CPSC 320 2021W2: Assignment 1

This assignment is due **Monday October 3, 2022 at 22:00 Pacific Time**. Assignments submitted **within 12 hours after the deadline**[1] will be accepted, but a penalty of up to 15% will be applied. Please follow these guidelines:

- Prepare your solution using LaTeX, and submit a pdf file. Easiest will be to use the .tex file provided. For questions where you need to select a circle, you can simply change `\fillinMCmath` to `\fillinMCmathsoln` .
- Enclose each paragraph of your solution with `\begin{soln}Your solution here...\end{soln}`. Your solution will then appear in dark blue, making it a lot easier for TAs to find what you wrote.
- Start each problem on a new page, using the same numbering and ordering as this assignment handout.
- Submit the assignment via GradeScope at `https://gradescope.ca`. Your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**.
- After uploading to Gradescope, link each question with the page of your pdf containing your solution. There are instructions for doing this on the CPSC 121 website, see `https://www.students.cs.ubc.ca/~cs-121/2020W2/index.php?page=assignments&menu=1&submenu=3`. Ignore the statement about group size that is on that page.

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode should communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. If you choose to use actual code, note that you may **neither** include what we consider to be irrelevant detail **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid language-specific notation like C/C++/Java's ternary (question-mark-colon) operator.)

Remember also to **justify/explain your answers**. We understand that gauging how much justification to provide can be tricky. Inevitably, judgment is applied by both student and grader as to how much is enough, or too much, and it's frustrating for all concerned when judgments don't align. Justifications/explanations need not be long or formal, but should be clear and specific (referring back to lines of pseudocode, for example). Proofs should be a bit more formal.

On the plus side, if you choose an incorrect answer when selecting an option but your reasoning shows partial understanding, you might get more marks than if no justification is provided. And the effort you expend in writing down the justification will hopefully help you gain deeper understanding and may well help you converge to the right selection :).

Ok, time to get started...

[1] For future assignments, the grace period will be 24 hours. We have shortened it for this assignment only so that the assignment solutions can be made available in advance of the first take-home test.

1 Statement on Collaboration and Use of Resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments in CPSC 320 (see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=3`).

   ◯ Yes ◯ No

2. We used the following resources (list books, online sources, etc. that you consulted):

3. One or more of us consulted with course staff during office hours.

   ◯ Yes ◯ No

4. One or more of us collaborated with other CPSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

   If yes, please list their name(s) here:

5. One or more of us collaborated with or consulted others outside of CPSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.

   ◯ Yes ◯ No

   If yes, please list their name(s) here:

2 It's a Job Seeker's Market

You're in charge of matching CS undergraduate students with companies for summer jobs. By astonishingly good luck, you have n employers for n student applicants. The job market for software developers is booming: as a result, every employer desperately wants someone to fill their summer job spot, but **not** every student desperately wants to get a job, because they can likely find a job outside the department program if need be (or they can take summer courses, backpack through Asia, etc.). As a result, every employer has a complete preference list of all n students, but some students are unwilling to work for some employers and therefore have incomplete preference lists. A student would rather work for an employer on her preference list than be unmatched, but would rather be unmatched than work for an employer **not** on her preference list.

For example, for $n = 3$, a student s_1 may have the preference list $[e_2, e_3]$, which indicates that she is unwilling to work employer e_1. She would prefer to work for e_2 or e_3 than be unmatched, but would rather be unmatched than work for e_1.

An employer would prefer to be matched with **any** student than to be unmatched.

1. **[2 marks]** We will need to expand our definition of "instability" for this problem. The definition we saw in class still applies to the CS internship problem: namely, when e_i is matched with a student and s_j is matched with an employer on their preference list, but e_i and s_j prefer each other to their current partner. We can also consider it to be an instability when a student is matched with an employer not on his preference list: in this case, the student has an incentive to break the imposed matching (by quitting) and be happier as a result.

   Describe two new types of instability that can occur between: (a) an unmatched student and a matched employer; and (b) a matched student and an unmatched employer.

The first instability occurs when student s_j is unmatched, e_i is matched with s_k, e_i prefers s_j to s_k, and e_i is on s_j's preference list. This is unstable because e_i and s_j can break the imposed matching (by having e_i fire s_k and hire s_j) and be happier as a result.

The second instability occurs when e_i is unmatched and s_j is matched with e_k but prefers e_i. Notice that we don't need to say anything about e_i's preferences because we know from the problem description that e_i prefers to be matched with anyone than to remain unmatched.

2. **[1 mark]** Give and briefly explain a small example in which no stable perfect matching (i.e., a stable matching where every employer and every student is paired up) is possible.

   Consider the following example:

```
e1:  s1  s2      s1:  e1
e2:  s2  s1      s2:  e1
```

   Clearly, we can't obtain a stable perfect matching because both students would prefer to be unmatched than to be matched with e_2. Hence any perfect matching will cause the kind of instability described in question 1.

Modifying Gale-Shapley

We will now modify Gale-Shapley to solve SMP with the change that not every student needs to list every employer in their preference list.

Here is the Gale-Shapley algorithm, with employers making offers:

```
1:  procedure GALE-SHAPLEY(E, S)
2:      initialize all employers in E and students in S to unmatched
3:      while an unmatched employer with at least one student on its preference list remains do
4:          choose such an employer e ∈ E
5:          make offer to next student s ∈ S on e's preference list
6:          if s is unmatched then
7:              Match e with s                                  ▷ s accepts e's offer
8:          else if s prefers e to their current employer e′ then
9:              Unmatch s and e′                                ▷ s rejects e′
10:             Match e with s                                  ▷ s accepts e's offer
11:         end if
12:         cross s off e's preference list
13:     end while
14:     report the set of matched pairs as the final matching
15: end procedure
```

With a small change, we can apply this algorithm and ensure that the (not necessarily perfect) matching produced never matches a student with an employer not in his preference list.

3. **[1 mark]** Make the small change necessary to the algorithm above.

   The necessary change is to modify line 6 to: "if s is unmatched *and e is on s's preference list* then". An alternative correct solution would be to have the students make the offers instead of the employers.

4. **[3 marks]** Prove that, in your modified G-S algorithm, a student can never end up matched with an employer not on their preference list.

   Proof if you modified line 6 as indicated above: the only way a student can end up matched to an employer is in the blocks of code after lines 6 or 8. This means that s must either meet the condition in line 6 (now "s is unmatched and e is on s's preference list") or the one on line 8 ("s prefers e to

their current employer e'"). If e is not on s's preference list then neither condition can hold; thus, s and e will never be matched.

Proof if you had the students make the offers instead: because of the condition on line 5, students will only make offers to employers on their preference list. Because the only way a student and an employer will be matched is if the student first makes an offer to the employer, we can conclude that no student can be matched to an employer not on their preference list.

Special-Case Reductions

5. **[3 marks]** Suppose that **every** student refuses to work for the **same** employers. Without loss of generality[2], suppose that every student's preference list includes employers $e_1, e_2, \ldots, e_j$ and excludes employers e_i for $i > j$, where $j \leq n$.

   Give a reduction from this special case of the CS internship problem to SMP. (Remember that your reduction must describe both how to convert the CS intership instance to an SMP instance **and** how to convert the solution from SMP back to a solution for the CS internship problem.)

   When every student has the same subset of employers in their preference list, we effectively have n students all competing to be matched with fewer than n positions. Our desired solution will necessarily be an "imperfect" matching, where some students are matched with the preferred employers, and some are unmatched. Intuitively, we want our reduced SMP solution to return a perfect matching where the students most preferred by the desired employers are matched with them, and the least preferred ones aren't. The easiest way to handle this is to add the undesired employers to the bottom of everyone's preference lists, and at the end simply remove from the matching all pairs involving one of the non-preferred employers.

   The complete reduction:

   - Append $e_{j+1}, \ldots, e_n$ to each student's preference list. (They can be appended in any order and the reduction will still be correct: the important thing is that all the extra employers must be added, and they must be placed below the employers already on the list.)
   - Solve the SMP instance given by the employer preference lists and the modified student preference lists.
   - Given the matching returned by the SMP solver, remove all pairs that contain any of the employers $e_{j+1}, \ldots, e_n$. Return the result as the final matching.

6. **[5 marks]** Prove the correctness of your reduction from 2.5. In other words, prove that if the matching returned by the SMP solver is correct (i.e., is a perfect matching that contains no instabilities), then the matching returned by your reduction will:

   (a) Have exactly one student matched with each non-rejected employer (we can think of this as our definition of a "valid" solution to this version of the problem); and

   (b) Contain no instabilities.

   We'll start with part (a). Because the SMP solver returns a perfect matching between *all* the employers and students, we know that our solution will have one student matched with each non-rejected employer.

   The proof of (b) is similar to the proof of correctness we did in class for the RHP to SMP reduction: we show that, if the result returned by the reduction contains an instability, it must have resulted from an instability in the solution returned by the underlying SMP solver. The difference here is that

[2] We can say "without loss of generality" here because, no matter which employers are excluded, we can just reorder the indices so that the rejected employers are at the end of the list.

there are **several** different kinds of instability we need to check. So, we will go through each of the five instabilities listed in question 1 and show that any such instability would imply an instability that came from the solution of the SMP solver.

- **Case 1: instability between a matched student and a matched employer.** This is where e_i is matched with a student and s_j is matched with an employer on their preference list, but e_i and s_j prefer each other. If our reduction returns this, it means that the SMP solver paired e_i and s_j with other people on their preference lists, despite e_i and s_j preferring each other. In other words, this instability came from the SMP solver.
- **Case 2: instability where a student is matched with an employer not on their preference list.** We actually know that this can't occur in our returned solution: in the last step of our reduction, we got rid of all the pairs that involved an employer not on anyone's preference list.
- **Case 3: instability between an unmatched student and a matched employer.** Suppose our returned solution has an unmatched student s_j and an employer e_i (on s_j's preference list and therefore not among the rejected employers) who is matched with s_k but prefers s_j. The SMP solver must have had e_i matched with s_k and s_j matched with e_m, where e_m is one of the employers not on anyone's preference list. By our reduction, e_m is on s_j's modified preference list to the SMP solver but is **below** e_i; therefore, e_i and s_j must constitute an instability in the matching returned by the SMP solver.
- **Case 4: instability between a matched student and an unmatched employer.** Suppose that our returned solution has e_i unmatched, and s_j matched with e_k but preferring e_i. Referring to our reduction, if e_i is unmatched it means that it was one of the employers not on anyone's preference list and whom we removed from the matching in the last step. But this means that this can't be an instability in our solution (because e_i was not on s_j's preference list, which means that s_j could not have preferred them to e_k).

We have now addressed every possible type of instability and either shown directly that they cannot exist in the solution, or that if the instability did exist it would have to have been caused by an incorrect (i.e., unstable) solution returned by the underlying SMP solver. This completes the proof.

3 Breaking the Chain

You're an administrator of a computer network. You want all computers in the network to be able to communicate with each other, and you also want to avoid a situation where one computer going offline breaks the communication path between other computers in the network.

It's common to represent networks as graphs, with each node representing a computer and edges between nodes representing the connections between computers in the network. You're interested in how the diameter of a network graph relates to the level of vulnerability of that network. Assume that your network is represented by a graph $G = (V, E)$, containing n nodes and m edges.

1. **[4 marks]** Suppose the diameter of the network is strictly greater than $n/2$, and let s and t denote the diametric nodes. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all paths from s to t. Hint: think about a BFS tree rooted at s.

   Per the hint, consider a BFS tree rooted at s. We know that t is at the bottom of the tree, and that the tree has more than $\frac{n}{2}$ levels below the root. This means there is at least one level of the tree (apart from the root) that has only one node – if all levels below the root had two or more nodes, then the total number of nodes would be greater than n.

   Now, suppose that level k of the BFS tree has only one node, which we'll call v. Because each node's position in the BFS tree is located along its shortest path from s, the only way to get from a node

at level $k-1$ to a node at level $k+1$ is through v. (If there were another way to reach the nodes at level $k+1$ besides going through v at level k, that would constitute a shorter path to those nodes, and those nodes would have appeared higher up in the tree.) Therefore, all paths from s to t must go through v, and the removal of v will destroy all paths from s to t.

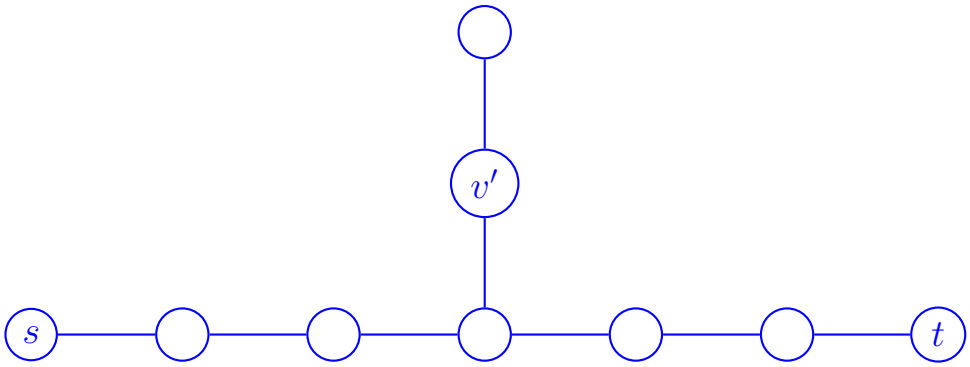
2. **[4 marks]** Give an algorithm to find such a node v. You may assume that you are given the nodes s and t, and that the distance between them is strictly greater than $n/2$. For full credit, your algorithm must run in $O(m+n)$ time.

Our approach is to run BFS from s, and once we reach a level of the tree (other than the root) that contains only a single node, we return that node. Below is the BFS algorithm as found in Section 3.3 of Kleinberg & Tardos. The only change we've made is at line 16; after determining what nodes are at layer $i+1$ of the BFS tree, we check how many nodes are there. If there's only one, that means we've found the node v, so we stop the BFS procedure and return v.

```
1: procedure FindV(G, s)
2:     Set Discovered[s] = true and Discovered[v] = false for all other v
3:     Initialize L[0] to consist of the single element s
4:     Set the layer count i = 0
5:     Set the current BFS tree T = ∅
6:     while L[i] is not empty do
7:         Initialize an empty list L[i + 1]
8:         for each node u ∈ [i] do
9:             Consider each edge (u, v) adjacent to u
10:            if Discovered[v] = false then
11:                Set Discovered[v] = true
12:                Add edge (u, v) to the tree T
13:                Add v to the list L[i + 1]
14:            end if
15:        end for
16:        if L[i + 1] has length 1 then return the single node v in L[i + 1]   ▷ found v! (only node at layer i + 1)
17:        end if
18:        Increment the layer counter i by one
19:    end while
20: end procedure
```

If you didn't manage to come up with the proof in 3.1, you may have had to use the brute force approach. A correct brute force algorithm would be to check every node not equal to s or t by removing it and all its incident edges from G, running BFS or DFS from s, and stopping once you found a node whose removal meant you could no longer reach t from s. Because this algorithm involves up to $O(n)$ calls to BFS/DFS, its runtime complexity is $O(n^2+nm)$.

There is another tempting but incorrect approach that you might have taken after reading worksheet 1. From the definition of v, it's clear that v is an articulation point in G. If you remembered about articulation points from worksheet 1 (and, in particular if you did the challenge problem about finding articulation points), you might be tempted to use as your algorithm: find all articulation points in G in $O(n+m)$ time (as described here) and return an articulation point that is not equal to s or t. However, this approach is incorrect because, while a point v whose removal breaks the path from s to t must be an articulation point in G, not all articulation points in G will necessarily break the paths from s to t if removed from the graph. To see why, consider the graph below:



Notice that v' is an articulation point, but it's not on the path from s to t. So any algorithm that starts by finding all articulation points would need to additionally check if the removal of those articulation point would separate s and t. To our knowledge, it won't be possible to generate a correct linear-time algorithm using this approach.

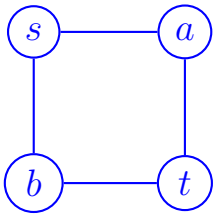
3. **[2 marks]** Briefly justify a good asymptotic bound on the runtime of your algorithm.

   Our algorithm is just BFS (which has runtime $O(m+n)$), with the only extra work being what's added on line 16. Checking the length of $L[i+1]$ is either constant time or linear in the size of $L[i+1]$, depending on implementation details. We would consider it reasonable to claim without further explanation this takes constant time, which would mean the algorithm is still $O(m+n)$. However, even if the runtime is linear in the length of $L[i+1]$, each node is going to appear in a list L at most once, meaning that the total cost of all these operations would be no more than $O(n)$. Therefore, the algorithm's runtime is $O(m+n)$, as required.

   (Obviously, the correct answer here is the runtime of **your** algorithm, which may not be $O(n+m)$. E.g., if you came up with a correct brute force approach the runtime would be $O(n^2+nm)$, as discussed in the solution to the previous question.)

4. **[3 marks]** Suppose $n \geq 4$ is even and the distance between the diametric nodes s and t is *equal to* $n/2$ (instead of strictly greater). Is there still always a node v not equal to s or t such that deleting v from G destroys all paths from s to t? If yes, provide a proof. If no, provide an example in which no such v exists (s and t should be clearly labelled; you may lose points if your example is needlessly large or confusing).

   No, such a node does not always exist. Consider the following counterexample with $n = 4$:



   Here s and t have a distance of 2, and there are two distinct length-2 paths from s to t: $s \to a \to t$, and $s \to b \to t$. No matter which of the nodes a or b is deleted, there is still a path remaining from s to t.

4 Colour Me Puzzled

Boolean satisfiability (SAT) is, as far as Computer Scientists know, a hard problem. We will define precisely what we mean by this later on in the course. This problem is defined as follows:

The input is a collection of m *clauses* over n boolean variables X_1, X_2, ... X_n. Each clause is a disjunction of some of the variables or their complements.

The problem consists in answering the question "Is there a way to assign truth values to each variable that makes **every** clause of the instance TRUE?

For instance, here is an instance of SAT made of 4 clauses over 3 variables:

1. $X_1 \vee \overline{X}_3$
2. $\overline{X}_1 \vee \overline{X}_2 \vee X_3$
3. $\overline{X}_2 \vee \overline{X}_3$
4. X_3

Equivalently, you could think of this instance as the single boolean expression

$$(X_1 \vee \overline{X}_3) \wedge$$
$$(\overline{X}_1 \vee \overline{X}_2 \vee X_3) \wedge$$
$$(\overline{X}_2 \vee \overline{X}_3) \wedge$$
$$X_3$$

There are many other problems whose level of difficulty is similar to that of Boolean Satisfiability, and this problem has attracted a lot of research. So while the problem is difficult to solve in general, a number of tools to solve instances of the problem have been developed (see the Wikipedia page for details and links). We can take advantage of these tools to solve other problems using *reductions*.

In this question, we consider the **Grid-3-colour problem**. We're given an $m \times n$ grid where each cell can be coloured red, green, or blue, and some cells have already been coloured. Our goal is to find a colouring for the remaining grid cells which is a valid **3-colouring**: that is, where we use no more than the three permitted colours and no neighbouring cells in the grid share a colour. A cell at position (i, j) has up to four neighbours: those at positions $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, and $(i, j-1)$ (if $i = 1$ or m and/or $j = 1$ or n, the cell is located on the border of the grid and has fewer than four neighbours).

For example, the 3×2 grid with (1,2) in red, (2,1) in blue, and (3,2) in blue:

(1,2)	(2,2)	(3,2)
(1,1)	(2,1)	(3,1)

has a 3-colouring if we colour (1,1) and (2,2) in green, and (3,1) either green or red.

If the same grid has the same initial colouring but with (2,1) in green,

(1,2)	(2,2)	(3,2)
(1,1)	(2,1)	(3,1)

then there is no valid 3-colouring because there's no colour we can legally use for (2,2).

1. **[5 marks]** Show how to reduce an arbitrary instance of the Grid-3-colour problem into an instance of Boolean Satisfiability (SAT). Hint: the solution I have in mind introduces 3 variables for each cell of the grid.

Typically, the best first step in a reduction to SAT is to determine what the variables should "mean", in the context of the original problem. Per the hint in the question to designate three variables per grid cell, we'll create a SAT variable that indicates whether a given grid cell is a particular colour. That is, the variable $X_{i,j,r}$ will be TRUE if and only if cell (i, j) is coloured red, $X_{i,j,b}$ will be TRUE if and only if cell (i, j) is blue, and $X_{i,j,g}$ will be TRUE if and only if cell (i, j) is green.

Next, we need to determine an appropriate set of clauses to encode the "rules" of the Grid-3-colour problem. The rules we need to encode are:

1) **Every cell must be assigned a colour (red, green, or blue).**
2) **A cell cannot be assigned more than one colour.**
3) **Neighbouring cells cannot be assigned the same colour.**
4) **The colours of some cells are already given to us.**

We will now proceed to add clauses for each of the stated rules. The clauses for the reduction are added in the underlined parts of the text denoted by Clauses Step (X):.

1) **Every cell must be assigned a colour (red, green, or blue).**
   We have designated variables to encode that a cell (i, j) is red or green or blue. The constraint that each cell must have a colour is equivalent to saying one of $X_{i,j,r}$, $X_{i,j,b}$ or $X_{i,j,g}$ must be true for all i, j.
   Clauses Step 1: For all $i \in [1, m]$ and $j \in [1, n]$, add the clause
   $$(X_{i,j,r} \vee X_{i,j,g} \vee X_{i,j,b})$$

2) **A cell cannot be assigned more than one colour.**
   To start, let's think about what this means in terms of just two colours, red and green. If cell (i, j) can't be more than one colour, that means it can't be both red and green. Because each clause needs to be written as a sequence of literals joined by a logical OR, we can think of this as: cell (i, j) must be not red or not green (or both, if the cell is blue). Similarly, it must also be "not red or not blue", and "not green or not blue".
   Clauses Step 2: For all $i \in [1, m]$ and $j \in [1, n]$, add the clauses
   $$(\overline{X}_{i,j,r} \vee \overline{X}_{i,j,g}) \wedge (\overline{X}_{i,j,r} \vee \overline{X}_{i,j,b}) \wedge (\overline{X}_{i,j,g} \vee \overline{X}_{i,j,b})$$

3) **Neighbouring cells cannot be assigned the same colour.**
   We can express this constraint pretty easily in the "language" of SAT if we break it into three cases: neighbouring cells can't both be red, and can't both be green, and can't both be blue. As in the previous step, we can convert this to a logical OR statement by saying: either a cell is not red or its neighbouring cell is not red, etc. We'll add the clauses in two steps – one for horizontal neighbours (adjacent in the x direction), and one for vertical neighbours (adjacent in the y direction).
   Clauses Step 3a (horizontal direction): For all $i \in [1, m-1]$ and $j \in [1, n]$, add the clauses
   $$(\overline{X}_{i,j,r} \vee \overline{X}_{i+1,j,r}) \wedge (\overline{X}_{i,j,g} \vee \overline{X}_{i+1,j,g}) \wedge (\overline{X}_{i,j,b} \vee \overline{X}_{i+1,j,b})$$
   Clauses Step 3b (vertical direction): For all $i \in [1, m]$ and $j \in [1, n-1]$, add the clauses
   $$(\overline{X}_{i,j,r} \vee \overline{X}_{i,j+1,r}) \wedge (\overline{X}_{i,j,g} \vee \overline{X}_{i,j+1,g}) \wedge (\overline{X}_{i,j,b} \vee \overline{X}_{i,j+1,b})$$

4) **The colours of some cells are already given to us.**

Fortunately, this one is easy – with how our reduction is set up, it just means we already know the values of some of the variables. For example, if cell (i, j) is blue, then $X_{i,j,b}$ is TRUE, which we can express by adding the clause $(X_{i,j,b})$. (We also could add $(\overline{X}_{i,j,r})$ and $(\overline{X}_{i,j,g})$, though this isn't necessary as those are necessarily implied by the clauses added in Step 2.)

Clauses Step 4: For each pre-coloured cell (i, j), add the clause

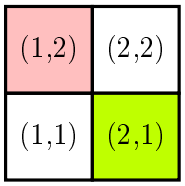
$$X_{i,j,c}$$

where c corresponds to the colour of the cell (r for red, g for green, b for blue).

A note on this reduction: you could have come up with a more efficient reduction (i.e., one that would generate a smaller SAT instance) by not assigning variables to the cells that are pre-coloured. However, this reduction is a little bit harder to "code" (as your indices will be a bit harder to manage, and you would need to include more cases to handle the "neighbouring cells cannot be assigned the same colour" constraint in particular). So, in the interest of creating a more easily readable solution, we did not take this approach here.

Another advantage of our approach over the approach that doesn't assign variables to pre-coloured cells is that ours automatically takes care of inputs where the input itself is an illegal colouring by returning not satisfiable in that instance. For example, suppose our input has two neighbouring squares that are both coloured in red. Our reduced SAT instance will require that $X_{i,j,r}$ be True for both squares, but the clauses added in Step 3 of the reduction will require that at least one of those cells not be red, so the instance will return not satisfiable/no valid colouring. If you opted for an approach that didn't assign variables to pre-coloured squares, you would either need to explicitly assume that the provided colours all satisfy the given constraints, or you would need to first check whether the initial colouring was valid.

2. **[1 marks]** Write the collection of clauses you would get for the 2×2 Grid-3-colour instance:



(where the (1,2) square is red and (2,1) is green).

Obviously, your solution to this question may be different, depending on how exactly you went about

the reduction. Here is our collection of clauses for this instance of Grid-3-colour:

$$
\begin{aligned}
&(X_{1,1,r} \vee X_{1,1,g} \vee X_{1,1,b}) \wedge \\
&(X_{1,2,r} \vee X_{1,2,g} \vee X_{1,2,b}) \wedge \\
&(X_{2,1,r} \vee X_{2,1,g} \vee X_{2,1,b}) \wedge \\
&(X_{2,2,r} \vee X_{2,2,g} \vee X_{2,2,b}) \wedge \\
&(\overline{X}_{1,1,r} \vee \overline{X}_{1,1,g}) \wedge (\overline{X}_{1,1,r} \vee \overline{X}_{1,1,b}) \wedge (\overline{X}_{1,1,g} \vee \overline{X}_{1,1,b}) \wedge \\
&(\overline{X}_{1,2,r} \vee \overline{X}_{1,2,g}) \wedge (\overline{X}_{1,2,r} \vee \overline{X}_{1,2,b}) \wedge (\overline{X}_{1,2,g} \vee \overline{X}_{1,2,b}) \wedge \\
&(\overline{X}_{2,1,r} \vee \overline{X}_{2,1,g}) \wedge (\overline{X}_{2,1,r} \vee \overline{X}_{2,1,b}) \wedge (\overline{X}_{2,1,g} \vee \overline{X}_{2,1,b}) \wedge \\
&(\overline{X}_{2,2,r} \vee \overline{X}_{2,2,g}) \wedge (\overline{X}_{2,2,r} \vee \overline{X}_{2,2,b}) \wedge (\overline{X}_{2,2,g} \vee \overline{X}_{2,2,b}) \wedge \\
&(\overline{X}_{1,1,r} \vee \overline{X}_{2,1,r}) \wedge (\overline{X}_{1,1,g} \vee \overline{X}_{2,1,g}) \wedge (\overline{X}_{1,1,b} \vee \overline{X}_{2,1,b}) \wedge \\
&(\overline{X}_{1,2,r} \vee \overline{X}_{2,2,r}) \wedge (\overline{X}_{1,2,g} \vee \overline{X}_{2,2,g}) \wedge (\overline{X}_{1,2,b} \vee \overline{X}_{2,2,b}) \wedge \\
&(\overline{X}_{1,1,r} \vee \overline{X}_{1,2,r}) \wedge (\overline{X}_{1,1,g} \vee \overline{X}_{1,2,g}) \wedge (\overline{X}_{1,1,b} \vee \overline{X}_{1,2,b}) \wedge \\
&(\overline{X}_{2,1,r} \vee \overline{X}_{2,2,r}) \wedge (\overline{X}_{2,1,g} \vee \overline{X}_{2,2,g}) \wedge (\overline{X}_{2,1,b} \vee \overline{X}_{2,2,b}) \wedge \\
&X_{1,2,r} \wedge \\
&X_{2,1,g}
\end{aligned}
$$

3. **[2 marks]** What is the length of the SAT instance (i.e., the total number of literals), as a function of m, n, and the number of pre-filled squares p?

   We'll go through each of the "Clauses Step" portions of our answer to 4.1 and count up the length that each portion contributes to the reduced instance. (As with the previous question, your answer may differ from ours and still be correct if you set up your reduction differently.)

   - Clauses Step 1: generates mn clauses, each of length 3 = total length $3mn$.
   - Clauses Step 2: generates $3mn$ clauses, each of length 2 = total length $6mn$.
   - Clauses Step 3a: $3(m-1)n$ clauses, each of length 2 = total length $6mn - 6n$.
   - Clauses Step 3b: $3m(n-1)$ clauses, each of length 2 = total length $6mn - 6m$.
   - Clauses Step 4: p clauses, each of length 1 = total length p.

   Therefore, the total length of the SAT instance generated by our reduction is $21mn - 6(n+m) + p$.

4. **[4 marks]** Explain briefly why, if the Grid-3-colour input has a valid 3-colouring, then there is a way to assign values to the variables in the reduced instance of SAT such that every clause evaluates to TRUE.

   We assume here that a valid 3-colouring exists for our problem. Our approach is to convert it to a solution (i.e., a set of truth assignments) of the reduced SAT problem, and prove that all clauses evaluate to TRUE. Given a valid 3-colouring, we construct a solution to the SAT problem as follows: if cell (i, j) is red, then we set $X_{i,j,r}$ to TRUE and $X_{i,j,b}$ and $X_{i,j,g}$ to FALSE. Similarly, if it's blue we set $X_{i,j,b}$ TRUE and the others to FALSE, and if it's green then we set $X_{i,j,g}$ to TRUE and the others to FALSE. We will now go through all the clauses added in our reduction and show that each evaluates to TRUE.

   All the clauses added in Clauses Step 1 will clearly be true, as every $X_{i,j}$ has a colour assigned (meaning that one literal is true). Similarly, the clauses added in Step 2 will be true, as only one $X_{i,j,c}$ value is set to true.

   Now consider the clauses added in Step 3a. Because our 3-colouring is assumed to be valid, the cells at position (i, j) and $(i+1, j)$ can't be the same colour. This in turn means that at least one of these

cells is not red, and at least one is not blue, and at least one is not green. So all three clauses in this step will evaluate to True. The clauses added in Step 3b will also evaluate to True, by identical reasoning.

Finally, all clauses in step 4 evaluate to True, because a valid solution to Grid-3-colour must not changes the colours of the pre-assigned squares.

5. **[4 marks]** Explain briefly why, if there is a why to assign values to the variables in SAT such that every clause evaluates to TRUE, then the grid in the Grid-3-colour input has a valid 3-colouring. Show how you would determine the colour of each cell.

   The obvious way to construct the 3-colouring is to take the values assigned to the variables in SAT and "translate" them to the corresponding cell colours. But, in order to do that we have to prove that the values assigned to SAT are in the correct form for us to "translate" – in our particular case, that means that every cell must have exactly one colour assigned to it (otherwise, we will need to do some extra processing to the truth variables to get a valid 3-colouring). Specifically, we would need $X_{i,j,c}$ to be true for **exactly one** c, for every i, j.

   Fortunately, we can show this is the case by Clauses Steps 1 and 2. In order for all the Step 1 clauses to evaluate to True, at least one of $X_{i,j,c}$ must be True for every i, j. And in order for all the Step 2 clauses to evaluate to True, you cannot have $X_{i,j,c}$ be True for more than one c. Thus, we have shown that our SAT solution translates to exactly one "True" colour per cell, so we can say: for all i, j, if $X_{i,j,r}$ is True then we colour cell (i, j) red, and if $X_{i,j,b}$ is True then we colour cell (i, j) blue, and if $X_{i,j,g}$ is True then we colour cell (i, j) green.

   From here we need to show that no neighbouring cells are the same colour, and we haven't changed any of the colours of the pre-assigned cells. The 3a clauses tell us that $X_{i,j}$ and $X_{i+1,j}$ must have different c values for which the variables are true (if they were both true for the same c value, the $(\overline{X}_{i,j,c} \vee \overline{X}_{i+1,j,c})$ clause would evaluate to False). This means that no horizontal neighbours have the same colour value assigned. Identical reasoning on the 3b clauses tells us that no vertical neighbours have the same colour value assigned. Thus, if all the Step 3 clauses are True, no cell of the Grid-3-colour solution has the same colour as its neighbour. Finally, the colours of the pre-assigned cells can't have been changed, because in Step 4 we added clauses that can only be True if $X_{i,j,c}$ is True for the correct colour of a pre-assigned cell (i, j).