RCL-March 13, 2019
Jonathan Crawford – June 22, 2019

Homework 3: IsBipartite Algorithm 50 Points

A *k−coloring* of a graph $G = (V,E)$ is a mapping *color* : $V \rightarrow \{1,...,k\}$ such that for any edge $e = [v,w] \in E$ $color(v)$!= $color(w)$. G is said to be *bipartite* iff G has a 2-coloring.

The idea of a k-coloring is that the vertices can be "colored" using k distinct colors so that the vertices of any edge have different colors. A bipartite graph is one that can be colored with two colors.

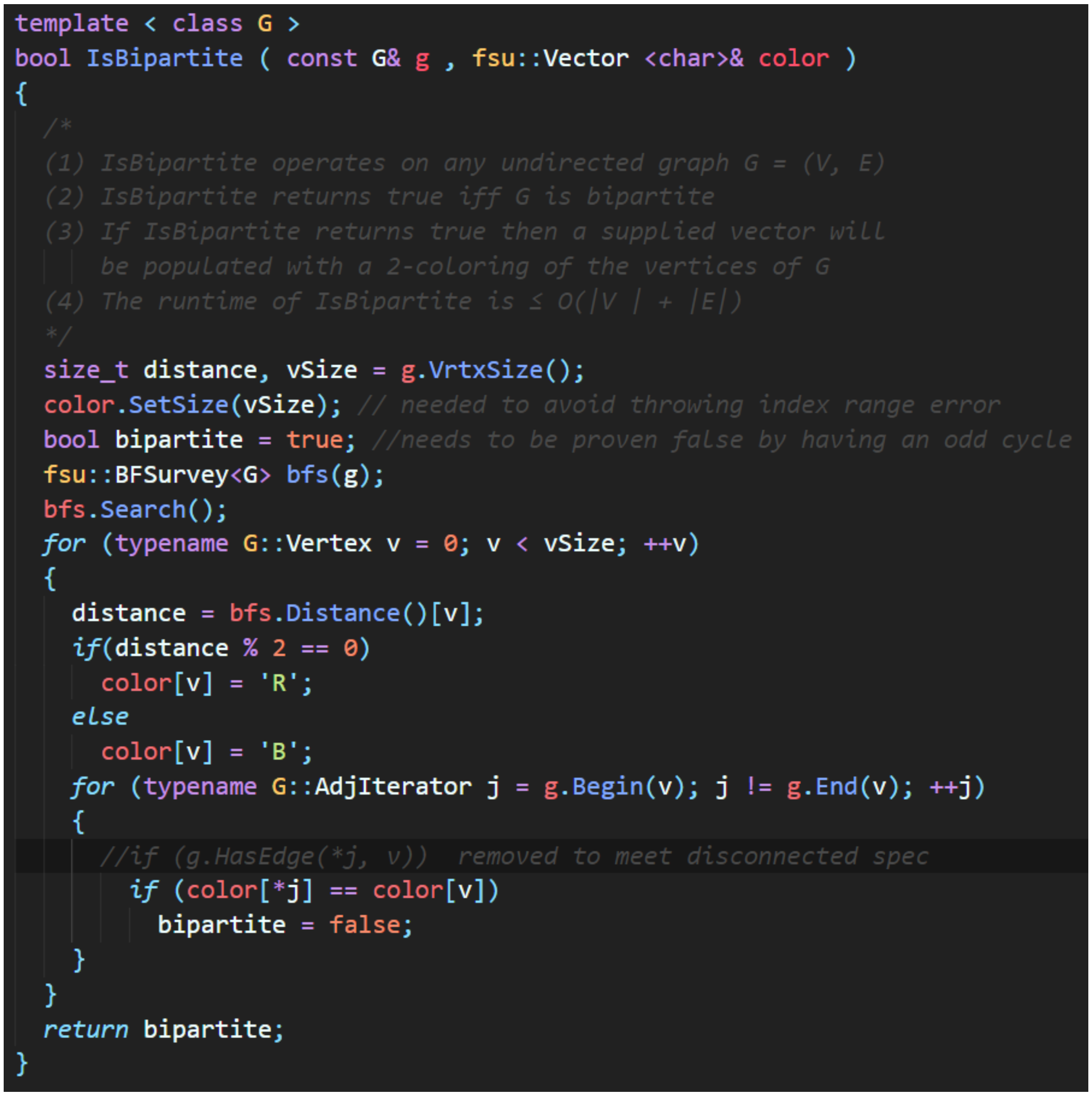
Part 1. Invent an algorithm named **IsBipartite** with these properties:

(1) IsBipartite operates on any undirected graph $G = (V,E)$
(2) IsBipartite returns true iff G is bipartite
(3) If IsBipartite returns true then a supplied vector will be populated with a 2-coloring of the vertices of G
(4) The runtime of IsBipartite is ≤ $O(|V| + |E|)$

We will use Theorem 4 from Rosen (2012, p. 657): *A simple graph is bipartite if and only if it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color.*

The algorithm entails using a passed or local vector color to assign alternating colors to the vertices so that they can be compared for adjacency. In formatting the algorithm like a proof, we will assume the graph is bipartite and then attempt to prove it false with an odd cycle. Rosen (2012) notes that the beginning color is arbitrary but that it much switch off, for that we will just use evens and odds using modulo as is common when alternating values in code. The page referenced on GeeksForGeeks uses a nested breadth-first search to search the vertices, but we will call that from the FSU library. The reason it is important to use a breadth first search is to provide an efficient way to iterate through the graph handling previously visited vertices and meet the runtime stipulation noted above as noted in RCL (2018). After coloring all the vertices, then loop through their adjacency list and check for any adjacent of the same color, this would violate the property of a bipartite graph by indicating an odd cycle and provide a contradiction setting the return Boolean to false. I was able to find most of the solution in the Rosen discrete math textbook and notes on bipartite graphs from MAD3105. After seeing Dr. Lacher's comment on the discussion board I removed the if case for HasEdge from the adjacency list loop taken from CheckSymmetry to account for graphs that are not connected.

Part 2. Code up the algorithm in C++ conformant with the stub below. Test the implementation on small graphs that can be hand verified and on some large graphs (such as the "Kevin Bacon" actor-movie abstract graph) and some very large graphs generated at random. Include some random maze graphs, and report any discoveries.



```
template < class G >
bool IsBipartite ( const G& g , fsu::Vector <char>& color )
{
  /*
  (1) IsBipartite operates on any undirected graph G = (V, E)
  (2) IsBipartite returns true iff G is bipartite
  (3) If IsBipartite returns true then a supplied vector will
      be populated with a 2-coloring of the vertices of G
  (4) The runtime of IsBipartite is ≤ O(|V | + |E|)
  */
  size_t distance, vSize = g.VrtxSize();
  color.SetSize(vSize); // needed to avoid throwing index range error
  bool bipartite = true; //needs to be proven false by having an odd cycle
  fsu::BFSurvey<G> bfs(g);
  bfs.Search();
  for (typename G::Vertex v = 0; v < vSize; ++v)
  {
    distance = bfs.Distance()[v];
    if(distance % 2 == 0)
      color[v] = 'R';
    else
      color[v] = 'B';
    for (typename G::AdjIterator j = g.Begin(v); j != g.End(v); ++j)
    {
      //if (g.HasEdge(*j, v))  removed to meet disconnected spec
        if (color[*j] == color[v])
          bipartite = false;
    }
  }
  return bipartite;
}
```

```
// unchanged
template < class G >
bool IsBipartite (const G& g)
{
  fsu::Vector<char> color (g.VrtxSize());
  return IsBipartite (g,color);
}
```

I used the 5 vertex pentagon as my small non-bipartite test for odd cycle, I am familiar with this graph C5 from my discrete math notes. I also made a C4 graph just a box as my small bipartite test and compared the outputs to agraph_i.x from the library. I had some problems with the vector returning a size error that I finally figured out with SetSize. I copied a lot of my loop logic from the CheckSymmetry() function and how to declare the search from Path_BFS(). I was unable to find the Kevin Bacon graph mentioned or make a nonbipartite maze I think because our code makes them symmetric. So I just decided to make some huge graphs with rangraph to test, and compare agraph.x with agraph_i.x to verify my findings on the larger graphs. I also made a couple mazes with 1 and larger components with ranmaze2t and then turned them into graphs with maze2graph to satisfy the random maze input specification. Below are some examples.

C4 graph: Welcome to graph analysis
```
 g.VrtxSize(): 4
 g.EdgeSize(): 4
 g bipartite?  YES:  Red = 2 , Black = 2
 ** nonsense
```
C5 graph: Welcome to graph analysis
```
 g.VrtxSize(): 5
 g.EdgeSize(): 5
 g bipartite?  NO
 ** nonsense
```
Giant Random Bipartite: Welcome to graph analysis
```
 g.VrtxSize(): 10000000
 g.EdgeSize(): 500000
 g bipartite?  YES:  Red = 9530928 , Black = 469072
 ** nonsense
```
Big Random Non-Bipartite: Welcome to graph analysis
```
 g.VrtxSize(): 615635
 g.EdgeSize(): 535450
 g bipartite?  NO
 ** nonsense
```
1 Component Maze: Welcome to graph analysis
```
 g.VrtxSize(): 10000
 g.EdgeSize(): 19998
 g bipartite?  YES:  Red = 5000 , Black = 5000
 ** nonsense
```
257 Component Maze: Welcome to graph analysis
```
 g.VrtxSize(): 10000
 g.EdgeSize(): 19486
 g bipartite?  YES:  Red = 5094 , Black = 4906
 ** nonsense
```

Part 3. Provide a proof that your algorithm is correct.

Bipartite Definition (Rosen 2012, p. 656) :
A simple graph G is called bipartite if its vertex set V can be partitioned into two disjoint sets V1 and V2 such that every edge in the graph connects a vertex in V1 and a vertex in V2 (so that no edge in G connects either two vertices in V1 or two vertices in V2). When this condition holds, we call the pair (V1, V2) a bipartition of the vertex set V of G.

Theorem 4 (Rosen 2012, p. 657)
A simple graph is bipartite if and only if it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color.

To prove a graph G is not bipartite, assume that it is. Then in respect to the definition of a bipartite graph there will be 2 distinct sets V1 and V2 within G such that no adjacency exists. We will walk each vertex of G and color them red or black to indicate these two distinct sets. Then by definition there exists no adjacency of the same color red or black within G. After coloring the vertex, walk the adjacency list of the vertex and inspect the color of each adjacency with respect to this vertex. If the color is a match then it has violated the definition of a bipartite graph and this outcome provides a proof by contradiction returning false. As a graph G either has or does not have two distinct sets of vertices by definition Q.E.D.

Part 4. Provide a proof that your algorithm has runtime ≤ $O(|V| + |E|)$

To prove the implementation has runtime ≤ $O(|V| + |E|)$ consider that in RCL (2018) the runtime of a breadth-first search starting at the beginning to search for each vertex unvisited using Search() is $\Theta(|V| + |E|)$. The entire graph must be searched, this is clear. Then the cost of the for loop to walk the vertices and color them is also bounded by the same vertices V. As is the internal loop to walk the adjacency lists, as a visited vertex is not visited again. Using aggregate analysis as in the last assignment these values equal up to ≤ $O(|V| + |E|)$ as they are the same vertices and edges and no further elements to be bounded by.

References

1. Bipartite Graphs, https://www.geeksforgeeks.org/bipartite-graph/
2. Rosen, Discrete Mathematics and its Applications, 7th Ed. 2012. Pp 656 -658.