Please enter your name and uID below.

Name: Qianlang Chen
uID: u1172983

Submission notes

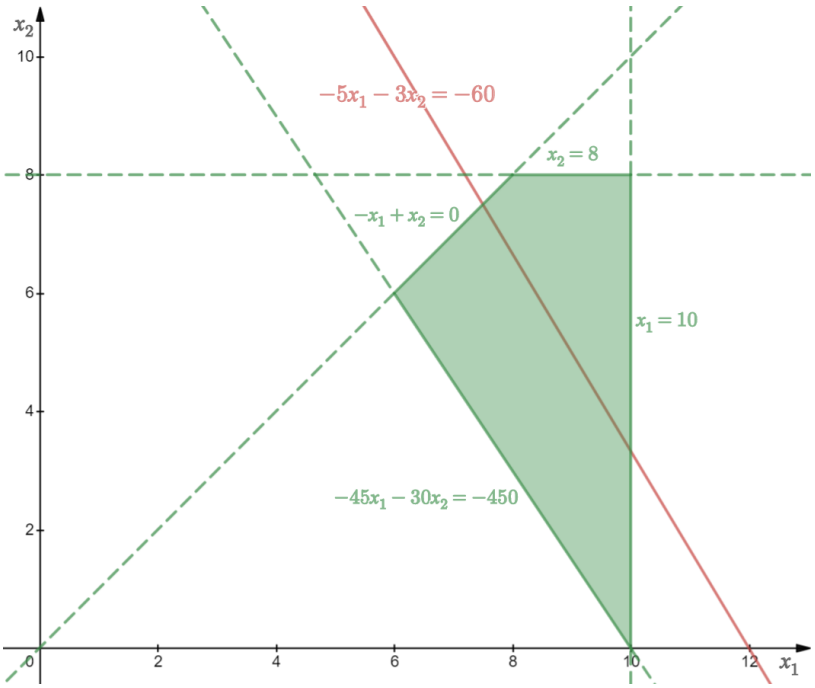
- Due at 11:59 pm on Friday, October 23.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Choosing Candy)

   Let x_1 and x_2 respectively represent the number of bags of Good & Plenty and Necco Wafers the professor should buy. Now, we can build a linear program (in canonical form) for this problem:

$$\begin{aligned} \text{maximize} \quad & -5x_1 - 3x_2 \\ \text{subjective to} \quad & -45x_1 - 30x_2 \leq -450 \\ & x_1 \leq 10 \\ & x_2 \leq 8 \\ & -x_1 + x_2 \leq 0 \\ & x_1, x_2 \geq 0 \end{aligned}$$

   This plot shows the feasible region for this linear program (in green shadow) and the line representing "spending 60 dollars" (in red):



   Since the optimal solution in this case must occur on one of the corners of the feasible region, let's calculate the objective function value at each corner:

x_1	x_2	Objective function value
10	0	$-5 \times 10 - 3 \times 0 = -50$
10	8	$-5 \times 10 - 3 \times 8 = -74$
8	8	$-5 \times 8 - 3 \times 8 = -64$
6	6	$-5 \times 6 - 3 \times 6 = -48$

   Since we turned the objective function into a maximizing function by taking the opposite of the expression, the output values are all opposite as well. The table tells us that -48 is the optimal solution to the above linear program, and the corresponding spending is 48 dollars for the professor, which is done by buying 6 bags of Necco Wafers and 6 bags of Good & Plenty.

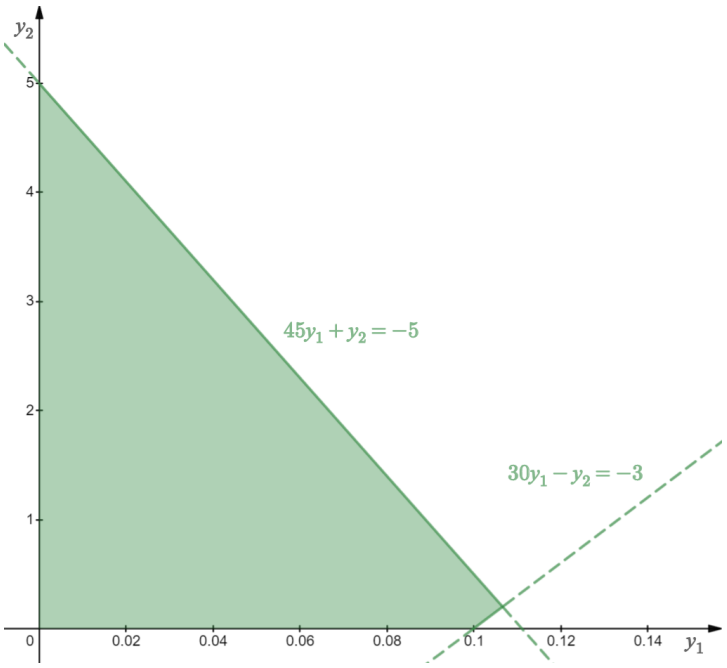
2. (LP Duality)

It turned out that the optimal solution occurred at buying 6 bags of each, which is located at the lower left corner of the plot. Removing the shelf-constraints would only extend the region forever to the top and to the right, and since this is a linear program, any point in the extended region cannot possibly do better than this optimal solution located at lower-left, meaning that the current optimal solution would still be an optimal solution in the new linear program.

After removing the shelf-constraints, let's create the dual of the original linear program, which should have same optimal solution:

$$\begin{aligned} \text{minimize} \quad & -450y_1 \\ \text{subjective to} \quad & -45y_1 - y_2 \geq -5 \\ & -30y_1 + y_2 \geq -3 \\ & y_1, y_2 \geq 0 \end{aligned}$$

The feasible region of the above linear program is as follows:



Again, since the feasible region is bounded, the optimal solution must occur at one of its corners:

y_1	y_2	Objective function value
0	0	$-450 \times 0 = 0$
$\frac{1}{10}$	0	$-450 \times \frac{1}{10} = -45$
$\frac{8}{75}$	$\frac{1}{5}$	$-450 \times \frac{8}{75} = -48$
0	5	$-450 \times 0 = 0$

As seen from the table, the optimal solution of this linear program is also -48.

3. (Complement Connectivity)

Proof. Let some disconnected graph $G = (V, E)$ be given. Let the complement of G be called $\bar{G} = (V, [V^2] \setminus E)$, and let $\bar{E} = [V^2] \setminus E$. Note that V must have at least two vertices because otherwise G wouldn't be disconnected anymore. So, let any two vertices $u, v \in V$ be given. One of the following cases about u and v must be true:

- **Case 1,** u and v are in the same connected component in G. Since G is disconnected, G must have some other connected component different from the one containing u and v. Let some vertex w from the other component be given. Since w is in a different component than both u and v, we know that $uw \notin E$ and $vw \notin E$, meaning that $uw, vw \in \bar{E}$, implying that there's a walk from u to v in $\bar{G}$ and that u and v are in the same component in $\bar{G}$.
- **Case 2,** u and v are not in the same connected component in G, so we know that $uv \notin E$. This immediately implies that $uv \in \bar{E}$, which means that there's a walk from u to v in $\bar{G}$ and that u and v are in the same component in $\bar{G}$.

Now, we've shown that u and v are always in the same connected component in $\bar{G}$. Since $\bar{G}$ and G share the same vertex set, $\bar{G}$ can only have one connected component and is therefore connected. □

4. (Greedy Coloring, Part 1)

(a) Proof. Let's assume that the algorithm `GreedyColoring` uses more than $\Delta+1$ colors for some graph G, where Δ is the highest degree in G. Now, let's see why this cannot happen.

Apart from the initialization of the array `color` at line 4, the algorithm sets the color for each vertex i at line 9, where i gets the color equal to the minimum value in the set V. We know that V was defined to be $S \setminus C$, where C is the set of colors that i's neighbors have and S is the constant set $\{1, 2, \cdots, n\}$. If the algorithm uses more than $\Delta + 1$ colors, there must be some vertex i that's got assigned a color more than $\Delta + 1$ since 1 is the smallest color the algorithm uses (1 is the smallest element in S). By the assignment process we've just described, it means that the minimum value in $S \setminus C$ is greater than $\Delta + 1$, which can only happen when $C \supseteq \{1, 2, \cdots, \Delta + 1\}$. In other words, i has neighbors who are colored $\{1, 2, \cdots, \Delta + 1\}$, but this would mean that i has at least $\Delta + 1$ neighbors, which would disagree with the fact that Δ was the highest degree in G. Such vertex i cannot exist, and such graph G cannot exist, so our assumption in the beginning must've been wrong. The algorithm `GreedyColoring` always uses at most $\Delta + 1$ colors for any graph. □

(b) After all, when assigning a color for a vertex i, we're only interested in the minimum value in $S \setminus C$, which is the smallest color that hasn't appeared within i's neighbors. Since such smallest color can only be at most $(\deg(i) + 1)$, we can use an array of this size to mark the appeared colors and loop again at the end to find the smallest unmarked color:

Algorithm: `GreedyColoring`

```
   Input: set<int> A[1..n]
   Output: int[1..n]
 1 int color[1..n] ← filled with 0
 6 for int i from 1 to n do
 7     int deg ← length of A[i]
 8     bool mark[1..deg] ← filled with false
 9     for each int nei in A[i] do
10         if 1 ≤ color[nei] ≤ deg then
11             mark[color[nei]] ← true
12     int min ← 1
13     while min ≤ deg and mark[min] do
14         min ← min + 1
15     color[i] ← min
16 return color
```

(c) First of all, it's easy to see that the initialization of the array `color` (lines 3-5 from the original pseudocode) takes $O(V)$ time. Now, the main for-loop (lines 6-10) iterates exactly V times, and each iteration takes an additional $O(\deg(i))$ for vertex i; in total, the main for-loop takes $O(V) + O(\deg(1)) + O(\deg(2)) + \cdots + O(\deg(V))$. Since we know for a fact that the degrees of any (undirected) graph sum to twice the number of edges, the main for-loop will take $O(V + 2 \cdot E) = O(V + E)$ time. Therefore, the entire algorithm takes $O(V + E)$.

5. (Greedy Coloring, Part 2)

(a) Let's introduce the following strategy for building a bipartite graph G_n with an even number of vertices $n \geq 2$:

- Create n vertices named $1, 2, \cdots, n$.
- For each odd vertex j from 3 up to $n-1$, create edges that connect j to all even vertices up to $j-1$. (For example, connect vertex 7 to vertices 2, 4, and 6.)
- For each even vertex k from 4 up to n, create edges that connect k to all odd vertices up to $k-3$. (For example, connect vertex 8 to vertices 1, 3, and 5.)

It's easy to see that this strategy will create a bipartite graph, with the even and odd numbers being the two partite sets, since we only create edges between vertices with different parities.

Now, let's see why a graph $G_n = (V_n, E_n)$ created by this strategy makes `GreedyColoring` use $n/2$ colors:

- **Claim.** For any even number k where $2 \leq k \leq n$, the algorithm `GreedyColoring` uses the colors $1..(k/2)$ for all odd vertices up to $k-1$ and for all even vertices up to k.
- **Proof.** Let's induct on k:

  Base case: Since the algorithm colors in vertices in numerical order, the vertices 1 and 2 will both get the color 1 because there isn't an edge between vertices 1 and 2. This shows that the claim holds for $k = 2$.

  Inductive Step: Assume that the claim holds for some $k = i$ (even), meaning that all odd vertices up to $i-1$ have $i/2$ different colors and similarly for all even vertices up to i. The algorithm will have to give the vertex $i+1$ the color $(i/2)+1 = (i+2)/2$ because the vertex $i+1$ is connected to all the even vertices up to i that already have the colors $1..(i/2)$. Similarly, the vertex $i+2$ will also get the color $(i+2)/2$ because it's connected to all the odd vertices up to $i-1$ that already have the colors $1..(i/2)$. This shows that the claim holds for $k = i+2$, which completes the inductive step and shows that the claim is true for $k = n$. □

Since we know that a bipartite graph can be colored using at most two colors, the graph G_n shows that the algorithm `GreedyColoring` doesn't always produce the optimal coloring.

(b) Here's an example graph G:

$$G = (V, E); V = \{1, 2, 3\}, E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$$

(c) No, there are permutations on such graphs where `PermutedGreedy` doesn't produce a suboptimal coloring. (That is, there are situations where `PermutedGreedy` will produce an optimal coloring.) Imagine a bipartite graph created with the strategy from *part (a)* that has at least four vertices. Consider a permutation that lists all odd vertices before the even ones. Since there are no edges between any two vertices from the same partite set, the algorithm will give all the odd vertices the color 1 and then give all the even vertices the color 2 since 1 is the only color appeared within the neighbors of any even vertex. Only two colors are used given this permutation, which is optimal.