Assignment 4: Graph Coloring
CS 730/830, Fall 2015
Electronic submission due at **11:30pm on Wed, Sept 30**
Hardcopy submission due at **start of class on Thu, Oct 1**

Overview

You will write a program to color a given graph using a given number of colors such that no two adjacent vertices have the same color or prove that no such coloring is possible. This problem is fundamentally similar to scheduling (edges represent conflicts between jobs, colors represent time slots) and to register assignment in compilers (edges represent overlap between lifespans of variables and colors represent registers). Your program should treat the problem as a CSP and search the tree of possible assignments.

Input

Your program will read a graph from standard input and take an algorithm name (`dfs`, `fc`, or `mcv`) and a number of colors (integer > 0) as its command-line arguments. `dfs` should be basic depth-first search, backtracking as soon as a constraint violation is detected. `fc` should augment this with forward checking to reduce domains, and `mcv` will augment this to branch on a variable chosen randomly among those with the smallest domains ('most constrained variable'). You do not need to implement full arc consistency checking.

We will use a subset of the standard DIMACS graph format:

```
c This is a tiny example.
p edge 4 3
e 1 2
e 1 3
e 1 4
```

Our format is:

c Comments lines start with `c`.

p Before any edges comes a line starting with `p edge`. The rest of the line will be two numbers: the number of vertices and the number of edges.

e The rest of a line starting with `e` will contain two numbers, representing the two vertices that the edge connects. Vertices are numbered starting with 1. The symmetric edge (listing the same vertices in the opposite order) will not be listed in the file—the graph is undirected.

You should tolerate blank lines in the input.

Output

We will use the standard DIMACS color format:

```
s col 3
l 1 1
l 2 2
l 3 2
l 4 3
```

The format is:

s After `s col` should come the number of colors used.

l After `l` comes a vertex and a color (starting from 1).

If the graph has no k-coloring for the given k, instead print `No solution.` on its own line.

Your output should end with the number of branching nodes explored during the search:

```
345 branching nodes explored.
```

Execution

Please use `make.sh` and `run.sh` scripts as usual.
We supply:

`*.col` a few example benchmark problems.

`make-graph` takes two command line arguments: the number of vertices and the probability that two vertices are connected. Someday it might also take a flag `-reduce` k that takes a parameter k and preprocesses the graph to make it smaller (but doesn't change its k-colorability).

`color-reference` a sample solution. Also available as a jar file.

`color-validator` runs your program and validates its output. For example:

```
color-validator ./run.sh fc 5 < tiny-1.col
```

will run `./run.sh` with arguments `fc 5` on `tiny-1.col`. The validator expects a graph on standard input and will pass it to your program. A one minute timeout is hardcoded into the validator.

Submission

Electronically submit your source code using the instructions on the course web page.
Submit a brief write-up in class answering the following questions:

1. What is the size of the state space for this problem?
2. Describe any implementation choices you made that you felt were important. Mention anything else that we should know when evaluating your program.
3. What's the average speed-up you get for `fc` over `dfs`? For `mcv` over `fc`?
4. What suggestions do you have for improving this assignment in the future?

Graduate Extensions

Graduates students don't have to do anything extra for this assignment!

Evaluation

Grading is done with an automated script, so be sure you have one algorithm working before starting the next, so that we can give you some credit even if you don't implement both.
Rough guide to grading:

0 nothing

1 write up is correct but no code works

3 write up good, `dfs` works.

8 `fc` works too!

10 Everything runs smoothly and correctly. The implementation roughly on par with the reference solution even for large problems. Write-up is clear and convincing, with no errors.

Design Suggestions

Remember what you remove from domains during forward checking so that you can put it back when you backtrack.
You only need to check constraints related to the variable that just changed.