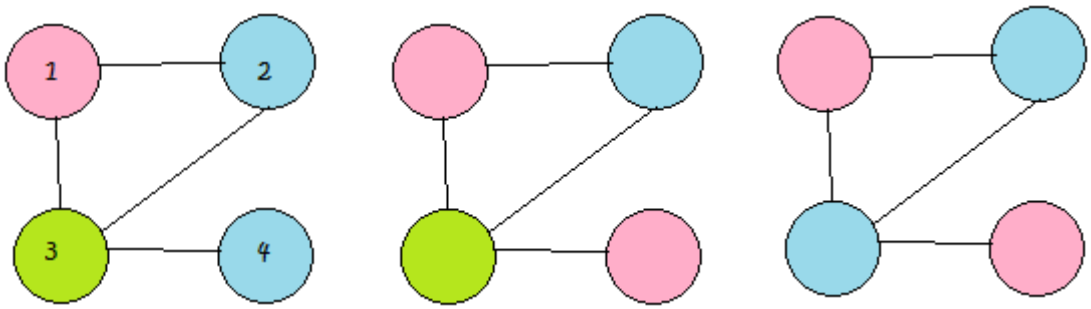
Local Search

Your search space is really large? Dynamic programming failed? Greedy solutions aren't good enough? Local search always works! It's almost like this magic blackbox that takes even the most primitive moves and gives out a pretty good solution *eventually*. 'Eventually' is the key word—while these searches usually converge to nice solutions, they can take a LOT of time depending on the parameters of your search. We'll look into those parameters in this lab.

Today's local search problem is the age-old toy problem—Graph coloring. The objective is to generate a coloring for the graph such that no two nodes connected by an edge share the same color. The task is to find a legal coloring allowing as few colors as possible.



(2 legal colorings using 3 colors, and 1 illegal coloring using 2 colors—the two blue nodes share an edge)

You will use simulated annealing to solve this problem. While the annealing algorithms itself is written for you, you are required to fill in important methods that will decide how effectively the local search proceeds, namely: the **cost function**, the **cooling schedule**, and most importantly, the **local move**.

You will need to play around with the following methods/variables:

Init_temp: the initial temperature for your search. Temperature represents how *rash* you are in accepting moves. If you start out too low, your SA becomes local search, and you defeat the purpose.

colors: the number of colors you want to allow in the solution. While we'll let you know the optimum number of colors for each test case, we still need a valid coloring from you. If you can't get to the optimum, no problem! Take 1 extra color and get us a valid coloring.

Local move function: take a state as an input, give a new state after you do the local move as the output. Remember, you can have multiple local moves occurring with different probabilities. The more diverse your set of moves are, the richer your overall local move is. How rich is sort of a mystery. You'll have to experiment.

Cooling schedule function: Again, if it's TOO slow, then you defeat the motive behind SA and make it a normal local search

Cost function: take a state, return a coloring.

Disp: set this to a positive integer 'n' to see your graph after every n iterations. Use this only for the small graphs (you'll still see your final coloring after SA is finished). Set it to a negative integer when you get tired of watching pretty graphs pop up.

The output shows you your **best score so far**, the **current temperature**, and the **current cost** of the state being processed.

Language specific details:

Python:

- The state is represented by an n-tuple of integers, where each integer represents a color. For example, the first state above is represented as (0, 1, 2, 1)
- The graph is stored as a dictionary keyed by nodes. Each node has a list of adjacent nodes.
  For the graph above: G={1:[2,3], 2:[1,3], 3:[1,2,4],4:[3]}
- Run using: python anneal.py < "data_gc/gc_4_1"

C++:

- The state is represented by a string of length n, where each character represents the coloring (starting at 0. Note: these are characters, not integers so If your coloring has more than 10 colors, you'll just get the symbols that come after 9)
  The first state above is represented as "0121"
- The graph is stored as an unordered_map<int, vector<int≫ keyed by nodes. Each node has aa vector of adjacent nodes.
  For the graph above: G={1:{2,3}, 2:{1,3}, 3:{1,2,4},4:{3}}
- compile using: g++ anneal.cpp –std=c++11
- run using: ./anneal "data_gc/gc_4_1" 3 (if you want to use 3 colors)

Java:

- The state is represented by an n-length integer array, where each integer represents a color. For example, the first state above is represented as {0, 1, 2, 1}
- The graph is stored as a HashMap<Integer, ArrayList<Integer≫ keyed by nodes. Each node has an arraylist of adjacent nodes.
  For the graph above: G={1:{2,3}, 2:{1,3}, 3:{1,2,4},4:{3}}
- Compile using javac Anneal.java
- Run using java Anneal <"data_gc_4_1"

We know that your code will take multiple runs to find a solution, so we'll run your code a whole bunch of times, don't worry. We will judge you based on how quickly you arrive at a solution (how rich your local move is), and how many colors you used.

Of course, local search is also very problem instance dependent, so you're free to change your parameters across the different test cases as well. Please do tell us what testcase corresponds to what parameter so we can recreate it when we're grading.

The optimal number of colors for the test cases are as follows:

4_1: 2
20_7: 8
50_3: 6
70_3: 8
100_1: 5

You need to submit the following:

- A .txt file with the best coloring you can get for each test case (number of colors, followed by the coloring itself—copy paste the coloring from the terminal), and what parameters you used for each test case
- The anneal code file with your modifications
- The result.png for the 100_1 test case from your best run (optional, but it'll be nice to see)