Paul Holser – IMSE 884 – Spring 2018 – Term Project

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

Méndez-Díaz and Zabala describe this straightforward formulation[i].

Let variable $x_{ik} = 1$ if node i is assigned color k, 0 else, $\forall\ i \in V$, $\forall\ k \in K$. Let variable $w_k = 1$ if color k is used on at least one node, 0 else, $\forall\ k \in K$. Then the vertex coloring problem becomes:

$$\begin{gathered}\text{Minimize } \textstyle\sum_{k\in K} w_k \\ \text{Subject to: } \textstyle\sum_{k\in K} x_{ik} = 1,\ \ \forall\ i \in V \\ x_{ik} + x_{jk} \le w_k,\ \forall\ (i, j) \text{ in } E,\ \forall\ k \in K \\ x_{ik} \in \{0, 1\},\ \forall\ i \in V,\ \forall\ k \in K \\ w_k \in \{0, 1\},\ \forall\ k \in K\end{gathered}$$

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:

$$w_k \le \textstyle\sum_{i\in V} x_{ik},\ \forall\ k \in K$$

$$w_k \geq w_{k+1},\ \forall\ k \in K \setminus \{c\}$$

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

Méndez-Díaz 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:

$$\Sigma_{i \in Q}\, x_{ik} \leq w_k,\ \ \forall\ k \in K \setminus \{c\}$$

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 receive 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[ii] models the vertex coloring problem using the notion of independent sets without resorting to 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 designates exactly one node as its ***representative***. Let $N_{-}(i)$ denote the *anti-neighborhood* of node i: $\{v \in V : (i, v) \notin E\}$, and let $N_{-}[i]$ denote $N_{-}(i) \cup \{i\}$. Let G[S] be the subgraph induced by some $S \subset V$. Let E[S] be the edge set of G[S]. Let variable $x_{ij} = 1$ iff node i represents the color of node j, 0 else. Then the vertex coloring problem becomes:

$$\text{Minimize } \Sigma_{i \in V}\, x_{ii}$$
$$\text{Subject to: } \Sigma_{j \in N_{-}[i]}\, x_{ij} \geq 1,\ \forall\ i \in V$$
$$x_{ij} + x_{ik} <= x_{ii},\ \forall\ i \in V,\ \forall\ (j, k) \in E[N_{-}[u]]$$

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).

This formulation suffers from classes of symmetrical solutions as well: identically-colored solutions with different color class representatives are equivalent. For purposes of this report, we make no attempt to eliminate such symmetry from the formulation.

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 $i \in V$. Let $Q \subseteq N_{-}(i)$ so that $G[Q]$ is a maximal clique of $G[N_{-}(i)]$. Then the following inequality defines a facet:

$$\sum_{j \in Q} x_{ij} \leq x_{ii}, \ \ \forall\, k \in K \setminus \{c\}$$

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, $c = |V|$. We assume that the graph is undirected.

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 of adding clique inequalities on the solution times of the aforementioned 0-1 IP formulations of the vertex coloring problem, we implemented a computer program using:

- Python[iii] 2.7.14
- IBM ILOG CPLEX Optimization Studio[iv] 12.8, and its Python 2.7 bindings
- NetworkX[v] 2.1, for modeling graph structures and performing algorithms on them
- pytest[vi] 3.5.0, for writing tests
- matplotlib[vii] 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[viii] 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.py` script. 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[ix]. Several examples, some from previous DIMACS challenges[x] 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 an 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.

Class `colorassignment.NodeGettingColorConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $\Sigma_{k \in K} x_{ik} = 1$ for a given node i.

Class `colorassignment.AdjacentNodeColorConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $x_{ik} + x_{jk} \leq w_k$, for a given graph edge (i, j) and color k. To satisfy CPLEX, the constraint is rephrased as $x_{ik} + x_{jk} - w_k \leq 0$.

Class `colorassignment.ColorUsedOnlyIfMarksNodeConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $w_k \leq \Sigma_{i \in V} x_{ik}$, for a given color k. To satisfy CPLEX, the constraint is rephrased as $(\Sigma_{i \in V} x_{ik}) - w_k \geq 0$.

Class `colorassignment.UseLowerNumberedColorFirstConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $w_k \geq w_{k+1}$, for a given color k. To satisfy CPLEX, the constraint is rephrased as $w_k - w_{k+1} \geq 0$.

Class `colorassignment.CliqueCut`

Instances of this subclass of `Cut` model clique cuts for the assignment formulation. These are inequalities of the form $\Sigma_{i \in Q} x_{ik} \leq w_k$, for a given clique Q and color k. To satisfy CPLEX, the cut is rephrased as $\Sigma_{i \in Q} x_{ik} - w_k \leq 0$.

Class `representative.Problem`

Instances of this class describe representative formulations of the vertex coloring problem. Given a NetworkX representation of a graph, on construction instances retain the graph, its complement, and views on the graph's 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 "node represents own color class" variables, zeros for every other variable
- `add_constraints()` using:
  - One `RepresentativeConstraint` for each node in the graph
  - One `DistinctRepresentativesForNeighborsConstraint` for each node, and each of the edges in the node's anti-neighborhood

Method `solve()` calls `cplex_solve()` and wraps the result in a `representative.Solution`.

Class `representative.RepresentativeConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $\Sigma_{j \in N\text{-}[i]} x_{ij} \geq 1$, for a given node *i*.

Class `representative.DistinctRepresentativesForNeighborsConstraint`

Instances of this subclass of `Constraint` model assignment formulation constraints of the form $x_{ij} + x_{ik} <= x_{ii}$, for a given node *i* and edge (*j*, *k*) such that *j* and *k* are in the anti-neighborhood of *i*. To satisfy CPLEX, the constraint is rephrased as $x_{ij} + x_{ik} - x_{ii} <= 0$.

Class `representative.CliqueCut`

Instances of this subclass of `Cut` model clique cuts for the representative formulation. These are inequalities of the form $\Sigma_{j \in Q} x_{ij} \leq x_{ii}$, for a given clique Q and color *k*. To satisfy CPLEX, the cut is rephrased as $(\Sigma_{i \in Q} x_{ik}) - x_{ii} \leq 0$.

Module `dimacs`

This module contains helper classes `Parser` and `Formatter` for parsing DIMACS graph input and emitting graphs to DIMACS format.

Note: Some files in the DIMACS test set seemed to represent undirected graphs using edges in both directions: i.e. an edge (*i*, *j*) would have a line for (*i*, *j*) and a line for (*j*, *i*). For this reason, a Parser sanity-checks a file against the number of expected edges and the number of expected edges divided by 2.

Note: Nodes are assumed to be numbered in sequence. If for some reason a graph is given that has isolated nodes (not implied by any edge in the graph file), their numbers are assumed to fall within the range [min(node number), min(node number) + number of expected nodes].

solver.py

This module contains the main routine, and a function `plot()` that will plot an integer coloring using matplotlib if such is found.

After parsing command line arguments as described above in Running the Program, the main routine proceeds as follows:

- Read the graph to be colored from a file named in the `-g` option
- While we don't have a solution and there are no more violated cuts to apply:
    - If there is no problem instance yet created, make an instance of `VertexColoringProblem` corresponding to the formulation specified by

the `-f` option, to be solved as either a 0-1 IP or a linear relaxation thereof as specific by the `-s` option. Retain a dictionary of candidate clique cuts to apply, keyed by a unique ID, by asking the problem for `clique_cuts()`.
   - Otherwise:
      - If "warm" restart mode is specified via the `-r` option, and we've solved the current problem before, add any newly violated clique cuts to the problem via `add_constraints()`. Otherwise, make a fresh instance of the problem, and add any previously collected clique cuts to it via `add_constraints()`.
   - Suppress CPLEX output via `suppress_output()`. Write the next rendition of the problem (including added cuts) to a file via `emit_to()`.
   - Ask CPLEX to solve the problem via `solve()`. Show the time take to solve, and the objective value. If verbosity was requested via the `-v` option, show the values of all the variables in the current solution.
   - Decide which of the as-yet-unapplied clique cuts will cut off the current solution by testing each candidate cut's `allows()` method. Retain the violators, if any, for the next iteration, and exclude these as candidates for subsequent consideration.
- When there are no more violated cuts, show the current solution's objective value and the values of all the variables.
- If the solution is integer and plotting was requested via the `-p` option, call `plot()` to plot the solution.

Results

To assess the effects of adding violated clique cuts to vertex coloring problem formulations, we performed the following steps:

- Generated the following test graphs:
   - Four random graphs with 10 nodes each, with probability of edge between two nodes = 0.2, 0.4, 0.6, and 0.8, respectively
   - Four random graphs with 20 nodes each, with probability of edge between two nodes = 0.2, 0.4, 0.6, and 0.8, respectively
   - Four random graphs with 30 nodes each, with probability of edge between two nodes = 0.2, 0.4, 0.6, and 0.8, respectively
   - Two random graphs with 50 nodes each, with probability of edge between two nodes = 0.2 and 0.4, respectively

  These graphs live in directory `tests/data/benchmark` in the source distribution, and are named `[#nodes]_[probability].col` according to their number of nodes and edge probability.

- Executed `solver.py` against each of these graphs, with both the assignment and representative formulations, solving the problems as 0-1 integer programs. Table 1 shows the execution times of the solver in deterministic ticks.

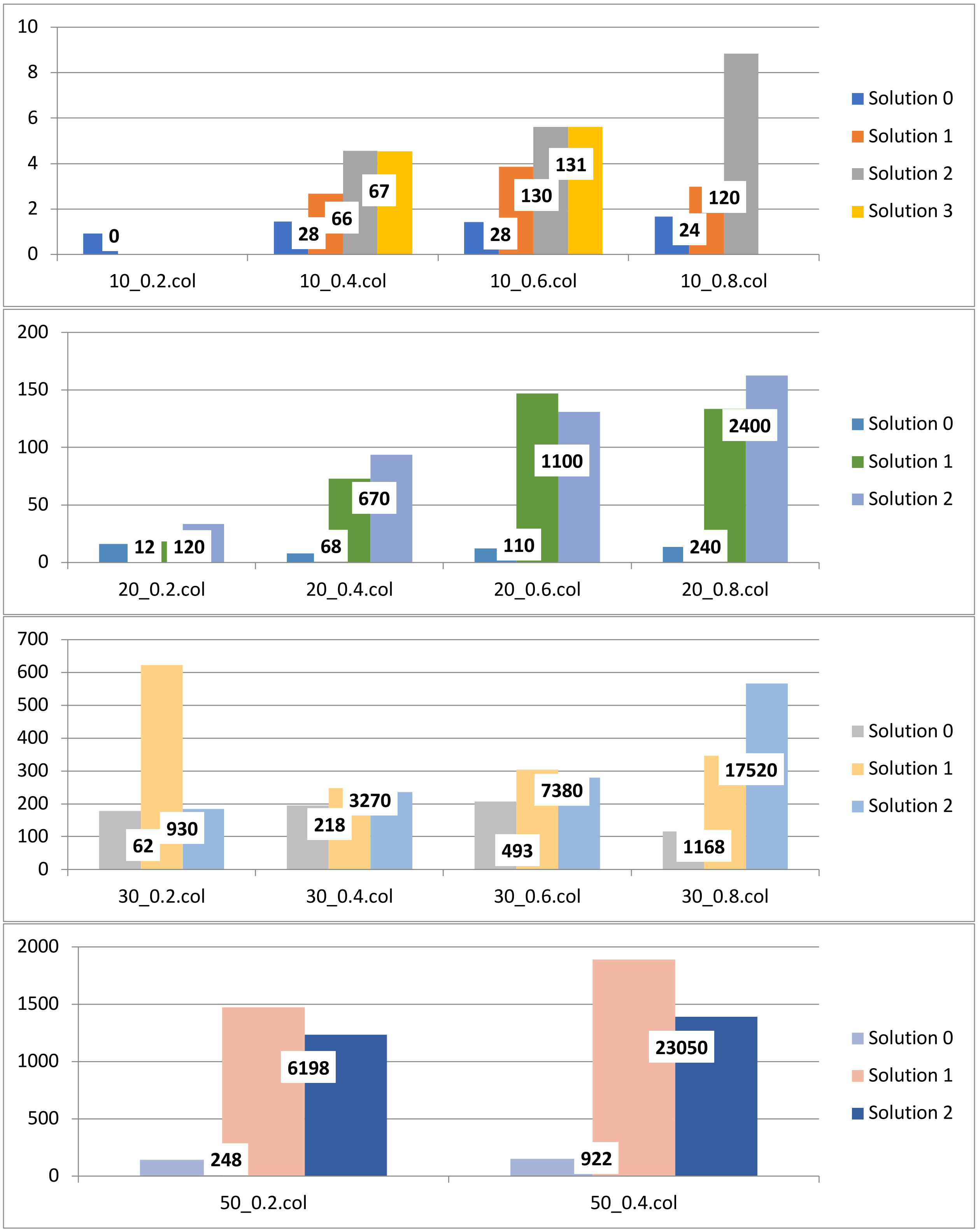
Table 1: Solution times for vertex coloring IP formulations

Problem	Assignment IP solution time	Representative IP solution time
10_0.2.col	1.288811684	0.784894943
10_0.4.col	3.146530151	0.485424042
10_0.6.col	6.014997482	0.464651108
10_0.8.col	4.215065002	0.244623184
20_0.2.col	19.07796478	6.652316093
20_0.4.col	33.80988884	4.448896408
20_0.6.col	66.86366081	2.760329247
20_0.8.col	147.4937983	0.554646492
30_0.2.col	111.5356503	76.04312897
30_0.4.col	633.3190336	198.2262821
30_0.6.col	1655.356151	125.4106655
30_0.8.col	1121.25187	3.804841995
50_0.2.col	1585.080634	57992.71706
50_0.4.col	6500.987433	6094.631468

- Executed `solver.py` against each of these graphs, with the assignment formulation, solving the problems as linear relaxations of 0-1 integer programs, in "cold restart" mode. We solve the initial linear relaxations; find any clique cuts that the current linear relaxation solution violates, then reload the initial problem with all violated cuts found so far added; and repeat the process until no more clique cuts are violated. This may or may not result in an integer solution. Figure 1 shows the execution times for each iteration for each linear relaxation solution in deterministic ticks.

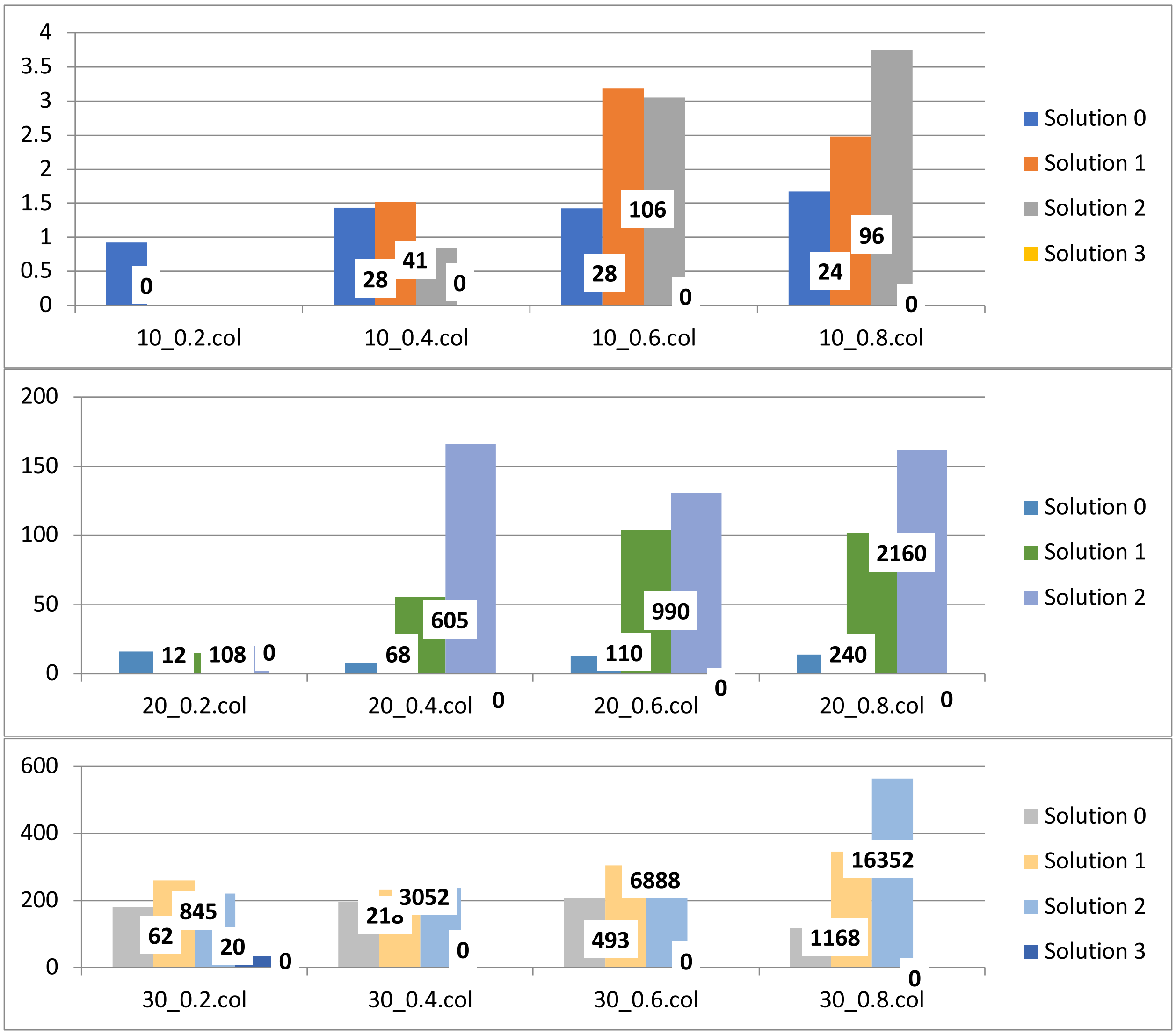
  Note: In this and follow-on bar graphs, the labels that straddle neighboring bars on the graph indicate the number of violated clique cuts added after solution i *but before solution (*i + 1*).*

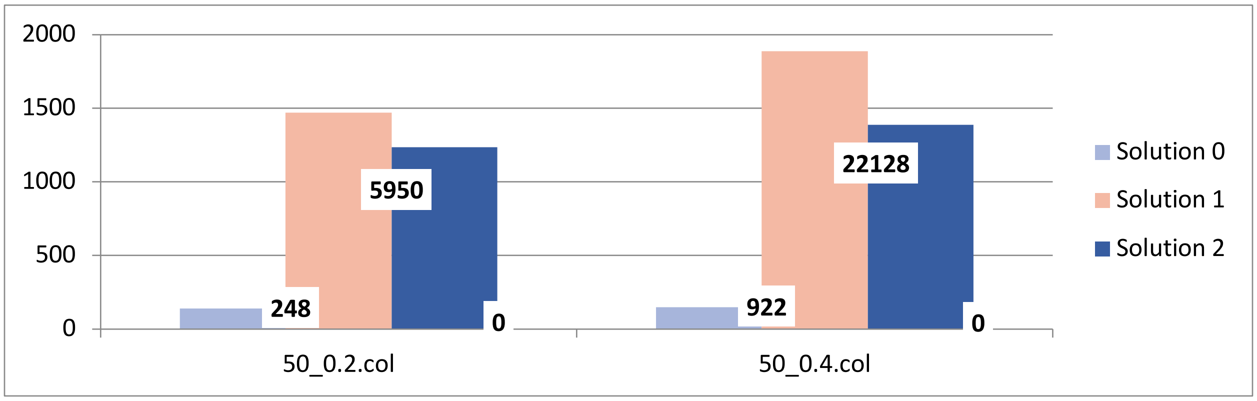
Figure 1: Solution times for assignment formulation LR, cold restart



- Executed `solver.py` against each of these graphs, with the assignment formulation, solving the problems as linear relaxations of 0-1 integer programs, in "warm restart" mode. We solve the initial linear relaxations; find any clique cuts that the current linear relaxation solution violates, then add them to the current problem without reloading; and repeat the process until no more clique cuts are violated. This may or may not result in an integer solution. Figure 2 shows the execution times for each iteration for each linear relaxation solution in deterministic ticks.

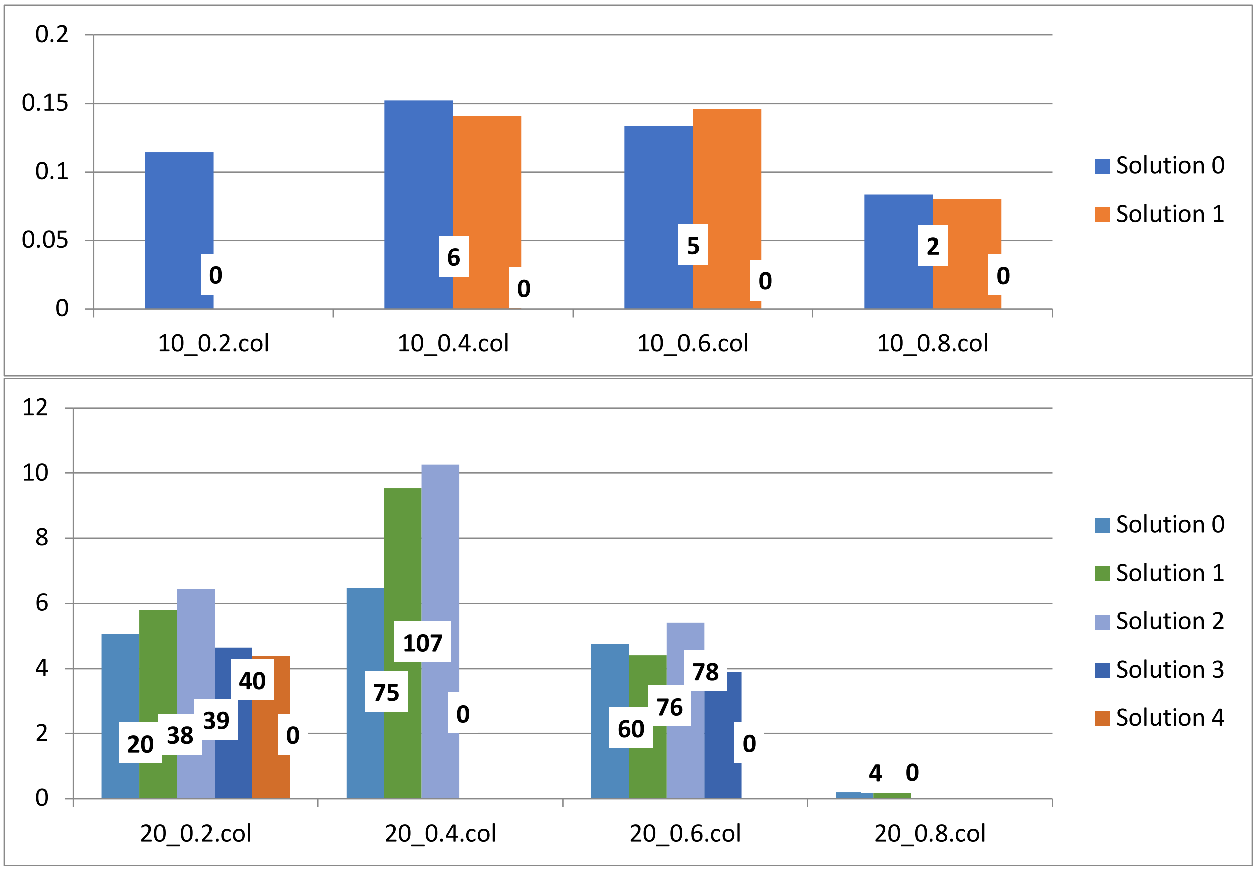
Figure 2: Solution times for assignment formulation LR, warm restart

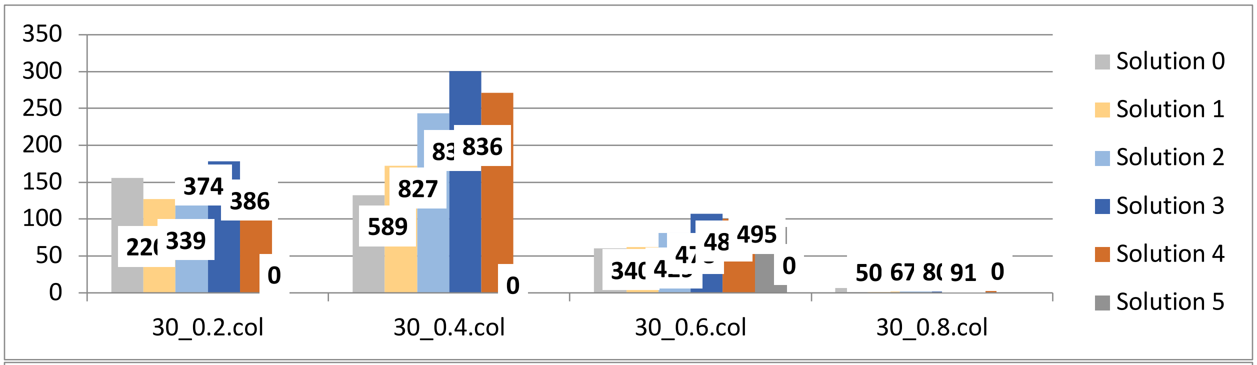


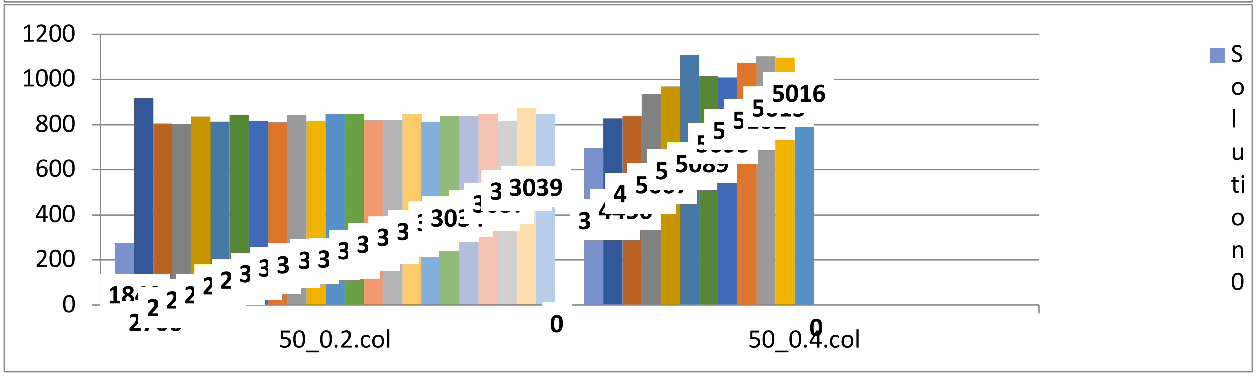


- Executed `solver.py` against each of these graphs, with the representative formulation, solving the problems as linear relaxations of 0-1 integer programs, in "cold restart" mode. Figure 3 shows the execution times for each iteration for each linear relaxation solution in deterministic ticks.

Figure 3: Solution times for representative formulation LR, cold restart

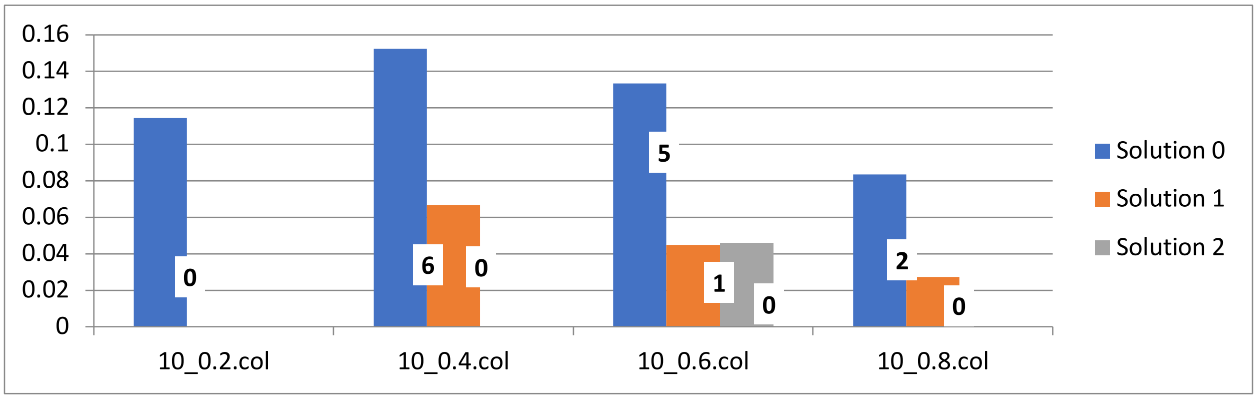


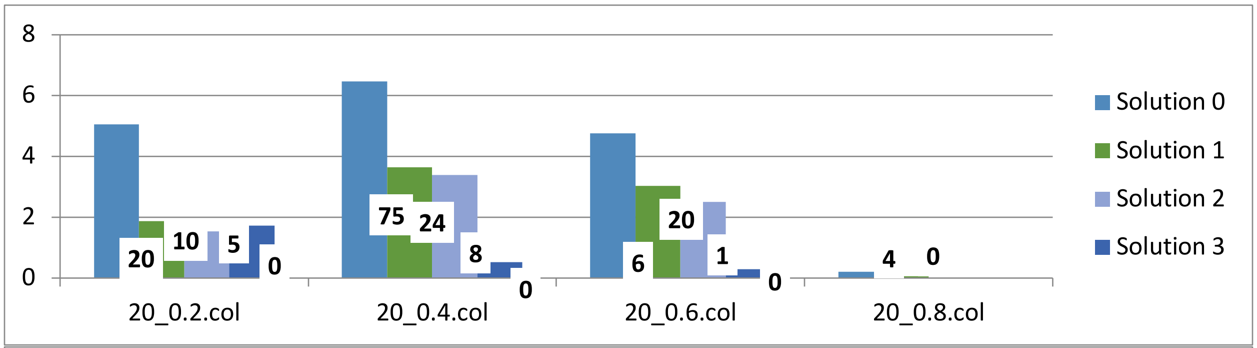


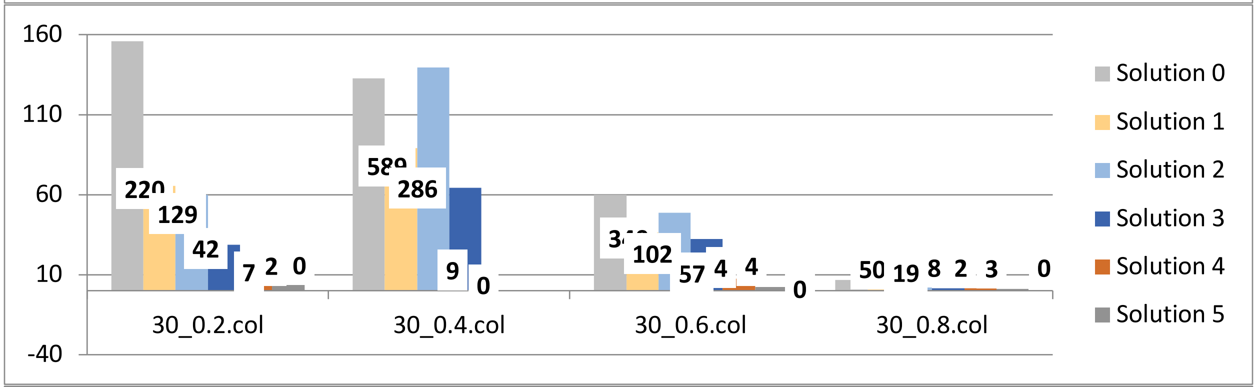


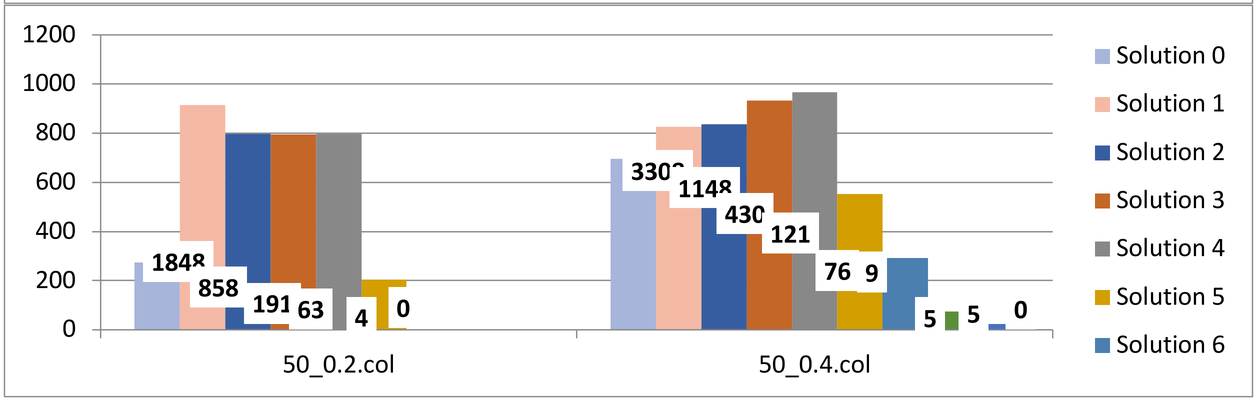
- Executed `solver.py` against each of these graphs, with the representative formulation, solving the problems as linear relaxations of 0-1 integer programs, in "warm restart" mode. shows the execution times for each iteration for each linear relaxation solution in deterministic ticks.

Figure 4: Solution times for representative formulation LR, warm restart









Analysis

We note that no amount of additional hand-rolled clique cuts to linear relaxations seems to improve much on the solution times that CPLEX exhibits in its IP solving mode. Very likely, any preprocessing CPLEX does to reduce the size of the problem and the vast array of generated cuts at its disposal allow for branch-and-bound to reach an optimum integer solution far more quickly.

In general, the representative formulation outperforms the assignment formulation on the graphs tested here. Further, successive adding of violated clique cuts does not appear to decrease solution times for the assignment formulation, whereas for the representative formulation we see decreased solution times after such successive applications. It may well be that the increased dimension of the assignment formulation compared to the representative formulation means that the additional cuts do more harm than good; after each linear relaxation we find more and more assignment clique cuts violated, whereas with the representative formulation, either the increase in violations is less pronounced or in fact is a decrease.

Directions for Further Exploration

There are other classes of valid inequalities for both the assignment and representative formulations of the vertex coloring problem. The abstractions we have built atop CPLEX to represent constraints and cuts should allow for straightforward integration of different kinds of valid inequalities so that we can assess their effects on solution times.

It would be interesting to see how certain bits of preprocessing would improve solution times. For example, if we were to find a large clique Q in a graph, in the assignment formulation we could automatically choose $|Q|$ colors for the nodes in that clique, and remove the nodes from the problem entirely.

We purposely chose sample graphs that were of reasonable size – perhaps too modestly small for many real-world applications. We should try out larger graphs and explore branch-and-cut or column-generation approaches for applying integer programming techniques to vertex coloring problems of greater magnitude.

[i] Méndez-Díaz, Isabel & Zabala, Paula. (2006). A Branch-and-Cut algorithm for Graph Coloring. Discrete Appl. Math.. 154. 826-847. 10.1016/j.dam.2005.05.022.
[ii] Campelo, Manoel & Correa, Ricardo & Frota, Yuri. (2004). Cliques, holes and the vertex coloring polytope. Information Processing Letters. 89. 159-164. 10.1016/j.ipl.2003.11.005.
[iii] https://www.python.org
[iv]https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/Optimization_Studio/topics/COS_relnotes_intro.html
[v] http://networkx.github.io
[vi] https://docs.pytest.org/en/latest/
[vii] https://matplotlib.org
[viii] https://brew.sh
[ix] http://prolland.free.fr/works/research/dsat/dimacs.html
[x] http://mat.gsia.cmu.edu/COLOR/instances.html