Functional $\Delta + 1$ Edge-Coloring Algorithm

Seth Yoder

May 26, 2015

This edge-coloring algorithm, written in the purely-functional programming language Haskell, follows the proof of Vizing's 1964 theorem, which states that the edges of every simple undirected graph may be colored using a number of colors that is at most one larger than the maximum degree Δ of the graph. The proof followed in this document is found in:

Gibbons, *Algorithmic Graph Theory.* Cambridge: Cambridge University Press, 1985: pp. 197-198

We begin the module by importing necessary functions:

```
module GraphTheory.Algorithm.EdgeColor
(deltaPlusOneColor, isValidColoring) where
import GraphTheory.Graph (Edge (..), Graph, vertices, edges,
  incidentEdges, degreeMap, highestDegree, adjacencyList,
  modifyEdgeUnsafe, setEdgeColor)
import GraphTheory.Algorithm.DFS (getComponents)
import Data.List ((\\), intersect, elemIndex, nub)
import Data.Maybe (fromJust)
```

The entry point of the algorithm is a wrapper around the induction loop.

```
deltaPlusOneColor :: (Ord v, Eq w, Eq v) => Graph v w Integer -> Graph v w Integer
deltaPlusOneColor g = inductLoop g g'
  where g' = g {vertices = [], edges = []}
```

In the proof, we suppose that all edges of a graph G have been properly colored using at most $(\Delta + 1)$ colors except for the edge (v_0, v_1). We model this algorithmically by building a graph G' iteratively from the edges of G. *doDeltaPlusOneColor* is called on each iteration of G'.

```
inductLoop :: (Ord v, Eq w, Eq v) =>
  Graph v w Integer -> Graph v w Integer -> Graph v w Integer
inductLoop g g'
  | null $ edges g = g'
  | otherwise = inductLoop g {edges = restEdges} $ doDeltaPlusOneColor nextg'
```

$$
\begin{array}{l}
\quad \textbf{where } \mathit{nextg}' = g' \{ \mathit{edges} = g'\_\mathit{edges}, \mathit{vertices} = g'\_\mathit{vertices} \} \\
\qquad (\mathit{firstEdge} : \mathit{restEdges}) = \mathit{edges}\ g \\
\qquad g'\_\mathit{edges} = (\mathit{firstEdge} \{ \mathit{edgeData} = -1 \}) : \mathit{edges}\ g' \\
\qquad g'\_\mathit{vertices} = \mathit{foldl}\ \mathit{vertCheck}\ (\mathit{vertices}\ g')\ [v0, v1] \\
\qquad\quad \textbf{where } \mathit{vertCheck}\ \mathit{oldVs}\ \mathit{newV} = \textbf{if } \mathit{elem}\ \mathit{newV}\ \mathit{oldVs}\ \textbf{then } \mathit{oldVs} \\
\qquad\qquad \textbf{else } \mathit{newV} : \mathit{oldVs} \\
\qquad\qquad (v0, v1) = \mathit{endpoints}\ \mathit{firstEdge}
\end{array}
$$

We know that there must be at least one color missing from v_0 and one color missing from v_1. If the same color is missing at both v_0 and v_1, this can be assigned to the edge (v_0, v_1) by calling the function *easyColor*. Otherwise, we call the function *hardColor* to continue with the harder cases.

$$
\begin{array}{l}
\mathit{doDeltaPlusOneColor} :: (\mathit{Ord}\ v, \mathit{Eq}\ w) \Rightarrow \\
\quad \mathit{Graph}\ v\ w\ \mathit{Integer} \rightarrow \mathit{Graph}\ v\ w\ \mathit{Integer} \\
\mathit{doDeltaPlusOneColor}\ g = \textbf{if } (\neg \circ \mathit{null})\ \mathit{missingAtBoth} \\
\quad \textbf{then } \mathit{easyColor}\ g\ (\mathit{head}\ \mathit{missingAtBoth}) \\
\quad \textbf{else } \mathit{hardColor}\ g\ (\mathit{head}\ \$\ \mathit{edges}\ g)\ \mathit{gColors} \\
\quad \textbf{where } \mathit{missingAtBoth} = \mathit{intersect}\ (\mathit{missingAt}\ v0)\ (\mathit{missingAt}\ v1) \\
\qquad (v0, v1) = \mathit{endpoints} \circ \mathit{head}\ \$\ \mathit{edges}\ g \\
\qquad \mathit{missingAt} = \mathit{colorsMissingAt}\ g\ \mathit{gColors} \\
\qquad \mathit{gColors} = \mathit{colorList}\ g
\end{array}
$$

The simple function *easyColor* is given a graph and an integer c, and it returns a modified graph with its first edge colored c, which terminates the current iteration of the induction loop.

$$
\begin{array}{l}
\mathit{easyColor} :: \mathit{Graph}\ v\ w\ \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Graph}\ v\ w\ \mathit{Integer} \\
\mathit{easyColor}\ g\ c = g \{ \mathit{edges} = \mathit{new}\_e : \mathit{es} \} \\
\quad \textbf{where } (e : \mathit{es}) = \mathit{edges}\ g \\
\qquad \mathit{new}\_e = e \{ \mathit{edgeData} = c \}
\end{array}
$$

hardColor is the entry point for the more complex case of the algorithm. We let C_0 be a color that is missing at v_0 and present at v_1, and let C_1 be a color that is missing at v_1 but present at v_0. We then pass these values to *makeColorSequence*.

$$
\begin{array}{l}
\mathit{hardColor} :: (\mathit{Ord}\ v, \mathit{Eq}\ v, \mathit{Eq}\ w) \Rightarrow \\
\quad \mathit{Graph}\ v\ w\ \mathit{Integer} \rightarrow \mathit{Edge}\ v\ w\ \mathit{Integer} \rightarrow [\mathit{Integer}] \rightarrow \mathit{Graph}\ v\ w\ \mathit{Integer} \\
\mathit{hardColor}\ g\ e\ \mathit{gColors} = \mathit{makeColorSequence}\ g\ \mathit{gColors}\ [c0, c1]\ [(v0, v1)] \\
\quad \textbf{where } (v0, v1) = \mathit{endpoints}\ e \\
\qquad \mathit{presentAt} = \mathit{colorsPresentAt}\ g\ \mathit{gColors} \\
\qquad \mathit{missingAt} = \mathit{colorsMissingAt}\ g\ \mathit{gColors} \\
\qquad c0 = \mathit{head}\ \$\ \mathit{intersect}\ (\mathit{missingAt}\ v0)\ (\mathit{presentAt}\ v1) \\
\qquad c1 = \mathit{head}\ \$\ \mathit{intersect}\ (\mathit{missingAt}\ v1)\ (\mathit{presentAt}\ v0)
\end{array}
$$

In Gibbons' words, "We construct a sequence of edges $(v_0, v_1), (v_0, v_2), (v_0, v_3) \ldots$ and a sequence of colors $C_0, C_1, C_2, C_3 \ldots$ such that C_i is missing at v_i and

(v_0, v_{i+1}) is colored C_i. Let the sequences at some stage of the construction be $(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_i)$ and $C_1, C_2, \ldots, C_i$."

Then, we find a v that is not already present in the edge sequence such that (v_0, v) has color C_i . If such a v does not exist, we terminate the sequence and call *hcCase*1. If such a v does exist but is already present in the edge sequence, we terminate the sequence and call *hcCase*2.

```
makeColorSequence :: (Ord v, Eq v, Eq w) ⇒
  Graph v w Integer → [Integer] → [Integer] → [(v, v)] → Graph v w Integer
makeColorSequence g gColors cSeq eSeq
  | null vLs = hcCase1 g cSeq eSeq
  | elem v (map snd eSeq) = hcCase2 g cSeq eSeq v
  | otherwise = makeColorSequence g gColors next_cSeq next_eSeq
  where c_i = last cSeq
    v0 = (fst ∘ head) eSeq
    v = (otherVertex v0 ∘ endpoints ∘ head) vLs
    vLs = take 1 $ edgesWithColor c_i
    edgesWithColor c = filter (vColorEqual c) $ incidentEdges g v0
    vColorEqual c e = edgeData e ≡ c
    next_cSeq = cSeq ++ [head $ colorsMissingAt g gColors v]
    next_eSeq = eSeq ++ [(v0, v)]
```

In case 1, we simply recolor each edge (v_0, v_i) for $i <= j$ with C_i. We can achieve this by zipping the edge sequence and the color sequence together (in this implementation, C_0 is at the beginning of the color sequence, so we use *tail cSeq* to ignore the first element), then folding the result over G' with the function *setEdgeColor*[1]. This terminates the current iteration of the induction loop.

```
hcCase1 :: (Eq v, Eq w) ⇒
  Graph v w Integer → [Integer] → [(v, v)] → Graph v w Integer
hcCase1 g cSeq eSeq = foldl setEdgeColor g (zip eSeq (tail cSeq))
  where buildEdge (v1, v2) = Edge (v1, v2) 1
```

Case 2 is more involved: We first recolor all edges (v_0, v_i) for $i < j$ with the color C_i, then uncolor (v_0, v_k) by setting its color to -1. The recolored graph is given to *kempeBranch*.

```
hcCase2 :: (Ord v, Eq v, Eq w) ⇒
  Graph v w Integer → [Integer] → [(v, v)] → v → Graph v w Integer
hcCase2 g cSeq eSeq vk = kempeBranch recoloredG cSeq eSeq vk
  where recoloredG = setEdgeColor (setEdgeColors g zipped) ((v0, vk), −1)
    v0 = (fst ∘ head) eSeq
```

[1]This results in some efficiency loss - *setEdgeColor* has worst-case $O(n)$ complexity, since we must find the correct edge in G' and modify it. A worthwhile improvement for this algorithm would be to use a hash table to store edge colors.

```
    vj = (snd ∘ last) eSeq
    eSeqBeforevk = takeWhile (λx → snd x ≢ vk) eSeq
    zipped = zip eSeqBeforevk $ tail cSeq
```

For some edge-colored graph G', the Kempe subgraph $H(C_1, C_2)$ is described as the subgraph induced by those edges of G' which are colored either C_1 or C_2. The Kempe subgraph of G' must either be a path or a circuit, since (due to the induction hypothesis) there is at most one edge colored C_0 and one edge colored C_j at any vertex. We let $H_{v_k}(C_0, C_j)$ (written *h_vk* in the code) denote the component of $H(C_0C_j)$ containing v_k. Similarly, we let $H_{v_j}(C_0, C_j)$ (written *h_vj* in the code) denote the component of $H(C_0C_j)$ containing v_j. If v_0 is not in $H_{v_k}(C_0, C_j)$, we call *branchA*. Otherwise, v_0 is not in $H_{v_j}(C_0, C_j)$, and we call *branchB*.

```
kempeBranch :: (Ord v, Eq v, Eq w) ⇒
  Graph v w Integer → [Integer] → [(v, v)] → v → Graph v w Integer
kempeBranch g cSeq eSeq vk = if notElem v0 (map fst $ h vk) ∧ notElem v0 (map snd $ h vk)
  then branchA g (h vk) c0 cj v0 vk
  else branchB g (h vj) c0 cj cSeq eSeq v0 vk
  where v0 = (fst ∘ head) eSeq; vj = (snd ∘ last) eSeq
    c0 = head cSeq; cj = last cSeq
    h = kempeCompEdges g c0 cj
    h_vk = h vk; h_vj = h vj;
```

If we reach branch A, we simply interchange the colors in $H_{v_k}(C_0, C_j)$, so that C_0 is missing at v_k. C_0 is also missing at v_0, so we color (v_0, v_k) with C_0. G' now has a proper coloring, so we terminate this iteration of the induction loop.

```
branchA :: (Eq v, Eq w) ⇒ Graph v w Integer → [(v, v)] → Integer → Integer →
  v → v → Graph v w Integer
branchA g hvk c0 cj v0 vk = setEdgeColor (interchangeColors g hvk c0 cj) ((v0, vk), c0)
```

If we reach branch B, we first recolor each edge (v_0, v_i) for $k \leqslant i < j$ with C_i and leave (v_0, v_j) uncolored. Since neither C_0 or C_j are used in this recoloring, $H(C_0, C_j)$ remains unaltered. We then interchange the colors C_0 and C_j in $H_{v_j}(C_0, C_j)$, so that C_0 is absent from v_j. We also know that C_0 is absent from v_0, so we can color (v_0, v_j) with C_0. G' has a proper coloring, so this iteration of the induction loop can terminate by returning G'.

```
branchB :: (Eq v, Eq w) ⇒ Graph v w Integer → [(v, v)] →
  Integer → Integer → [Integer] →
  [(v, v)] → v → v → Graph v w Integer
branchB g hvj c0 cj cSeq eSeq v0 vk = setEdgeColor interchanged_g ((v0, vj), c0)
  where k = fromJust $ elemIndex vk (map snd eSeq)
    vj = (snd ∘ last) eSeq
    cSeq_end = init $ drop (k + 1) cSeq
    eSeq_end = init $ drop k eSeq
```

$$recolored\_g = setEdgeColors\ g\ (zip\ eSeq\_end\ cSeq\_end)$$
$$interchanged\_g = interchangeColors\ recolored\_g\ hvj\ c0\ cj$$

With that, the $(\Delta + 1)$ edge coloring is complete in all cases. Below are the various helper functions that were used throughout this code.

Helper functions

otherVertex is given a vertex and a vertex pair, and it returns the other vertex in the pair. This algorithm can only use otherVertex in a safe manner; otherwise the algorithm will terminate with an error.

```
otherVertex :: (Eq v) ⇒ v → (v, v) → v
otherVertex v (v1, v2)
  | v ≡ v1 = v2
  | v ≡ v2 = v1
  | otherwise = error "otherVertex failed."
```

colorList is given a graph, and it returns a default list of usable colors for the graph, based on the Δ value for the graph.

```
colorList :: (Ord v) ⇒ Graph v w Integer → [Integer]
colorList g = [1 . . highestDegree g + 1]
```

colorsMissingAt is given a graph, a list of available colors for the graph, and a vertex. It returns a list of colors that are not present at the vertex.

```
colorsMissingAt :: (Ord v, Eq c) ⇒ Graph v e c → [c] → v → [c]
colorsMissingAt g gColors v = gColors \\ map edgeData (incidentEdges g v)
```

colorsPresentAt is given a graph, a list of available colors for the grpah, and a vertex. It returns a list of colors that are present at the vertex.

```
colorsPresentAt :: (Ord v, Eq c) ⇒ Graph v e c → [c] → v → [c]
colorsPresentAt g gColors v = intersect gColors ∘ map edgeData $ incidentEdges g v
```

kempeSubgraph is given a graph G and two colors C_1 and C_2, and it returns a subgraph of G that contains only the edges colored C_1 or C_2.

```
kempeSubgraph :: Graph v w Integer → Integer → Integer → Graph v w Integer
kempeSubgraph g c1 c2 = g {edges = filter newEdgeF $ edges g}
  where newEdgeF e = let c = edgeData e in c ≡ c1 ∨ c ≡ c2
```

interchangeColors is given a graph, a list of edges, and two colors. The graph is updated by interchanging the colors of each edge present in the edge list.

```
interchangeColors g componentEdgeList c1 c2 = foldl setColor g componentEdgeList
  where setColor gr edge = modifyEdgeUnsafe gr edge edgeFunc
```

$$\begin{aligned}&\mathit{edgeFunc}\ e = \mathbf{let}\ \mathit{newColor} = \mathbf{if}\ \mathit{edgeData}\ e \equiv \mathit{c1}\ \mathbf{then}\ \mathit{c2}\ \mathbf{else}\ \mathit{c1}\\ &\quad \mathbf{in}\ e\ \{\mathit{edgeData} = \mathit{newColor}\}\end{aligned}$$

$setEdgeColors$ is given a graph and a list of tuples. Each tuple t contains a tuple of vertices and a color. The list is folded over the graph with the function $setEdgeColor$, which takes a graph and a tuple $((v_1, v_2), c)$ and returns the graph with edge (v_1, v_2) set to color c.

$$\begin{aligned}&\mathit{setEdgeColors} :: (\mathit{Eq}\ v, \mathit{Eq}\ e) \Rightarrow \mathit{Graph}\ v\ e\ c \to [((v, v), c)] \to \mathit{Graph}\ v\ e\ c\\ &\mathit{setEdgeColors} = \mathit{foldl}\ \mathit{setEdgeColor}\end{aligned}$$

$isValidColoring$ is used for testing this algorithm. It determines if the graph has a valid edge coloring.

$$\begin{aligned}&\mathit{isValidColoring} :: (\mathit{Ord}\ v) \Rightarrow \mathit{Graph}\ v\ e\ \mathit{Integer} \to \mathit{Bool}\\ &\mathit{isValidColoring}\ g = \mathit{all}\ (\mathit{noneRepeated} \circ \mathit{colorsAt})\ (\mathit{vertices}\ g)\\ &\quad \mathbf{where}\ \mathit{colorLs} = \mathit{colorList}\ g\\ &\qquad \mathit{colorsAt}\ v = \mathit{foldl}\ (\mathit{foldF}\ v)\ [\,]\ (\mathit{edges}\ g)\\ &\qquad \mathit{foldF}\ v\ \mathit{cls}\ e = \mathbf{let}\ (\mathit{v1}, \mathit{v2}) = \mathit{endpoints}\ e\ \mathbf{in}\\ &\qquad\quad \mathbf{if}\ v \equiv \mathit{v1} \lor v \equiv \mathit{v2}\ \mathbf{then}\ \mathit{edgeData}\ e : \mathit{cls}\\ &\qquad\quad \mathbf{else}\ \mathit{cls}\end{aligned}$$

$noneRepeated$ returns true if there are no duplicates in a list.

$$\begin{aligned}&\mathit{noneRepeated} :: (\mathit{Eq}\ a) \Rightarrow [a] \to \mathit{Bool}\\ &\mathit{noneRepeated}\ \mathit{ls} = \mathit{ls} \equiv \mathit{nub}\ \mathit{ls}\end{aligned}$$

$kempeCompEdges$ is given a graph, two colors C_1 and C_2, and a vertex v. It returns a list of the edges (represented as vertex tuples) in the kempe subgraph $H(C_1, C_2)$ that are in the same component of $H(C_1, C_2)$ as v.

$$\begin{aligned}&\mathit{kempeCompEdges} :: (\mathit{Ord}\ v, \mathit{Eq}\ v, \mathit{Eq}\ w) \Rightarrow\\ &\quad \mathit{Graph}\ v\ w\ \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \to v \to [(v, v)]\\ &\mathit{kempeCompEdges}\ g\ \mathit{c1}\ \mathit{c2}\ \mathit{vert} = \mathit{map}\ \mathit{endpoints}\ \$\ \mathit{filter}\ \mathit{kempeFilter}\ (\mathit{edges}\ \mathit{kempe})\\ &\quad \mathbf{where}\ \mathit{kempeFilter}\ e = \mathbf{let}\ k = \mathit{kempeCompVertices}; (\mathit{v1}, \mathit{v2}) = \mathit{endpoints}\ e\\ &\qquad \mathbf{in}\ (\mathit{elem}\ \mathit{v1}\ k \land \mathit{elem}\ \mathit{v2}\ k)\\ &\qquad \mathit{kempeCompVertices} = \mathit{map}\ \mathit{fst}\ \$\ \mathit{filter}\ \mathit{f1}\ \mathit{kempeComponents}\\ &\qquad \mathit{f1}\ \mathit{tuple} = \mathit{snd}\ \mathit{tuple} \equiv \mathit{componentIndex}\\ &\qquad \mathit{componentIndex} = \mathit{snd} \circ \mathit{head}\ \$\ \mathit{filter}\ \mathit{f2}\ \mathit{kempeComponents}\\ &\qquad \mathit{f2}\ \mathit{tuple} = \mathit{fst}\ \mathit{tuple} \equiv \mathit{vert}\\ &\qquad \mathit{kempeComponents} = \mathit{getComponents}\ \mathit{kempe}\\ &\qquad \mathit{kempe} = \mathit{kempeSubgraph}\ g\ \mathit{c1}\ \mathit{c2}\end{aligned}$$