Ma6 PSET 3

Victoria Liu

November 3, 2020

Problem 1

We will describe a rather brute-force algorithm that will logically work, but might be computationally difficult to implement. We design a graph as such: place each possible orientation of the Rubik's cube in a vertex, and vertices are connected by an edge if there is a single rotation (of any one face) that can relate the two configurations. Of course, one of the possible orientations will be the original (or "finished") position, which has each face as a single color. We then run BFS from our present configuration (it'll probably be difficult to find our present configuration from a quintillion vertices, but it's the logic that counts) to find the shortest path between our configuration's vertex and the original configuration's vertex. The following two paragraphs will clarify exactly what the configurations are and how BFS will be run.

We notice that any rotation of the Rubik's cube keeps the relative orientations of the six center panels in place; no matter what configuration the Rubik's cube is in, we can move the entire cube (we are not rotating individual faces) in our hands until the middle panels have the same orientation as the original position. Knowing this, let's fix the original positions of the middle panels and only count configurations according to how the corner and edge pieces move. There are still a quintillion or more different configurations the cube can be in–the eight corner pieces have 8! different possible locations and a little less than 3^8 different spatial orientations within their given positions. A similar argument can be made for the 12 edge pieces, which have 12! different possible locations and a little less than 2^{12} different spatial configurations. We say "a little less than," because some spatial configurations are impossible since a single rotation will change two spatial orientations in the same way. We could multiply all of these large numbers above to find the total number of configurations, and each configuration would be represented by a vertex in our graph.

We run BFS from our current configuration, in exactly the way it was described in lecture. We use a vertex coloring scheme and queue data structure, and we record the parents of each vertex and its distance from the root. To initiate the search, set the starting vertex's distance from itself as 0 and set all other vertices' distances to ∞. Also set all vertices to white. Now, we initiate the queue with a single element, the starting vertex. Every time a vertex is put into the queue, we color it gray, so now the starting vertex is colored gray. Vertices are colored black when we are done doing BFS on it, as described below. While we haven't found the original configuration yet, we do the following steps:

1. Set the head of the queue to u
2. For each adjacent vertex v to u, if the v is colored white, we color v gray, set the distance of v as $u + 1$, define the parent of v as u, and insert v into the queue.

3. Remove u from the queue and color it black

In other words, once we find an adjacent vertex v that is the "finished" configuration, we can exit after finishing that particular iteration of the while loop and return the distance from this configuration to our starting vertex, as well a list of all of the parents (nodes) leading back to our starting node. This string of parent vertices would represent the shortest path from our current Rubik's cube position to the finished position. To get from our current configuration to the finished configuration, we would simply rotate the Rubik's cube as directed by this shortest path.

Problem 2

2.a

Proof. We will show that in a $2k$ regular, connected G, there is a k-regular subgraph H with the same vertex set by describing an algorithm to find the k-regular subgraph.

We proved in class that a connected graph $G = (V, E)$ has an Eulerian cycle if and only if every vertex V has an even degree. Since G is $2k$-regular, each vertex has an even degree, and G contains an Eulerian cycle. Let's start at any vertex v_1 and go through the Eulerian cycle leading back to itself; let's also make this a directed path, so each vertex has k in-degrees and k out-degrees. The edges traversed will look like $v_1 - v_2, v_2 - v_3, v_3 - v_4, v_4 - v_5 ... v_i - v_1$, and we traverse an even number of edges (as per the problem statement). Notice that consecutive edges in the path (i.e $v_1 - v_2, v_2 - v_3$) represent leaving one vertex(v_1), visiting a second vertex(v_2), leaving the second vertex, and entering a third vertex(v_3). Given two consecutive edges, if we remove the first edge, we are removing one outgoing edge of the first vertex(v_1) and one ingoing edge of the second vertex(v_2). Then examining the next pair of consecutive edges (i.e. $v_3 - v_4, v_4 - v_5$), if we again remove the first edge, then we are removing an outgoing edge from the third vertex(v_3) and an ingoing edge to the fourth vertex(v_4). Basically, removing a directed edge results in removing one degree (in or out, *but not both*) from the vertices that the edge connects.

Ok, so now let's formalize this intuition. We separate the set of directed edges in our Eulerian path into two sets–the first set consists of every "odd" edge (i.e. the first edge traversed, the third edge, the fifth, etc), and the second set consists of every "even" edge. We call the set of odd edges A and the set of even edges B. We notice that since we had an even number of edges in G, A and B are the same size and evenly divide the edge set. Most importantly, A and B are both k-regular subgraphs H of G. This is because by removing every other consecutive edge, we are removing one (in or out) degree from each vertex each time we pass through it. Normally in G, visiting an edge means giving it two degrees (one in and one out). A or B represent cutting the degree of each vertex in half, thus making it a k-regular graph.

Of course, during our proof reasoning, we made G and A and B directed graphs to visualize the in/out degrees. We can simply make the graphs undirected again, and the logic still holds; taking away an incoming edge (or an outgoing edge) is the same as removing one single degree.

□

2.b

When G has an odd number of edges, it will no longer have a k-regular subgraph H because there is no way to evenly bisect the set of edges; we need to be able to bisect it evenly so that we

can remove one half of the edges (as we did above) and ensure that each vertex retains half of its original degree. Another way to look at it is that nk is the total number of edges in graph G, where n is the number of vertices. In a subgraph that is k-regular, there would be $\frac{nk}{2}$ edges. If the total number of edges in graph G is odd, then $\frac{nk}{2}$ wouldn't be an integer, and a subgraph that is k-regular wouldn't exist. Finally, one can easily verify with a graph in the shape of a triangle (three vertices, three edges, $k = 1$) that there is not subgraph that is 1-regular.

Problem 3

Proof. All vertices have different sequences of $0/1$'s of length n, where $n > 1$. Let's zoom in on a single vertex out of these vertices. We'll infer some properties about this vertex and then realize these properties are true for all the vertices. Our vertex, which we will call A, will have outgoing edges to vertices whose first $n-1$ numbers are the same as A's last $n-1$ numbers. We realize that there are exactly two such vertices–since the first $n-1$ numbers are pre-defined, we only have the two choices for the last number. Here, we can either have 1 or 0; we won't have any more because the vertex set, which is of size 2^n, only consists of unique representations of $0/1$ sequences of length n. Now, let's consider the number of incoming edges to A. There will be an incoming edge from vertices whose last $n-1$ numbers match the first $n-1$ numbers of vertex A. Using the same line of logic as before, we see that there will be exactly 2 such vertices, since we can only choose between 2 possibilities for the first number of the sequence of the vertex. This means that the in-degree and out-degree of vertex A will both be 2. Since we had no restrictions in choosing A, all vertices of our multigraph will have an in-degree and out-degree of 2 and 2, respectively. There will be cases where vertices will have self-loops; for example, if $n = 4$ and the vertex is $(0,0,0,0)$, there will be a self-directed loop, one outgoing edge to $(0,0,0,1)$, and one incoming edge from $(1,0,0,0)$. For these vertices, since a self-loop adds one degree to both the in-degree and out-degree, the total in-degree and out-degree count are still 2 and 2, respectively.

It is very easy to show that an Eulerian path exists (refer to the last paragraph lecture 8): we learned in class that for a digraph G, if the indegree of every vertex equals its outdegree, then G admits an Eulerian cycle. This makes intuitive sense, since this is the same condition for undirected graph (even degree counts for each vertex), with the added restriction that the number of edges to visit the vertex equals the number edges to leave the vertex.

Because there is an Eulerian cycle, we can visit all 2^n vertices in one cycle. We build a sequence of 0's and 1's as we traverse the cycle, and the sequence will be less than 2^{n+1} and contain every possible $0/1$ sequence of length $n+1$. Here is how to construct the sequence: start with an empty sequence, and add the full, n-length sequence of the first vertex. For each subsequent vertex in the cycle, add the n^{th} number of the vertex's sequence. The sequence will be $2^n + n - 1$ long by the time we end the cycle. First we note that the cycle is less than 2^{n+1} because

$$2^n + n - 1 < 2^n + 2^n - 1 < 2^{n+1} \tag{1}$$

To get a sequence of length exactly 2^{n+1}, we just need to spool around at the end. We can also see that this sequence contains all sequences of 0's and 1's of length $n+1$; after visiting any vertex A, the last n numbers of our (still-growing) sequence will be the exact sequence of vertex A. Vertex A has two outgoing edges, one edge that will cause the sequence to add a 1, and one edge that will cause the sequence to add a 0. In the course of our Eulerian cycle, we will visit A twice and

traverse both outgoing edges in due course. The same logic follows for every single vertex of the graph, meaning that our finished sequence will have the desired $n+1$-length sequence of 0's and 1's. Another way of visualizing this is by thinking of an $n+1$-length sequence as an n-length sequence, and then adding either a 1 or a 0 at the end.

□

For our concrete example of when $n = 4$, let's first enumerate all of our 16 vertices:

$$a = 0, 0, 0, 0 \tag{2}$$

$$b = 0, 0, 0, 1 \tag{3}$$

$$c = 0, 0, 1, 0 \tag{4}$$

$$d = 0, 1, 0, 0 \tag{5}$$

$$e = 1, 0, 0, 0 \tag{6}$$

$$f = 0, 0, 1, 1 \tag{7}$$

$$g = 0, 1, 0, 1 \tag{8}$$

$$h = 1, 0, 0, 1 \tag{9}$$

$$i = 0, 1, 1, 0 \tag{10}$$

$$j = 1, 0, 1, 0 \tag{11}$$

$$k = 1, 1, 0, 0 \tag{12}$$

$$l = 0, 1, 1, 1 \tag{13}$$

$$m = 1, 0, 1, 1 \tag{14}$$

$$n = 1, 1, 0, 1 \tag{15}$$

$$o = 1, 1, 1, 0 \tag{16}$$

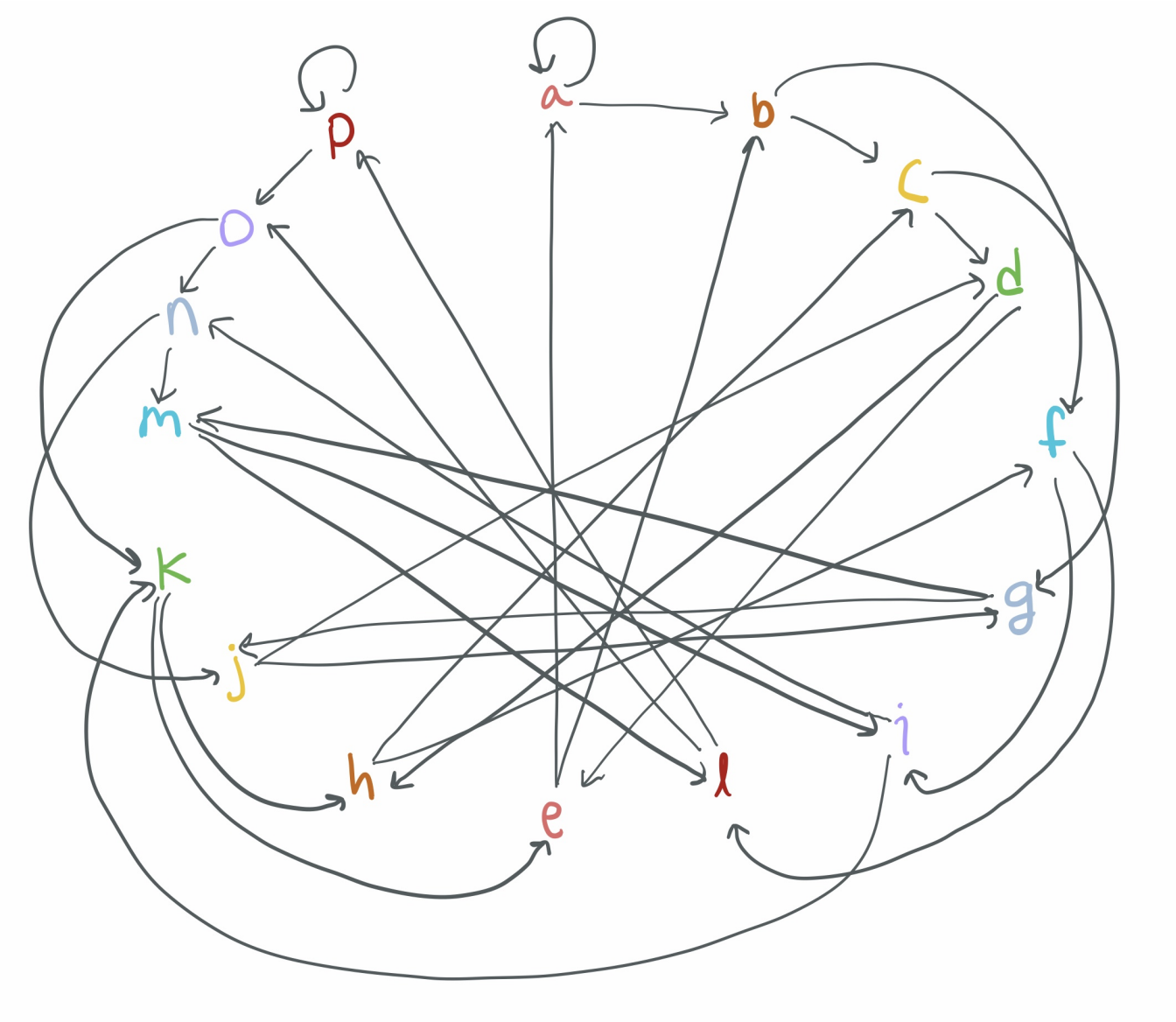
$$p = 1, 1, 1, 1 \tag{17}$$

$$\tag{18}$$

Then, we construct the graph ensuring that a directed edge goes from u to v if the last $n-1$ entries of u are the same as the first $n-1$ entries of v. The resulting graph would have the following edges:

$$
\begin{aligned}
&(a \to a), (a \to b), (b \to c), (b \to f), (c \to d), (c \to g), (d \to e), (d \to h),\\
&(e \to a), (e \to b), (f \to l), (f \to i), (g \to m), (g \to j), (h \to c), (h \to f),\\
&(i \to n), (i \to k), (j \to d), (j \to g), (k \to e), (k \to h), (l \to o), (l \to p),\\
&(m \to l), (m \to i), (n \to m), (n \to j), (o \to n), (o \to k), (p \to o), (p \to p)
\end{aligned}
\tag{19}
$$

Let's plot this and see what it looks like:



Ok, now we will define an Eulerian cycle on this graph:

$$
\begin{aligned}
&(a \to b \to c \to d \to e \to b \to f \to l \to o \to n \to j \to d \to \\
&h \to c \to g \to j \to g \to m \to i \to n \to m \to l \to p \to p \to \\
&o \to k \to h \to f \to i \to k \to e \to a \to a)
\end{aligned} \tag{20}
$$

There are exactly 32 edges and each edge is crossed exactly once. Let's write the corresponding $0-1$ sequence of this Eulerian cycle:

$$
0,0,0,0,1,0,0,0,1,1,1,0,1,0,0,1,0,1,0,1,1,0,1,1,1,1,1,0,0,1,1,0,0,0 \tag{21}
$$

Problem 4

4.a

Proof. The notation is a little confusing, but when I refer to $\chi(G)$, I am referring to a constant with regards to a particular G.

We assume that the graph is simple, since adding loops and parallel edges would only increase the number of edges without creating new connections that might warrant more colors; we are trying to find the minimum number of edges, so it makes sense to assume the graph is simple. We do a direct proof, where we show why each assumption we make must be true by showing that the opposite is impossible.

In order to admit a $\chi(G)$ coloring, G needs at least $\chi(G)$ vertices. If we had less than $\chi(G)$ vertices, it's pretty obvious that we can color it with less than $\chi(G)$ colors. So now let's consider a graph of exactly $\chi(G)$ vertices. These $\chi(G)$ vertices must be completely connected, and each vertex has degree $\chi(G)-1$. If the graph was not a complete graph, then at least one vertex would have less than $\chi(G)-1$ degrees, and we would be able to find a $\chi(G)-1$ coloring (this follows from lecture; run BFS from a vertex with less than $\chi(G)-1$ degrees, and then color the vertices in decreasing order of the BFS levels. Within each level, a given vertex connects with at most $\chi(G)-2$ other nodes, so a $\chi(G)-1$ coloring is doable for every node within a level. When moving to the parent level, the parent is connected to at most $\chi(G)-2$ nodes at its level or below, so a $\chi(G)-1$ coloring is again possible. In the end, we can also color the root node since its degree is less than $\chi(G)-1$). Ok, so now we've established that we need all vertices to have $\chi(G)-1$ degrees. Thus, the smallest G must be a complete graph with $\chi(G)$ vertices. From lecture, we know that the smallest k-coloring of a complete graph G is the number of vertices, which is $\chi(G)$. This is exactly what the problem requires. Great, now let's see how many edges there are in G:

$$
number\ of\ edges = \frac{\chi(G) \cdot (\chi(G)-1)}{2} \tag{22}
$$

Looks like 22 is the same value as $\binom{\chi(G)}{2}$, so we've shown that the minimum number of edges in G is $\binom{\chi(G)}{2}$.

For clarification, we've shown that in order for a graph *with* $\chi(G)$ *vertices* to have $\chi(G)$-coloring, it must have at least $\binom{\chi(G)}{2}$ edges. Is it possible that a graph of greater than $\chi(G)$ vertices can

have fewer than $\binom{\chi(G)}{2}$ edges and also require at least $\chi(G)$-coloring? This is easily shown to be impossible; let's first start with an incomplete graph of $\chi(G)$ vertices. If we add a $\chi(G) + 1^{th}$ vertex, we would still need to create a graph where every vertex had at least $\chi(G) - 1$ degrees. Adding this new vertex would only use up unnecessary edges to connect it to our graph, when our end goal is the same as the case where we had exactly $\chi(G)$ vertices. If the new vertex was isolated, then we'd be able to color it with any color that was already used in the original graph.

To conclude, a graph with the least number of edges that requires a minimum $\chi(G)$-coloring is a complete graph with $\chi(G)$ vertices. We've shown that an incomplete graph of $\chi(G)$ vertices can use $\chi(G) - 1$ coloring, and we've also shown that graphs with more than $\chi(G)$ vertices that require $\chi(G)$-coloring always have more than $\binom{\chi(G)}{2}$ edges.

□

4.b

I feel like I already answered this question in part a, but basically the smallest connected graph that requires $\chi(G)$-coloring is a complete graph of $\chi(G)$ vertices. For an incomplete graph with $\chi(G)$ vertices, we will be able to find a $\chi(G) - 1$ coloring for it (refer to part a, where it is explained in depth), so only complete graphs of $\chi(G)$ vertices require $\chi(G)$ coloring.

For any graphs with more than $\chi(G)$ vertices, we still require $\chi(G)$ vertices with at least $\chi(G) - 1$ degree, in order to need $\chi(G)$-coloring. This graph would require at least $\frac{\chi(G)\cdot(\chi(G)-1)}{2}$ edges. Since our graph must be connected and we have more than $\chi(G)$ vertices, we have at least $\binom{\chi(G)}{2} + 1$ edges. The 1 comes from needing to connect the graph to a $\chi(G) + 1^th$ vertex. Thus, any connected, incomplete graph must have more than $\binom{\chi(G)}{2}$ edges.

Problem 5

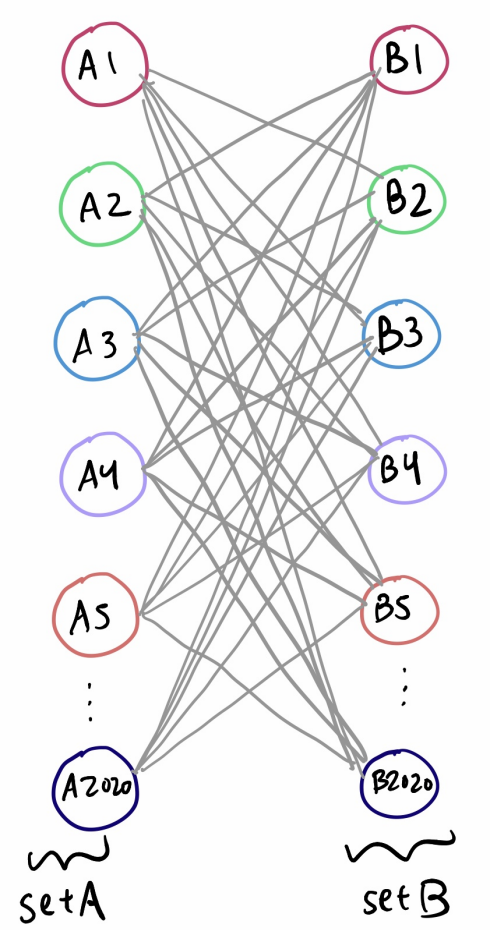
5.a

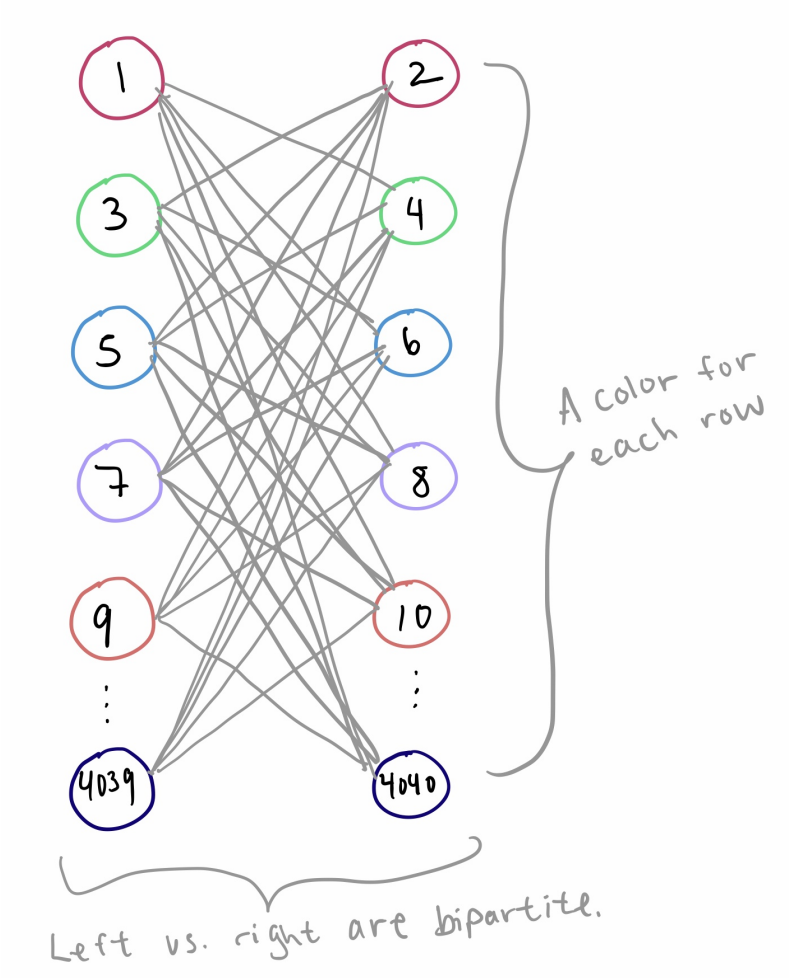
Luckily, we are only showing that such an ordering exists, rather than finding an algorithm to find it, given an uncolored graph (thought it was the second one at first..)! Since we are just proving existence, let's say that we already know how the graph G is minimally-colored; in other words, we know what color each vertex is, and there are exactly $\chi(G)$ colors. We can just order the vertices based on color. We order the colors, and we order the vertices by numbering all the vertices of one color before moving onto the next color. For example, if our graph is 3-colored, we number all the green vertices first, then move onto the red vertices, and finally the blue vertices. With this ordering, we will use exactly $\chi(G)$ colors; we can't use less because that would contradict the minimality of $\chi(G)$, and we won't use more because every vertex will be colored either its original color in G or a color "lower" (in our sequence of colors) than itself.

5.b

Since we are showing the existence of such a bipartite graph G, we can define it any way we want. Since G is bipartite, it has two "disjoint" vertex sets. Let's call the two disjoint vertex sets A and B. We have 2020 vertices in set A and 2020 vertices in set B, and we number the vertices from 1 to 2020. Let's connect vertex 1 from set A (henceforth known as $A1$) to every vertex in set B except for vertex $B1$, and we do the same for vertex $A2$, $A3$, ... etc. to $A2002$, except we

make sure not to connect to vertex $B2$, $B3$, ... etc. $B2002$, respectively. In other words, each vertex of the graph will be connected to exactly 2019 vertices in the other set, and the one vertex it will not be connected to is the one sharing its own number (refer to 2). Now, we can order the vertices by "row" (refer to the 3 for the vertex ordering). The order of vertices would be like: $A1, B1, A2, B2, A3, B3, ...A2019, B2019, A2020, B2020$. This way, every consecutive pair of vertices will have the same color (remember that vertices like $A1$ and $B1$ are not connected to each other), but consecutive pairs will have different colors from all other consecutive pairs. This is because every vertex of a consecutive pair (i.e. $A1$ of $A1, B1$) is connected to every vertex of the other disjoint set, except for the vertex it is in a consecutive pair with. This makes much more sense with a diagram, where the vertices are ordered, and it is clear that each row would be the same color but cannot be the same color as other rows.




1
2
3
4
5
6
7
8
9
10
4039
4040
A color for each row
Left vs. right are bipartite.