Project 8: Graph Coloring

Using a graph coloring algorithm to solve problems.

Overview

Many real-world problems can be represented by graphs and solved by applying various algorithms to the graph (GPS navigation, utility networks, social networks are just some of the systems that rely on graphs). In this assignment, you will implement a graph structure and a graph coloring algorithm. You will then use the graph and graph-coloring algorithm to solve various problems involving conflict resolution and finding an optimal assignment of resources to accomplish a task or goal.

Learning Goals:

- Demonstrate an understanding of undirected, unweighted, graphs, sufficient to implement several non-trivial methods.
- Accomplish the implementation of a graph representation, and the implementation of a greedy graph coloring algorithm.
- Apply the graph and algorithm implementation to solve several problems.
- Gain further experience debugging and testing code.

Project Specification:

Background

Graph Coloring

Given an undirected graph, a graph coloring is an assignment of labels, also called "colors", to each vertex. A graph coloring must have the property that any two adjacent vertices must not be assigned the same color.

The coloring problem has its origins in map coloring, where any adjacent states or countries in a map do not share the same color.

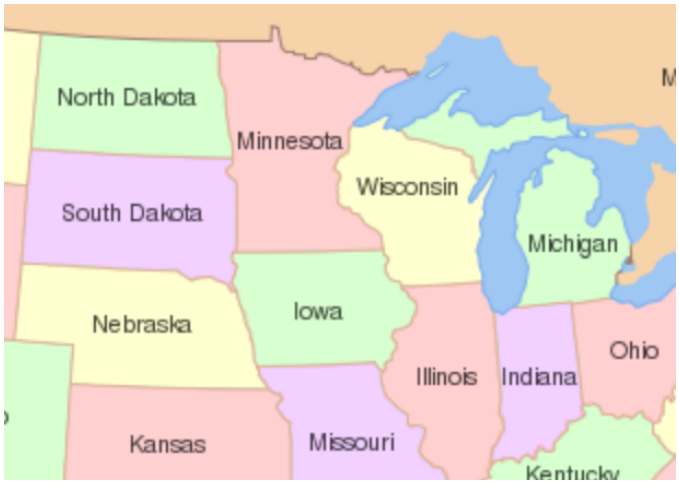


Figure 1: US States and their Colors in a Map.

In map coloring, as in the map above, states are represented as vertices and adjacent states share an undirected edge in the graph. A graph coloring algorithm will assign a color to each state such that no adjacent states share the same color. Furthermore, the algorithm should use the minimum number of colors.

Note also that the graph of all 50 of the US states is not connected (review what the graph term “connected” means as Hawaii and Alaska are not adjacent to any other states in the graph. Since these states are not adjacent to any other states, assigning one color to them is sufficient. In fact, a graph of any number of vertices can be colored with one color if there are no edges at all.

How difficult is the graph coloring problem? Deciding whether a given graph is k-colorable for k > 2 is an NP-complete problem. That means for large graphs, an exact solution cannot be computed with the current means of computation used by our computers. Luckily, for small graphs we can generally compute an exact solution.

What approach can be taken to solve this coloring problem? Imagine a “brute-force” search: consider every possible assignment of k colors to the vertices, and check whether any of them are correct. This approach would be extremely computationally expensive, with a runtime on the order of O((n+1)!). Yes, the ! means factorial.

Another approach is called a “greedy” algorithm. A ***greedy algorithm*** proceeds in a step by step manner, making the optimal choice at each step, without considering the overall state of

the problem. For example, given a vertex to color, look at its adjacent vertices and choose the color not used by its neighbors.

For example, assume we have an undirected, connected graph, and we have “colors” represented as integers: 0, 1, …, k. We want to assign a color to each vertex such that no adjacent vertices have the same color. A greedy algorithm could be:

for each vertex V_i in the graph:
  assign V_i to the lowest color not used in any adjacent vertex.
  if all k colors are used by the adjacent vertices, the graph cannot be k-colored.

You might see some similarities with breadth-first search, since with this approach the graph is traversed in that manner. This algorithm works well on reasonably sized graphs.

Applications

To illustrate graph coloring, we present an example with map coloring, which is one of the original problems that was solved by this method. Let’s say we have these US states (see Fig 1): Wisconsin, Minnesota, Iowa, Indiana, Michigan. We represent the states by vertices in a graph. The edges represent states that are adjacent to each other in the map. We then have this graph:

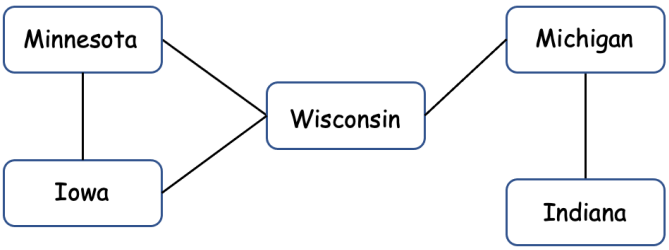


Figure 2: Graph of the 5 States.

What is the minimum number of colors required to color this graph? If we apply the greedy algorithm we can obtain this coloring, using integers instead of actual colors:

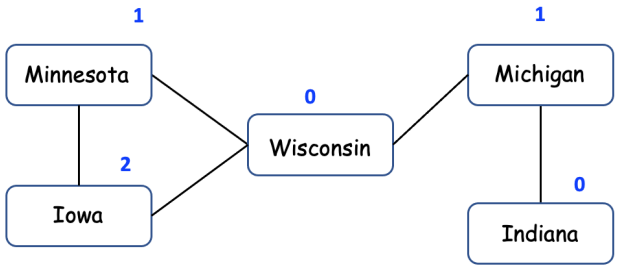


Figure 3: Coloring of States Map.

The minimum number of colors, also called the chromatic number, for this graph is 3. What if all of the US states were included? The chromatic number of that graph is 4. What about unconnected states such as Hawaii and Alaska? They could be represented as connected to one other state, or they could be unconnected vertices. That depends on the implementation of the problem. In any case, they can be colored with any color previously used in the connected graph. You do not have to deal with unconnected vertices in this project.

The graph coloring algorithm can be applied to many optimization problems, not just map coloring. For example, scheduling problems, assigning tasks to workers, assigning frequencies to cell phones, and generally resolving conflicts of any kind. Given a problem of this kind, you can represent it as a graph of vertices, edges, and colors, and then apply the coloring algorithm to find a coloring and the chromatic number of the graph, which is the minimum number of colors required.

Code Structure.

This project contains the following important folders (see figure 4):

src/graph: This is the source folder where all code you are submitting must go. All of your tasks are in the **`UndirectedUnweightedGraph`** class for this project.
src/test: This is the package folder where you can find all of the public unit tests.
lib: This is where you can find libraries that are included with the project. At the very least you will find two jar files that are used to run the JUnit test framework.

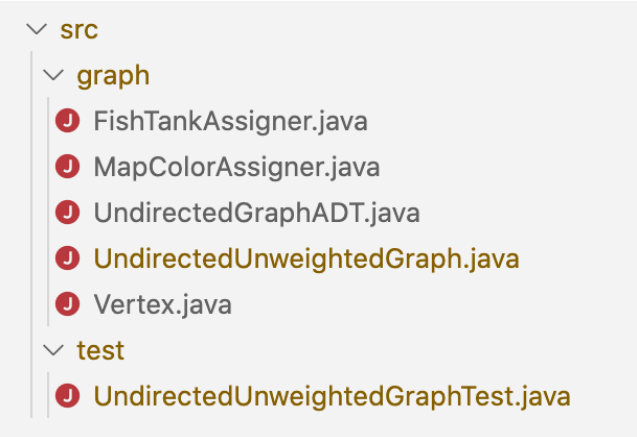


Fig 4: Code structure for the project.

Your main task is to complete the implementation of the **`UndirectedUnweightedGraph`** class. That class implements the **`UndirectedGraphADT`** Interface. Note that the generic type T should provide an implementation of the **`equals`** method. Remember that using the == operator in Java is not the same as calling the **`equals`** method.

There are two classes: **`MapColorAssigner`** and **`FishTankAssigner`** that allow you to test your implementation by applying your **`UndirectedUnweightedGraph`** class to a map coloring problem and a fish to aquarium allocation problem. The fish to aquarium allocation problem is identical to the graph coloring problem in that we want to use the fewest number of aquariums possible (colors in the map coloring problem), and certain fish (places on the map) cannot live together in the same aquarium (are adjacent on a map).

Tasks and TODOs

The project TODOs denote where you need to make changes in the source code. We do this using comments with the word TODO written in the comment. Most IDEs typically recognize TODO comments and provide an interface for navigating each TODO. VSCode has the ***Todo Tree*** extension which you can install if you haven't already done so.

https://marketplace.visualstudio.com/items?itemName=Gruntfuggly.todo-tree

Complete `UndirectedUnweightedGraph.java`

This class must provide implementations for all public methods in the **`UndirectedGraphADT`** interface. You will decide on how to represent the undirected graph for this project. You may use any classes from the `java.util library`, such as `ArrayList`. Review the text and lecture material on graphs for ideas.

The **`Vertex`** class is provided for you to use in your graph. One way to represent a graph is with an array of Vertex objects and an array (either 2D or 1D arrays can work) of boolean values that represents an adjacency table (see Figure 5).

Vertices

0	Wisconsin
1	Minnesota
2	Iowa
3	Michigan
4	Indiana

Adjacency Table

	0	1	2	3	4
0	F	T	T	T	F
1	T	F	T	F	F
2	T	T	F	F	F
3	T	F	F	F	T
4	F	F	F	T	F

Figure 5: Array Representation of a Graph.

Note that the diagonal in the adjacency table indicates that a vertex is not considered adjacent to itself. This is an arbitrary decision and depends on your implementation.

Java note: Due to the way that generics works with arrays in Java, it is easier to work with an **`ArrayList of Vertex<T>`** objects rather than working with an array.
Examples:
This is a declaration of a reference to an ArrayList of Vertex objects:

```
ArrayList<Vertex<T>> vertices;
```

This is an instantiation of the ArrayList:

```
vertices = new ArrayList<Vertex<T>>();
```

For an ArrayList that stores generic type T:

```
ArrayList<T> list = new ArrayList<T>();
```

You may use a linked implementation instead of the array implementation, or a combination of the two.

Your class must also represent colors as well as vertices and edges. Colors are represented as integers (0, 1, …, k) in this project. If you use a greedy algorithm, you really only need to keep track of the colors used by a vertex's neighbors. One idea is to use an array of boolean values that are true if a color is used in the graph. The index could be used as the color value. So, the first color would be 0, and if it is used in the graph its value at index 0 would be true. If the value of index 1 was false, that means color 1 is not currently used in the graph. The chromatic number of a graph is the total number of colors used in the coloring.

Note that there are two limits that you must work with:

1. the maximum number of vertices allowed in the graph, **`MAX_VERTICES`**, and
2. the maximum number of colors allowed, **`MAX_COLORS`**. The number of vertices that are actually in the graph depends on how many vertices have been added by calling the addVertex method. This may not be the same as **`MAX_VERTICES`**. Likewise, **`MAX_COLORS`** is a limit on the number of colors that may be used to color a graph. So colors are integers in the range of 0 to **`MAX_COLORS-1`** (the -1 is because we include 0 as the first color). Your algorithm will most likely not use all of the colors available.

Tasks

Your first task is to decide on an implementation of the graph. Then declare the variables you need and initialize them in the constructor. It is recommended that you design your implementation before coding it. Use a simple example of a graph, say three vertices and two edges to start. Then step through how you will add a vertex and an edge, and then implement the rest of the required methods. Keep the **`getChromaticNumber`** method, the graph coloring algorithm, for last. Test that your graph representation works before attempting the coloring algorithm code.

You may write any private methods you wish to support your code. We encourage you to do this as it makes your code easier to read, test, and develop. It is the proper way to write code.

Complete the **`getChromaticNumber`** method.

Once you have your graph implementation working, you can concentrate on implementing the coloring algorithm. This algorithm may (and should) utilize private helper methods that you create. Your algorithm should assign colors to **`Vertex`** objects in the graph, and the **`getChromaticNumbe`**r method should return the minimum number of colors that are required to color the graph. For example, the graph coloring in figure 3 shows a coloring and chromatic number of 3.

The following is pseudocode for a possible implementation of **`getChromaticNumber`**:

```
Returns the minimum number of colors required to color the vertices
for this graph as determined by a graph coloring algorithm.  Colors
are represented by integers  >0 and <= MAX_COLORS.
An error is generated if more than the maximum number of colors are
required to color this graph.

int getChromaticNumber() {
      int highestColorUsed = -1;
      int colorToUse = -1;
      // Start with a vertex (we are dealing with a connected graph).
      for each Vertex curVert in vertices   {
      if curVert is not yet colored
            colorToUse = getColorToUse(curVert);//a private method
            set  curVert's color to colorToUse
            update highestColorUsed if colorToUse is > highestColorUsed
      }
      }
      return highestColorUsed
   }
```

*Remember that the number of available colors is limited, so this method, or some helper method, should throw an Exception if the available colors have all been used by a vertex's adjacent vertices and another color is needed.

More about the pseudocode above:
You must find the lowest color that is not used by the vertices that are adjacent to the current vertex v. For example, if these colors are used: 0, 2, 3, then you would select color 1 to assign

to the current vertex. You probably want to record that this color is used for the graph as a whole, so that you can easily report on the total colors used at the end (the chromatic number). If all possible colors are used by the set of adjacent vertices, an Exception should be thrown because more than the maximum number of colors are required to color this vertex, and therefore, the graph.

This implementation keeps track of the highest color used and returns that number + 1. The reason 1 is added is that the colors start at 0, and the method returns the total number of colors used. This assumes that the colors that are assigned are always the lowest available color, so that there are no “gaps” in the final coloring. This is because of the way the greedy algorithm chooses coloring of a vertex based on its neighbors. For example, at the end you would not have this coloring at the end of the algorithm: 0, 1, 2, , 4, where color 3 was not used. If this were the case, then you would have to report the chromatic number differently.

Private Methods

Private methods are mostly used as “helper” methods; they are methods that are called from other methods to solve a bigger problem.

When you are coding (and designing) a procedure, you will often find that there are smaller tasks that need to be done. One example in the pseudocode for **`getChromaticNumber`** is the method **`getColorToUse`**, which finds the color to use for the current Vertex- or throws an Exception if there are no more colors left (if all of the adjacent Vertex objects have been assigned all of the available colors). Pseudocode for this method is provided below:

```
This method returns the color to assign to the vertex passed in.
It uses an array to keep track of the colors used by the adjacent
vertices,
and then assigns the lowest unused color to the curVertex. Since
colors are represented by integers  >0 and <= MAX_COLORS, the color
can be used as an index in a boolean array.
An error is thrown if there are no unused colors as the graph cannot
be colored with the current max colors allowed. Assume colors are
integers

 int getColorToUse(curVertex)
      int colorToUse = -1;
      boolean[] adjColorsUsed = new boolean[MAX_COLORS];// initially
all false.
      adjVertsList = getAdjacentVertices(curVertex); //a private method
      for each curAdjVert in adjVertsList  {
          if(curAdjVert is colored)
```

```
                    mark adjColorsUsed at this color to be true;
        }
        // use the array to find the lowest color to use:
        colorToUse = first unused color in the adjColorsUsed array
        if (allColorsUsed)
                    error: "All colors have been used, the Max number of
colors has been exceeded."
        else
                    return colorToUse;
    }
```

This taskNotice the method above calls another private “helper” method: **`getAdjacentVertices`** method. This task could be done by a private method that takes a Vertex as an argument and returns a List of adjacent Vertex objects.

The private methods you write will depend on how you design your algorithm and how you have implemented the graph, but the idea of allocating smaller tasks to helper methods is the same.

Testing

We provide some unit tests to aid your development. We strongly suggest you also use the code in the **`MapColorAssigner`** and **`FishTankAssigner`** to test your graph code.

For example, after completing **`UndirectedUnweightedGraph,`** you can uncomment the following code:

```
/**
   ArrayList<String> stateAdjacencies =  graph.getAdjacentData("Wisconsin");
      for(String item: stateAdjacencies) {
   System.out.println(item);
}
*/
```

Then, you can run **`MapColorAssigner`** to get the adjacencies for the state of Wisconsin:

```
The number of colors required is: 4
Michigan
Minnesota
Illinois
Iowa
```

Feel free to modify the code in these files for your testing purposes as they will not be evaluated.

These are some cases to use with the resulting chromatic number (CN):

1. A graph of one Vertex. CN = 1
2. A graph of two Vertices with one edge. CN = 2
3. A - B - C CN = 2
4. A CN = 3

   / \

   B - C

Export and Submit

Step 1: Export your project

Use the Archive extension or other means to create the correct zip file. If your zip file is not in the correct form, the autograder will not process your code and you will not receive any credit. The autograder will not run (often stating an error occurred) if your code does not compile or if your zip file is not correctly made.

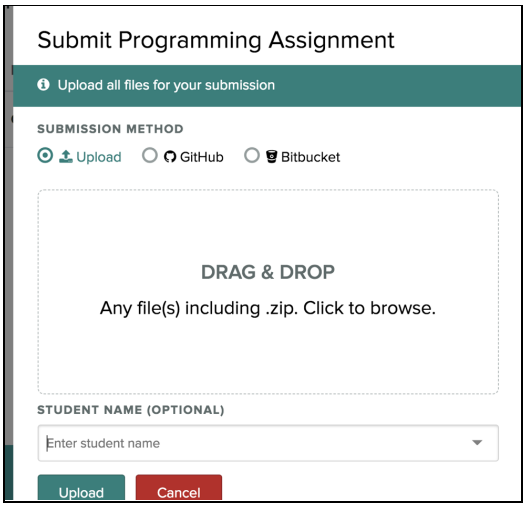
Important

Your top level directory must be named: `Proj7-GraphColoring-starter`

Do not import any libraries or change the starter code method signatures, or modify any existing instance variables or method signatures.

Step 2: Submit the zip file to Gradescope

Log into Gradescope, select the assignment, and submit the zip file for grading.



There are usually more tests in the autograder than provided with the starter code. If your code passes all provided tests, it is a good indication that your code will pass all of the autograder tests. If that is not the case, you are likely not considering some of the “edge” cases in your algorithm. Re-examine your code, use the debugger to troubleshoot. Pose specific questions on

piazza for some outside help. Remember that these projects take longer than you estimate. Starting early means you have more time to get help if you are stuck.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does. Attend office hours for help or post your questions in Piazza.