Zijian Gao
26319560
3/24/2016
CS311 Graph Coloring Project

Jar Details:

- Simple scanner that reads the file at the location of the first argument given to the file (run using the command “java -Xss512m -jar 2Color.jar [Input Graph]”
- All output is dumped to stdout, an output redirection “>” can be used to direct output to a file.
- The algorithm, when implemented recursively in Java requires increased stack space, which is given with the “-Xssxm” command.

Data Structures:

- **Graph.java**, a graph represented using the adjacency list representation. For an edge that exists between nodes *u* and *v,* it is true that *G.adj(u)* contains *v* and *G.adj(v)* contains *u.*
- **Colorable.java,** which encompasses the required stored states and algorithm to to determine if a given graph is 2-colorable.
- **Main.java**, the main entry point. The edge data is parsed here using a simple scanner and string split, adding the edges to the Graph object.

Algorithm Design:

This graph coloring algorithm uses a greedy approach, selecting nodes in ascending order to serve as the source nodes to run depth-first search from every node until either every node has been visited once, or a odd length cycle has been found.
A color is defined as either true for colored, or false for not, which is sufficient for two colors.
Nodes are assigned colors such that if a node is adjacent to some parent node, it will have an opposite color from its parent.
When the search finds a visited node that is the same color as its current source node, it has found a cycle and the definition of two colorable is violated.

The colors, parent node, and visited state per vertex are saved as arrays of length V.

The algorithm takes a text-formatted graph as input, and outputs true/false for colorability as well as the cycle, if it is not colorable.

Steps:

```
1. Read edges from file and create Graph object
2. While there are unvisited nodes, perform a modified DFS from source as follows:
    1. Mark s, source node, as visited. Color source nodes to True.
    2. For each node v in G.adj(s):
       If v is not visited
           Parent[v] = s
           Color[v] = !Color[s]
       Else if v has the same color as s
         v must have been discovered already further up the path, unwind the cycle by iterating
       through the parents of s until v is reached again
         return the cycle, graph is not colorable
3. If there are no vertices left and the DFS did not reach an odd cycle, return true
   Else return the cycle and false
```

Proof of Correctness:
It is a property of DFS that a DFS from any source *s* will explore all paths from that source such that any cycles in the same connected component will be found as a back-edge in the DFS tree.

The color that a node is started with can be arbitrarily chosen. This is because if a valid coloring existed for this graph, the colors could be inverted such that all nodes have the opposite color, in that way any node can be colored either one or the other color and so it does not matter what color is the starting color. In other words, there is no “correct” source node coloring as any correct coloring can just be inverted into an equivalently correct coloring, and this allows the DFS to correctly start coloring from any node using a sentinel True value.

This algorithm uses Koenig’s Theorem which states that any graph is two-colorable if and only if there do not exist any odd-length cycles. An odd-length cycle is a cycle from *u* to *v* that is found during the search where *u* and *v* have the same color, and v has been visited already.

If the algorithm reaches a node *v* from *u* that is the same color, it is correctly a cycle because of the properties of a DFS. The cycle must be odd-length because of the inverse of Koenig’s theorem.

Because of the nature of the two coloring problem, a node that is a descendent of a parent *u* in some DFS tree must always be colored the opposite color of the parent. This allows for the algorithm to correctly greedily alternatively color the nodes.

The cycles are correctly unchained because any cycle reachable from *u* to *v* must contain the edge from *u* to *v* and the subtree of the DFS tree ending at *u* up towards the parent node of *v,* which can be resolved by traversing the parents of *u* until *v* is reached again.

All cycles returned by this algorithm are guaranteed to be odd-length as the cycle length is checked in the end.

The DFS is guaranteed to terminate because it is run until there are no vertices left.

Time Analysis:

Step 1 takes $O(E)$ time to read the edges
Step 2 takes $O(V+E)$ time as this is a modified DFS with only constant time operations added. On a BFS, nodes must be initialized for $O(V)$ time and searched with $O(E)$ time for a total of $O(V+E)$. There is an added time of some factor, up to the largest cycle that could exist if a cycle is detected in order to unwind the cycle. This is only done once and the largest possible cycle is of size $O(E)$.

Step 3 takes $O(E)$ to check the cycle

Runtime: $O(V+E)$