10. A Weekend to Remember

Weekends don't count unless you spend them doing something completely pointless. – Bill Watterson.

Programming constructs and algorithmic paradigms covered in this puzzle: Graph representation using dictionaries. Recursive depth-first graph traversal.

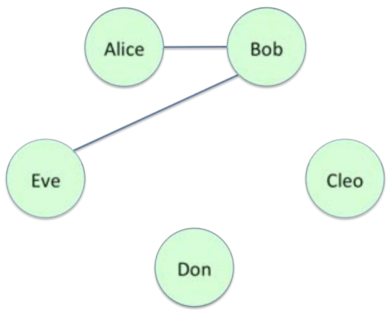
You have gotten into trouble with some of your, shall we say, difficult friends because they have realized that they are not being invited to your house parties. So you announce that you are going to have dinners on two consecutive nights, Friday and Saturday, and every one of your friends is going to get invited on exactly one of those two days. You are still worried about pairs of your friends who intensely dislike each other and want to invite them on different days.

To summarize:

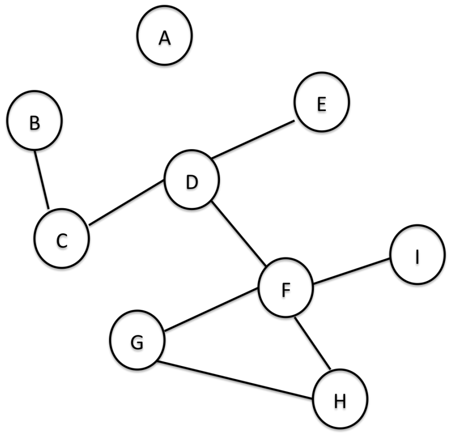
1. Each of your friends must attend exactly one of the two dinners.

2. If A dislikes B or B dislikes A, they cannot both be in the same dinner party.

Now, you are a bit worried because you don't know if you can pull this off. If you had a small social circle like the one below, it would be easy. Recall that in the graph below, the vertices are friends and an edge between a pair of vertices mean that the two corresponding friends dislike each other, and cannot be invited to the party on the same night.



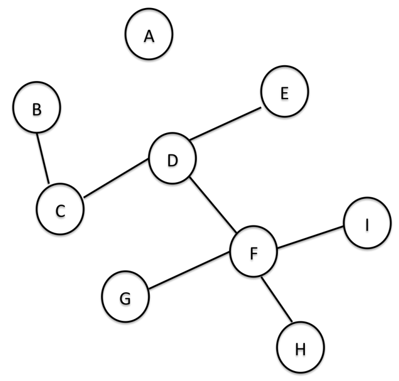
For Dinner 1 you could invite Bob, Don, and Cleo. And for Dinner 2 you could invite Alice and Eve. There are other possibilities as well. Unfortunately, that social circle was from many moons ago and you have moved on. Your social circle now looks like this:



Can you invite all of your friends A through I above following your self-imposed rules? Or are you going to have go to back on your word, and not invite someone?

You're hosed. If you look at the graph carefully, you will see that you have three friends F, G and H, each of whom can't be invited with other two. You will need three separate days to host each of them ☹

Okay, let's assume that you convince G and H to not cause a ruckus if they see each other. So your social graph now looks like this.



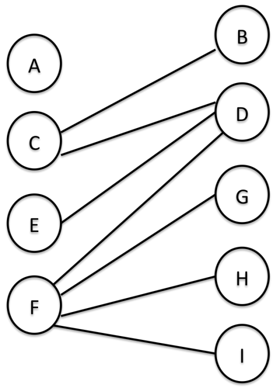
What about now?

Finding a Partition

With some work, you probably figured out that you could invite B, D, G, H and I to Dinner 1, and A, C, E and F to Dinner 2. Phew!

Your social circle is volatile and you might have to do this analysis week after week. You would like to quickly know if you can make an announcement any given week, such as "party on both nights, everyone gets to come to exactly one," with the confidence that it will be a boisterous but friendly weekend. Which means you want to write a program that can check if there is some way that you can "partition," i.e., split your friends across two nights, so if you look at the social circle graph, all the dislikes edges would be across the partition, and none within either side.

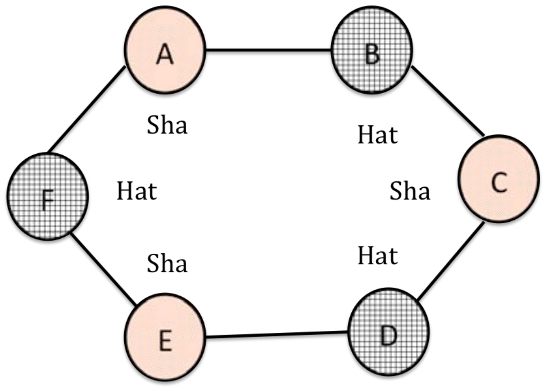
The partition we found looks like this pictorially.



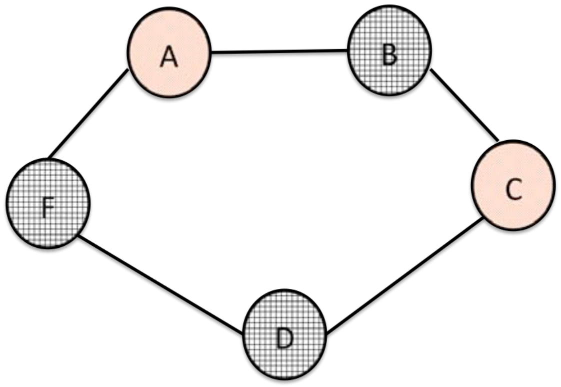
This is simply redrawing your social circle graph! It turns out the graph above has a name – it is a *bipartite graph*. A bipartite graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. We can also say that there is no edge that connects vertices of same set. In the above example U = {A, C, E, F} and V = {B, D, G, H, I}.

A graph is bipartite if the graph can be colored using two colors such that no two vertices that are adjacent, i.e., share an edge, have the same color. This is, of course, a restatement of our original problem, with colors substituted for dinner nights. We will refer sometimes refer to the coloring constraint as the *adjacency* constraint.

You might think that a bipartite graph can't have cycles. It is possible to color a graph with cycles comprising an *even* number of vertices using two colors, Shaded and Hatched. For example, see the following graph below.



U = {A, C, E} and V = {B, D, F}, and we can invite members of U on one day, and members in V on the second. However, it is not possible to color a graph with two colors if has a cycle involving an *odd* number of vertices. Consider F, G and H in our second example with 9 vertices where G and H have an edge between them. F, G and H are in a 3-cycle and hence that graph is not bipartite. The 5-cycle in the graph below renders it non-bipartite.



Changing the color of A to Hatched is not going to help. B and F need to be Shaded, and then C and D need to be Hatched, which violates an adjacency constraint.

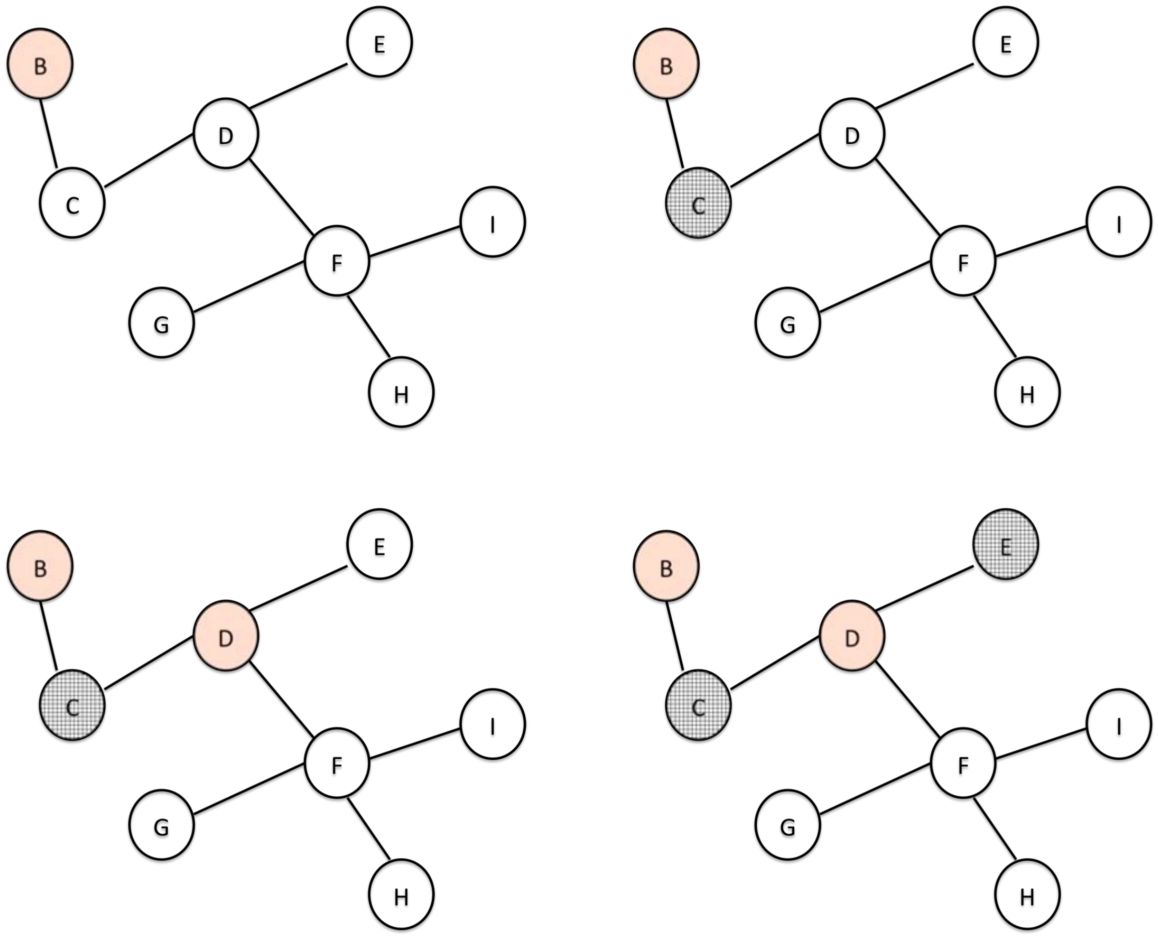
Checking if a Graph is Bipartite

We need an algorithm that given a graph successfully colors the vertices with two colors obeying the adjacency constraints if the graph is bipartite or tells us that the graph is not bipartite. One color will correspond to the set U, and the other to the set V. Here's a possible algorithm that uses a technique called *depth-first search*.

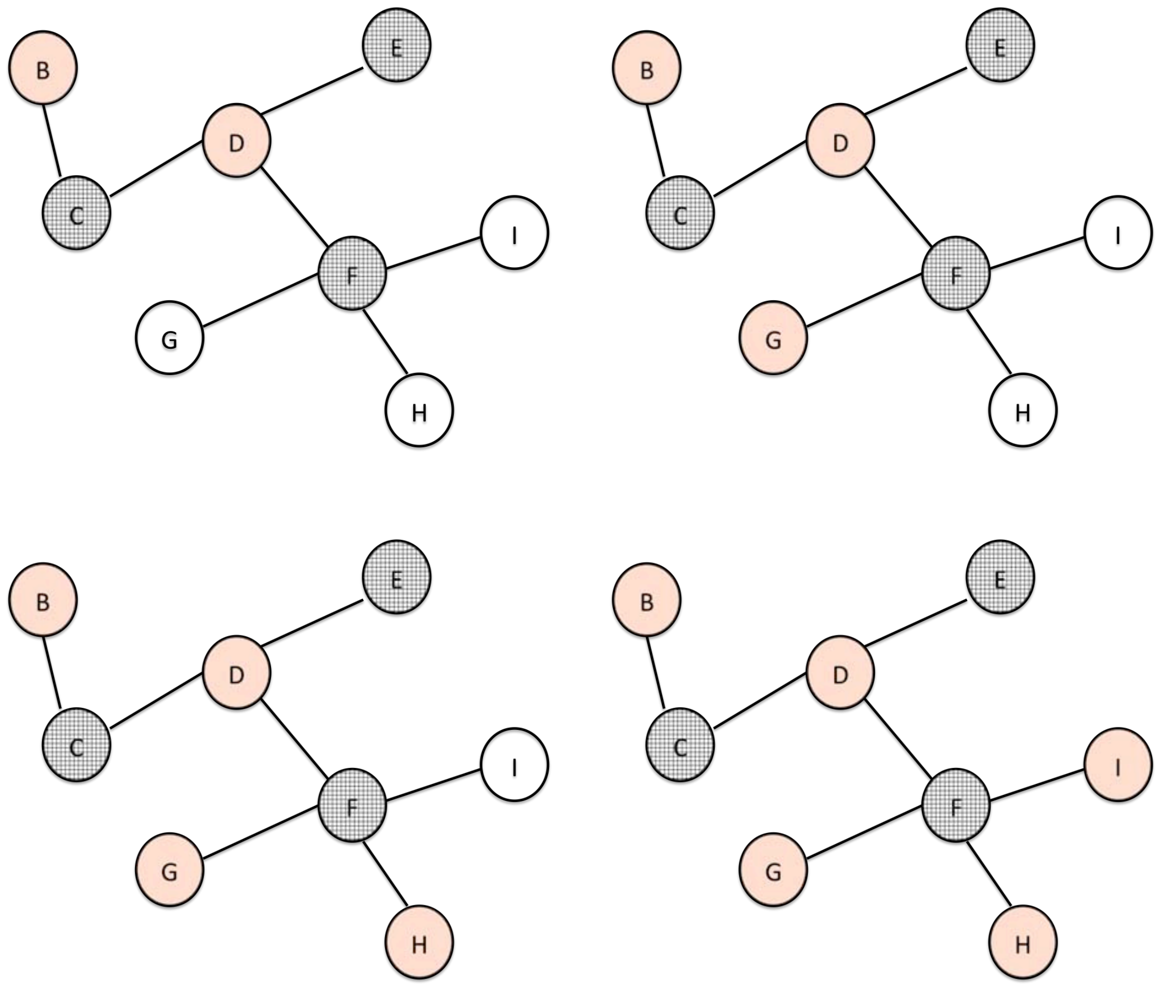
1. *color* = Shaded, vertex = starting vertex *w*.
2. If *w* has not yet been colored, color vertex *w* with *color*.
3. If *w* has already been colored with a different color from *color*, the graph is not bipartite, return **False**.
4. If *w* has already been colored with the correct color, return **True** and the (unmodified) coloring.

5. Flip *color*: Shaded to Hatched, or Hatched to Shaded.
6. For each neighbor *v* of *w*, recursively call the procedure with *v* and *color*, i.e., go to Step 2 with *w* = *v*. If any of the recursive calls return **False**, return **False**.
7. The graph is bipartite, return **True** and the coloring.

Let's run this algorithm on an example graph shown below beginning with coloring vertex B Shaded. C is the only vertex connected to B and is colored next. From C we go to D since B is already colored. Once we color D, since C is already colored, we have the choice of coloring E or F, and we color E first.



Since E has no neighbors other than D, we next move to F, which is a neighbor of D. We color the neighbors of F in the order G, H and I.



An observant reader may have noticed that our example above omitted vertex/friend A from the social circle example we presented earlier. This is because A is not "connected" to any of the other vertices – it does not have any neighbors. Of course, A could be colored either Shaded or Hatched.

We'll assume that the input graph is such that we can reach all vertices from a specified start vertex. One exercise at the end of the puzzle chapter will deal with the general case of possibly disconnected components in the input graph.

Graph Representation

Before we dive into coding the algorithm, we have to choose a data structure for our graph. The data structure should allow us to perform the operations we need for the algorithm, in particular, being able to take a vertex, find its neighbors, and then the neighbors' neighbors, etc. Here's a graph representation based on Python dictionaries that will serve our needs. It represents the graph that we just executed the algorithm on.

```
graph = {'B': ['C'],
         'C': ['B', 'D'],
         'D': ['C', 'E', 'F'],
         'E': ['D'],
         'F': ['D', 'G', 'H', 'I'],
         'G': ['F'],
         'H': ['F'],
         'I': ['F']}
```

The dictionary represents graph vertices and edges. We use strings to represent graph vertices: B in the graph picture is `'B'`, and so on. Each of the vertices is a key in the dictionary `graph`. Each line corresponds to a key: value pair, where the value is a list of edges from the vertex key. The edge is simply represented by the destination vertex of the edge. In our example, B has a single edge to C, and thus we have a single-item list for the value of the `'B'` key. On the other hand, vertex key `'F'` has a four-item list corresponding to its four edges.

We have undirected graphs in this puzzle, meaning that each edge is undirected. This means that if we can traverse an edge from vertex B to vertex C, we can traverse the edge in the opposite direction from vertex C to vertex B. For our dictionary representation, this means that if a vertex key X has a vertex Y in its value list, then vertex key Y will also have vertex X in its value list. Check the dictionary `graph` above for this symmetry condition.

It is important to note that we do not want to depend on any particular order of the keys in the dictionary. The code we write should discover that the following graph representation `graph2` corresponds to a bipartite graph and color it properly. If we start with the same vertex and color for input `graph` and `graph2`, we should get the exact same coloring.

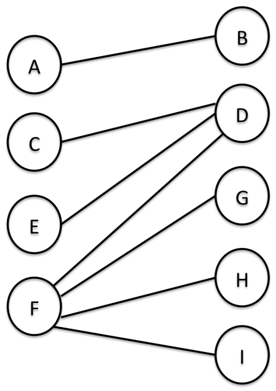
```
graph2 = {'F': ['D', 'G', 'H’, ‘I’],
          'B': ['C'],
          'D': ['C', 'E', 'F'],
          'E': ['D'],
          'H': ['F'],
          'C': [‘B’, 'D'],
          'G': ['F'],
          'I': ['F']}
```

Note that the execution of the algorithm shown in the 8 pictures corresponds to either dictionary `graph` or `graph2`. The order in which vertices are colored does depend on the ordering of the value lists. This is why, for example, after vertex D is colored, G is colored before H, which is colored before I.

The code for bipartite graph coloring in bipartite.py closely follows the pseudocode presented earlier.

Exercises

Exercise 1: The bipartite graph checker assumes that the graph is connected. Your social circle may contain disconnected components as in the example below – a variant of the earlier example.



Modify the code so it works on the above type of graph. The current code will only color two vertices A and B in the graph if the start vertex argument to `bipartiteGraphColor` is A or B, and leave C through I uncolored.

Create a parent procedure that invokes `bipartiteGraphColor` on the input graph with some starting vertex. Then, check that all vertices in the input graph are colored. If not, begin with an uncolored vertex and run `bipartiteGraphColor` starting with this vertex. Continue till all vertices in the input graph are colored. Once you have colored all of the vertices in each of the components, output dinner invitations provided all of the components are bipartite.

Exercise 2: Modify the procedure `bipartiteGraphColor` so it prints a cyclic path from the start vertex that cannot be colored using two colors, if such a path exists in the graph. Such a path will exist if the graph is not bipartite, and will not if the graph is bipartite. The cycle may not include the start vertex, but will be reachable from the start vertex in the case where the graph is not bipartite. Suppose you are given the graph below:

```
graphc = {'A': ['B', 'D' 'C'],
          'B': ['C', 'A', 'B'],
          'C': ['D', 'B', 'A'],
          'D': ['A', 'C', 'B']}
```

Your modified procedure should output:

```
Here is a cyclic path that cannot be colored ['A', 'B', 'C', 'D',
'B']
(False, {})
```

Exercise 3: The procedure `bipartiteGraphColor` embodies depth-first search. The code below makes recursive calls following a sequence of vertices.

```
    for vertex in graph[start]:
        val, coloring = bipartiteGraphColor(graph,\
                        vertex, coloring, newcolor)
```

Rather than coloring the graph, suppose we wish to find paths between pairs of vertices. Write a function `findPath` that finds and returns a path from a start vertex to an end vertex if such a path exists in the graph, and returns **`None`** if it does not. If we run `findPath` on the dictionary `graph`, with start vertex `'B'` and end vertex `'I'`, it should find the path `['B', 'C', 'D', 'F', 'I']`.