Advanced Functional Programming – Autumn 2020
Assignment 2 - Graph Coloring

Li Ju

27th November 2020

Implementation of graph data structure
The implementation of graph data structure is very straightforward: a list of tuples is maintained. Each tuple (V, Es) in the graph is consisted of two parts: V is an term represents the vertex and Es is a list of terms representing all neighbours of the vertex V.
Together with the data structure definition, 6 functions are created as well, and described as following:

1. newGraph: a helper function to generate an empty graph
2. insertEdge: a function to insert an edge to an graph
3. insertVertex: a function to insert a vertex to an graph
4. getAdj: a function to get all neighbour vertices of a vertex in a graph
5. getEdges: a helper function to get all edges of a graph
6. getVerts: a helper function to get all vertices of a graph

Pros:
a). The implementation is by accident exactly the same as the given input format in the assignment, so there is no need to construct the graph by adding vertices and edges one by one. Instead, the input can be used for calculation directly.
b). All neighbour vertices of each vertex are collected already. When calculating adjacent vertices, this leads to performance advantage.
Cons:
The graph in the assignment is undirected, however in the implementation, each edge is recorded with two vertex pairs. This involves some slight redundant.

Property-based testing for graph implementation
Property-based testing is conducted for the implementation of graph data structure. 2 properties are used:

1. For any random undirected graph, if vertex a is the neighbour of vertex b, then vertex b must be the neighbour of vertex a
2. For any random undirected graph, the set of vertices of edges must be a subset of all vertices (due to some unconnected vertices).

A new generator is constructed to generate random graph. Starting from a list of random integers and an empty graph, the generator insert an edge to the an empty graph recursively. 2 edges number required will be picked from the random list in order.
For the first test, a random vertex V will be checked whether V is the neighbour of a random neighbour of V. The test is named by prop_pairwiseAdj().
For the second test, all edges will be collected with function getEdges and check if all vertices of edges are subset of all vertices of the graph, returned by getVerts. The test is named by prop_edgeVerts
All 6 functions are covered in the 2 tests.

Algorithm of kcolor problem and implementation
Algorithm used for the problem can be referenced here[1]. The brute force algorithm can be described as following:

1. Generate all possible coloring combinations of given colors and all vertices
2. check color scheme one by one from the list of all combinations if the scheme is valid (e.g.no adjacent vertices are of the same color)
3. If a valid coloring is found, return the scheme, otherwise return Nothing

When implementing the algorithm, laziness of Haskell is used in two aspects: 1. both generating all possible combinations and check validity is done with list comprehension: this guarantees the scheme will be generated only if it is required. 2. To return as soon as the first valid coloring is found, half-applied function "take 1" is used: if the valid scheme is found, the program will not do computation for left schemes and return.
Monad Maybe is also used: if function "take 1" returns an empty list, indicating no valid scheme exists, Nothing will be returned, otherwise Just "valid scheme" is returned.

Property-based testing for kcolor
To test kcolor function, 2 property-based testings are conducted.
The first property is very loose and straightforward: if a graph can be colored, all vertices should be colored in the returned results. This test was done by call kcolor with random graphs (generated by generator described above) and number of vertices divided by 2. If it can be colored, number of colored vertices will be compared with number of all vertices in the graph: if equal, return true, otherwise false. This test is named by prop_colorLength().
The second property I used for testing kcolor function is that if the graph can be colored, then both vertices of each edge should be colored with different colors. This was done again by call kcolor with a random graphs and number of vertices divided by 2. For those who can be colored, each edge will be check that if vertices it connects are of different color. This test is named by prop_adjColor().

References

[1] Unknown contributors. "M-Coloring Problem - Backtracking-5 ". Geeksforgeeks, 2020, https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/