actor `A` and actor `B` have 5 movies in common, we will be calling `G.addEdge(A, B, 5)`. Note that since this graph is undirected graph, both `(A,B)` and `(B,A)` will be mapped to `5`.

Finally, add a graph method `getLabel`, which returns the label of a given edge:

```
mapAtoM = getMapAtoM()
print (mapAtoM['Rock, Chris']) # the set of Chris Rock's movies
G = createActorGraph(mapAtoM)
print ('Bacon, Kevin' in G.neighbors('Costner, Kevin')) # should print True
k = KBNcompute(G, 'Coestner, Kevin') # should return 1
x1 = G.getLabel('Bacon, Kevin','Costner, Kevin') # should return 24
x2 = G.getLabel('Costner, Kevin', 'Bacon, Kevin') # should also return 24
```