Introduction and Background

Outline

This report details how some well known concepts in graph theory can be applied to solve problems in communications, specifically zero-error communication.

Background

Graphs

Graphs are entities made up by vertices/nodes (which can represent many different classes of information) and edges between those vertices. The problems addressed in this report deal only with *undirected* graphs, that is, graphs whose edges do not contain any information about direction between two vertices.

Strong Product

[todo]

Coloring

The graph coloring problem looks to find a minimal set of colors that can be assigned to the vertices of a graph, such that no two vertices connected by an edge share a color. Because finding an *optimal* (i.e. minimal size) set of colors requires consideration of all subsets of the vertices of a graph, all known methods of finding such colorings are of *exponential* time complexity with respect to the input. In other words, it takes an impractical amount of time to find an optimal coloring of a graph for all but the smallest input sizes.

Independent Sets

An independent set of vertices of a graph is a set such that no two vertices in the set are connected by any edge. A *maximum* independent set for a graph is the (or one of the) independent sets of greatest size. For the same reason as the graph coloring problem, solving the maximum independent set problem is also extremely time consuming for large graphs.

Algorithm Description

Coloring

The basis of the coloring algorithm used in this report is as follows:

For each vertex, check if it can safely (without breaking correctness) be assigned a color. If so, recursively attempt to color all remaining vertices. If this is not possible, unassign the assigned color. If this subroutine fails for all vertices, the graph is not colorable with the specified number of colors.

This routine is performed for all colorings 1 through k (some upper bound) until either the first, smallest correct coloring is found, or no coloring is found and therefore the graph is not k-colorable.

Maximum Independent Set

The basis of the maximum independent set algorithm used in this report is as follows:

- If the size of G is 0, return 0
- If the graph has any vertex V with degree 0:
    - return 1 + MIS(G - v)
- If G has any vertex V with degree 1 connected to W:
    - return max(1 + MIS(G with V's neighborhood removed), 1 + MIS(G with W's neighborhood removed))
- If G has any vertex with degree > 2:
    - return max(1 + MIS(G with V's neighborhood removed), MIS(G without V's neighborhood removed))
- else (all vertices U,V,W have degree 2):
    - return max(1 + MIS(G with U's neighborhood removed), 1 + MIS(G with V's neighborhood removed), 1 + MIS(G with W's neighborhood removed))

Problem Setup

Consider an nxnxn matrix S where a given x index is a value of X 0 to (n-1) containing an nxn matrix with rows given by Y values 0 to (n-1) and columns given by Y_r values 0 to (n-1). First, find the confusability graph $G_{X|Y,Y_r}$. This graph is defined as follows:

- vertices given by X
- edge between two vertices (x, x') if there exists some (y, y_r) pair such that (S[x][y][y_r])*(S[x][y][y_r]) > 0

Next, find the 2-fold strong product of that graph as defined previously. Then, generate the list of graphs $G_{R|K}$, whose vertices are the y_r values that are reachable via the x values in K, which is a maximum independent set of $G_{X|Y,Y_r}$. An edge exists between two vertices (y_r, y'_r) reachable in S if all of the following are true for some values y, x, x', and a given MIS, K:

- x and x' are in K
- x != x'
- y, x, and x' are reachable in S
- S[x][y][y_r] > 0
- S[x][y][y'_r] > 0

Next, generate the list of $G_{R|K}^2$ graphs with the 2-fold confusability graph in a similar manner, where edges exist between two reachable vertices (($y_{r,\ i}$, $y_{r,\ j}$),($y'_{r,\ i}$, $y'_{r,\ j}$)) if the following are true for some pairs (x_i, x'_i), (x_j, x'_j), (y_i, y_j):

- [todo]

Documentation of Functionality

coloring

`countColorsUsed(graph)`

- Description:
  Returns the number of colors used for a given networkx graph

- Inputs:

  - graph, a networkx graph

- Returns:

  - number of colors used in the graph (integer)

- Example usage:

```
if countColorsUsed(graph) < someValueOfInterest:
  somethingCoolHappens()
else:
  nothingCoolHappens()
```

`displayColoring(graph, name)`

- Description:
  Uses matplotlib to display *graph*'s nodes/edges and colors. Outputs the result as a png to the path specified in *name*

- Inputs:

  - graph, a networkx graph
  - name, the filename/path desired for the output

- Returns:

  - None

- Example usage:

```
... #some graph is generated
if minimalColoring(graph, maxValue) < maxValue:
  displayColoring(graph, '~/Documents/examples/mygraph')
```

`colorIsSafe(graph, neighbors, color)`

- Description:
  Utility function used in graph coloring.
- Inputs:
- graph, a networkx graph
- neighbors, the neighbors of a given node
- color, the color to check
- Returns:

    - True if the color is safe, `False` otherwise

`verifyColoring(graph)`

- Description:
  Utility function to verify that a coloring is correct (no two adjacent nodes share a color).
- Inputs:
    - graph, a networkx graph
- Returns:
    - True if the coloring is correct, otherwise `False`
- Example usage:

`colorUtil(vertex, vertexList, graph, k, n)`

- Description:
  Used to optimally color a graph using recursive color assignment. Note that in the worst case this can take an extremely long time, especially for large graphs. (It is probably a good idea to call this function indirectly using `minimalColoring()`)
- Inputs:
    - vertex, a networkx graph node
    - vertexList, a list object containing the nodes of a graph
    - graph, a networkx graph
    - k, the desired upper bound on colors for the graph
    - n, the index of the current vertex in the vertexList
- Returns:
  Modifies `graph` in place and returns True if a k-coloring was possible otherwise `False`.

`minimalColoring(g, r)`

- Description:
  Finds an optimal coloring of a graph g.
- Inputs:
    - g, a networkx graph
    - r, a desired lower bound on the number of colors for g (if none desired, call using `len(g.nodes())`
- Returns:
  The chromatic number of `g`, or the lower bound on its number of colors if that lower bound is greater than `r`.

```
r = someNumberOfInterest

if minimalColoring(graph, r):
  doSomething()
else:
  ...
```

independent sets

`setUtil(graph)`

- Description:
  Computes the maximum independent set size of a graph. (This is an NP-hard problem and so finding an exact solution can take an extremely long time for large graphs)
- Inputs:
  - graph, a networkx graph
- Returns:
  The MIS number for *graph*.
- Example usage:

```
alpha = setUtil(graph)

if alpha > someNumber:
  doSomething()
```

`isIndependent(graph, nodeSet)`

- Description:
  Checks if a set of vertices is independent.
- Inputs:
  - graph, a networkx graph
  - nodeSet, a list object containing the graph nodes to check
- Returns:
  True if the set is independent, otherwise `False`
- Example usage:

```
...#(some graph is generated)
nodes = graph.someFunctionReturningNodes()
if isIndependent(graph, nodes):
  doSomething()
else:
  doSomethingElse()
```

`generateGXYYR(s)`

- Description:
  Generates a confusability graph G of X given Y, Yr, given an nxnxn matrix s.
- Inputs:
  - s, an nxnxn matrix
- Returns:
  - the 1-fold confusability graph GXYYR
- Example usage:

```
s = tables.getTableGenerator()
for s_i in s:
  gx = generateGXYYR(s_i)
  ...
```

generateGRK(g, s)

- Description:
  Generates the list of graphs G of R given K. Enumerates all possible maximum independent sets of g and constructs a graph for each one.
- Inputs:
  - g, a confusability graph G of X given Y, Yr
  - s, an nxnxn matrix
- Returns:
  A list object containing graphs G_{R|K}
- Example usage:

```
s = tables.getTableGenerator()
for s_i in s:
  gx = generateGXYYR(s_i)
  grk = generateGRK(gx, s_i)

  doSomethingWithGRK(grk)
  ...
```