# Technical Report

## Introduction

Given a graph G = (V, E), where V is the set of vertices (nodes) in G and E is the set of edges in G, the ***vertex coloring problem*** is to assign a color from the set K = {1, ..., *c*} to each *v* in V such that the endpoints of each edge are assigned different colors, using as few colors as possible.

This report:

* Describes an implementation of two 0-1 integer programming (IP) formulations of the vertex coloring problem
* Explores the effects that adding particular classes of facet-defining constraints to these formulations has on the solution times of these formulations.

### IP Formulations

The following sections describe the 0-1 IP formulations implemented and explored in this report.

#### "Assignment" formulation, with symmetry breaking

Mendez-Diaz and Zabala describe this straightforward formulation in "A Branch-and-Cut Algorithm for Graph Coloring".

Let variable *xik* = 1 if node i is assigned color *k*, 0 else, ∀ *i* ∈ V, ∀ *k* ∈ K. Let variable *wk* = 1 if color *k* is used on at least one node, 0 else, ∀ *k* ∈ K. Then the vertex coloring problem becomes:

Minimize Σ*k*∈K *wk*

Subject to: Σ*k*∈K *xik* = 1, ∀ *i* ∈ V

*xik* + *xjk* ≤ *wk*, ∀ (*i*, *j*) in E, ∀ *k* ∈ K

*xik* ∈ {0, 1}, ∀ *i* ∈ V, ∀ *k* ∈ K

*wk* ∈ {0, 1}, ∀ *k* ∈ K

These constraints ensure that, respectively: each node is assigned exactly one color, and endpoints of an edge cannot receive the same color.

The assignment formulation suffers from inherent symmetry: given a particular solution, one can obtain many equivalent solutions by performing a one-to-one switch of one color for another: e.g. those nodes colored with color 1 get color 2 instead, and vice versa. To eliminate symmetrical equivalent solutions from consideration, add the following constraints:

*wk* ≤ Σ*i*∈V *xik*, ∀ *k* ∈ K

*wk* ≥ *wk+1*, ∀ *k* ∈ K \ {*c*}

These constraints ensure that, respectively: a color is not considered used unless at least one node is marked with it, and a greater-numbered color is not used unless all the lesser-numbered colors are used.

##### Clique constraints

Mendez-Diaz and Zabala identified numerous categories of valid inequalities for the polytope that this assignment formulation represents, including the facet-defining ***clique inequalities***. Let Q be a maximal clique of G; then the following inequalities define facets:

Σ*i*∈Q *xik* ≤ *wk*, ∀ *k* ∈ K \ {c}

We can view such constraints as stronger statements about adjacent nodes not receiving the same color, summing all members of the clique at once rather than leaning on the pairwise sums from the original formulation.

#### "Representative" formulation

Since no adjacent nodes can receive the same color in a vertex coloring, all nodes in a feasible coloring that have received the same color are an independent set. Then a feasible coloring can be considered a partition of the graph's nodes into some number of independent sets, and an optimal coloring as such a partition of the nodes into a minimal number of independent sets. The representative formulation of Campelo, Correa, and Frota [] builds on the work of Mehrotra and Trick [] to model the vertex coloring problem using the notion of independent sets without using one variable per maximal independent set.

Let every node in the graph that is assigned the same color be considered members of a color class. Suppose every color class has exactly one node designated as the ***representative*** of that color class. Let N-(*i*) denote the *anti-neighborhood* of node *i*: {*v* ∈ V : (*i*, *v*) ∉ E}, and let N-[*i*] denote N-(*i*) ∪ {*i*}. Let G[S] be the subgraph induced by some S ⊂ V, and let E[S] be the edge set of G[S]. Let variable *xij* = 1 iff node *i* represents the color of node *j*, 0 else. Then the vertex coloring problem becomes:

Minimize Σ*i*∈*V* *xii*

Subject to: Σ*j*∈*N-[i]* *xij* ≥ 1, ∀ *i* ∈ V

*xij* + *xik* <= *xii*, ∀ *i* ∈ V, ∀ (*j*, *k*) ∈ E[N-[*u*]]

These constraints ensure that, respectively: either a node represents its color class or some node not adjacent to it does, and that adjacent nodes cannot share a representative (and hence a color class).

##### Clique constraints

Campelo et al. identified numerous categories of valid inequalities for the polytope that this representative formulation describes, including the facet-defining ***clique inequalities***. Let *i* ∈ V and let Q ⊆ N-(i) so that G[Q] is a maximal clique of G[N-(i)]; then the following inequality defines a facet:

Σ*j*∈Q *xij* ≤ *xii*, ∀ *k* ∈ K \ {c}

We can view such constraints as stronger statements about adjacent nodes not sharing a color representative, summing all members of the clique at once rather than leaning on the pairwise sums from the original formulation.

## Assumptions

We assume that there are as many colors available for use in the coloring of a graph as there are vertices in the graph; that is, *k* = ⏐*V*⏐.

We assume, in the case of the representative formulation, that no node is universal (i.e. its anti-neighborhood is the empty set) and that no node's anti-neighborhood has isolated nodes (with no edges incident). Note that the assignment formulation has no such restriction.

## Methods and Analysis

To assess the effects that clique inequalities have on the solutions of the aforementioned 0-1 IP formulations of the vertex coloring problem, we implemented a computer program using:

* Python[[1]](#footnote-1) 2.7.14
* IBM ILOG CPLEX Optimization Studio[[2]](#footnote-2) 12.8, and its Python 2.7 bindings
* NetworkX[[3]](#footnote-3) 2.1, for modeling graph structures and performing algorithms on them
* pytest[[4]](#footnote-4) 3.5.0, for writing tests
* matplotlib[[5]](#footnote-5) 2.2.2, for plotting coloring solutions

The following sections describe how to install the program and its dependencies, how to run the program, the structure and purpose of components of the program, and gives some results of the program's execution on sample graphs. We assume that the reader is familiar with executing programs from a Unix shell or Windows command prompt, and with typical practices for installing Python libraries and any native-code dependencies.

### Installation

We assume Python 2.7.x is already installed with the appropriate CPLEX bindings. On Mac OS X, we were able to install software required for matplotlib using the Homebrew[[6]](#footnote-6) software and following shell commands ($ is the shell prompt):

$ brew install freetype

$ brew install pkg-config

$ brew install libpng

The following assumes that you have unpacked the source distribution accompanying this report and are in a command prompt whose working directory is the root of the distribution (the directory that contains these instructions you are reading).

To install the program and its other dependencies, you may be able to use the included setup.pyscript. From a shell prompt, type:

$ python setup.py install –-user

This should retrieve and install NetworkX, pytest, and matplotlib, and any transitive dependencies. If your Python installation includes setuptools but you do not wish to run setup.py, you may install the dependencies separately using pip:

$ python -m pip install –-user network pytest matplotlib

### Running the Automated Tests

After having installed the software as above, from a command prompt type:

$ python -m pytest tests

This will execute a number of unit tests against some of the program's components.

### Running the Program

The main routine for the program lives in solver.py. To see the command line options available, from a command prompt type:

### $ python solver.py -h

You should see a help screen similar to the following:

usage: solver.py [-h] -g GRAPH [-f {rep,assign}] [-d PROBLEM\_FILE\_DIR]

[-s {ip,lr}] [-p] [-v] [-r {warm,cold}]

optional arguments:

-h, --help show this help message and exit

-g GRAPH, --graph GRAPH

Path to graph description for graph to color (DIMACS

format) (default: None)

-f {rep,assign}, --formulation {rep,assign}

Desired formulation of vertex coloring (default:

assign)

-d PROBLEM\_FILE\_DIR, --problem-file-dir PROBLEM\_FILE\_DIR

Path to write CPLEX LP file for problem to (default:

.)

-s {ip,lr}, --solve-as {ip,lr}

Whether to solve as IP, or LR with cuts (default: ip)

-p, --plot-if-integer

Plot final solution if it is integer (default: False)

-v, --verbose Show values of variables in intermediate solutions

(default: False)

-r {warm,cold}, --restart-mode {warm,cold}

Warm restart allows reuse of previous LR solutions,

cold starts from scratch (default: warm)

The program expects to read in a file that represents the graph to color, specified by the -g option. The file is expected to be in the DIMACS format[[7]](#footnote-7). Several examples, some from previous DIMACS challenges[[8]](#footnote-8) and some created by hand, are included in directory tests/data.

Choose the "representative" or "assignment" 0-1 IP formulation of the vertex coloring problem using the -f option. If not specified, the "assignment" formulation is used.

Prior to solving either the initial formulation or a follow-on formulation with cuts added that a previous solution violates, the program emits a representation of the current formulation to a file named vertexcoloring.[graph].[n].lp, where [graph] is the base name of the graph being colored and [n] is the "iteration" number (0 for original formulation, 1 for the original formulation with first round of violated clique cuts added, and so forth). These files and in CPLEX LP file format. By default, these files are written to the current working directory; use the -d option to override this default.

The -s option controls whether the problem will be solved as a 0-1 integer program, or as a successive set of linear relaxations (with variables bounded between 0 and 1, inclusive) with violated clique cuts added at every LR solution.

The -p option, if specified, will have the program plot an optimal integer coloring if such is found.

The -v option will have the program print the values of variables for individual linear relaxation solutions. If not specified, you will only see the solution values printed when no more clique cuts are violated.

The -r option controls whether, after a linear relaxation solution is found, the optimal basis can be re-used after adding violated clique cuts ("warm" restart) or whether to discard that solution and re-solve the problem fresh with the clique cuts added ("cold" restart). "Warm" restart is the default.

#### Examples

To color the graph in file tests/data/50\_0.2.col, using the representative formulation as a 0-1 integer program, invoke the program like so:

$ python solver.py -g tests/data/50\_0.2.col -f rep -p

To color the graph in file tests/data/7\_with\_k5.col, using the assignment formulation and successive linear relaxation solutions, invoke the program like so:

$ python solver.py -g tests/data/7\_with\_k5.col -p -s lr

### Generating Graphs

The source distribution also includes a Python program that will generate random graphs: generate\_random\_graph.py. To see the command line options available, from a command prompt type:

### $ python generate\_random\_graph.py -h

You should see a help screen similar to the following:

usage: generate\_random\_graph.py [-h] -n NUMBER\_OF\_NODES [-p {0...1}] [-s SEED]

optional arguments:

-h, --help show this help message and exit

-n NUMBER\_OF\_NODES, --number-of-nodes NUMBER\_OF\_NODES

Desired number of nodes in the graph (default: None)

-p {0...1}, --probability-of-edge-creation {0...1}

Probability of an edge between any two nodes (default:

0.5)

-s SEED, --seed SEED Seed for the random number generator (default: None)

This program emits a graph in the aforementioned DIMACS format to the standard output.

#### Examples

To generate a graph with twenty nodes, with likelihood ½ that an edge between two nodes gets generated:

$ python generate\_random\_graph.py -n 20 -p 0.5

To generate a complete graph of five nodes:  
  
 $ python generate\_random\_graph.py -n 5 -p 1

### Program Structure

#### Class VertexColoringProblem

This is an abstract class that represents a formulation of the vertex coloring problem. Instances of this class encapsulate a Cplex object from the CPLEX Python API and a marker solve\_as that is assumed to be either the string 'ip' (to solve the problem as a 0-1 integer program) or the string 'lr' (to solve the problem as a linear relaxation of a 0-1 integer program), and the class's methods manipulate the Cplex instance in various ways:

* set\_sense\_minimize() instructs CPLEX to treat the problem as a minimization problem.
* set\_objective(coefficients, var\_names) instructs CPLEX to add variables to the problem and encode an objective function in terms of those variables. If the problem is to be solved as a 0-1 integer program, it tells CPLEX to use binary variables; if as a linear relaxation, it tells CPLEX to use real-valued variables with an upper bound of 1 and the implied lower bound of 0.
* add\_constraints(constraints) instructs CPLEX to add constraints to the problem. See Class Constraint below.
* suppress\_output() instructs CPLEX to shut off its log, error, warning, and results streams. This is useful during unit testing to de-clutter test output.
* emit\_to(path) instructs CPLEX to write a representation of the problem to the given file path, in CPLEX LP file format.
* cplex\_solve() asks CPLEX to solve the problem, and returns a tuple (solution, time), where solution is the CPLEX Python API representation of the solution and time is the difference between calls to the CPLEX get\_dettime() API call before and after solving, in deterministic ticks.
* clique\_cuts() is an abstract method that subclasses implement to generate clique cuts specific to their formulations. See Class Cut below.
* solve() is an abstract method that subclasses implement to perform formulation-specific manipulations for a solution. Typically, a subclass implementation will invoke cplex\_solve(), and pass the results along to a formulation-specific implementation of class VertexColoringSolution (see Class VertexColoringSolution below).
* all\_vars() is an abstract method that subclasses implement to give a list of all the variables in the current representation of the problem.

#### Class VertexColoringSolution

This is an abstract class that represents a solution to a vertex coloring problem. Instances of this class retain a solution object from the CPLEX Python API, the VertexColoringProblem that produced that solution, and the running time (in deterministic ticks) it took to obtain that solution. The class's methods manipulate the CPLEX solution instance in various ways:

* objective\_value() gives the value of the objective function for the solution.
* values() gives a dictionary whose keys are the names of variables in the problem formulation, and whose values are the values of those variables in the solution.
* value\_of(variable\_names) takes a variable-length argument list of variables names in the problem formulation and returns a list of equal length whose members are the corresponding values of those variables in the solution.
* show(to) prints the values of the variables in the solution to the output stream named to. to is the standard output if not specified.
* is\_integer() tells whether the solution's values are integer-valued. A value is considered integral "enough" if it passes the check of function isclose() in module vertexcoloring.is\_close – i.e., if the value is within 10-4 of the nearest integer.
* used\_colors() is an abstract method whose implementations give a list of those colors that were used in the solution. This result may not be useful if the solution is not integer.
* colors\_by\_node() is an abstract method whose implementations give a dictionary, whose keys are node names and whose values are the color assigned to the respective nodes. This result may not be useful if the solution is not integer.
* nodes\_by\_color() gives a dictionary whose keys are colors and whose values are lists of nodes that have been assigned the respective color. This result may not be useful if the solution is not integer.

#### Class Constraint

This is a purely abstract class that represents a linear constraint in a vertex coloring problem. Implementations of Its methods are to give results that a VertexColoringProblem feeds to CPLEX on a call to add\_constraints():

* Implementations of name() give an identifiable name for the constraint.
* Implementations of terms() give a list of two parallel lists of equal length: the first consisting of variables names in the constraint, and the second consisting of corresponding coefficients in the constraint, taken as a sum of linear terms.
* Implementations of rhs() give a constant value for the right-hand side of the constraint.
* Implementations of sense() give a value that CPLEX recognizes as greater-than-or-equal-to ('G'), equal-to ('E'), or less-than-or-equal-to ('L').

#### Class Cut

This is an abstract subclass of Constraint that adds one method:

* Implementations of allows(solution) tell whether the given VertexColoringSolution satisfies the constraint. This is used during the solution process, when deciding what clique cuts to add to the current formulation.

#### Class colorassignment.Problem

Instances of this class represent assignment formulations of the vertex coloring problem. Given a NetworkX representation of a graph, on construction instances retain the graph, views on the graph's nodes and edges, and a set of colors taken to be the set of nodes. It then immediately calls method init\_cplex() to build up the problem using the CPLEX Python API:

* set\_sense\_minimize()
* set\_objective() using ones for the color-used variables, zeros for the node-gets-color variables
* add\_constraints() using:
  + One NodeGettingColorConstraint for each node in the graph
  + One AdjacentNodeColorConstraint for each edge-color pair
  + One ColorUsedOnlyIfMarksNodeConstraint for each color
  + One UseLowerNumberedColorFirstConstraint for each color but the last

Method solve() calls cplex\_solve() and wraps the result in a colorassignment.Solution.

Method clique\_cuts() yields one clique cut per non-trivial (more than two nodes) maximal clique found by NetworkX's find\_cliques() algorithm. See Class colorassignment.CliqueCut below.

The remaining methods are for creating names for the variables of the problem.

### Results

### Analysis

1. <https://www.python.org> [↑](#footnote-ref-1)
2. <https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/Optimization_Studio/topics/COS_relnotes_intro.html> [↑](#footnote-ref-2)
3. <http://networkx.github.io> [↑](#footnote-ref-3)
4. <https://docs.pytest.org/en/latest/> [↑](#footnote-ref-4)
5. <https://matplotlib.org> [↑](#footnote-ref-5)
6. <https://brew.sh> [↑](#footnote-ref-6)
7. <http://prolland.free.fr/works/research/dsat/dimacs.html> [↑](#footnote-ref-7)
8. <http://mat.gsia.cmu.edu/COLOR/instances.html> [↑](#footnote-ref-8)