CS170 — Fall 2017— Homework 7 Solutions

Jonathan Sun, SID 25020651

0. Who Did You Work With?

Collaborators: Kevin Vo, Aleem Zaki, Jeremy Ou

1. A HeLPful Introduction

(a) The necessary and sufficient conditions on real numbers a and b that make the linear program infeasible do not exist. This is because no matter what a and b are, x and y can both equal 0 and this will always satisfy the constraints.

(b) The necessary and sufficient conditions on real numbers a and b that make the linear program unbounded are if $a < 0$ or $b < 0$. This is because the conditions to maximize $x + y$ and $ax + by \leq 1$ with either $a < 0$ or $b < 0$ would make the region's area infinite. In other words, if the slope of the line for $ax + by \leq 1$ is positive, then there is no maximum for either x or y making the region's area infinite. If the slope of the line is negative, then both x and y will have individual maximum values and so $x + y$ will be bounded. Therefore, the to make the linear program unbounded, either a or b will need to be less than zero to make the slope positive.

(c) The necessary and sufficient conditions on real numbers a and b that allow the linear program to have a unique optimal solution are that $a > 0$, $b > 0$, and $a \neq b$. This allows for the slope of the line for $ax + by \leq 1$ to be negative, which will give individual maximum bounds to both x and y. Furthermore, $a \leq b$ makes the slope not equal to -1, which would cause the maximum value of x to equal the maximum value of y. This would give multiple solutions since $maxx + y$ would all be the same value if the slope is -1. Therefore, we need the slope to be negative and also not equal to -1.

2. TeaOne

(a) The cost of creating a ZestyJuice is $5 \times 0.1 + 1 \times 0.2 + 8 \times 0.01 = 0.78$. Since a ZestyJuice sells for 4.5, the profit of selling one is $4.5 - 0.78 = 3.72$. The cost of creating a MilkTea is $12 \times 0.1 + 16 \times 0.2 = 4.4$. Since a MilkTea sells for 5, the profit of selling one is $5 - 4.4 = 0.6$. With this, I will represent ZestyJuice as z and MilkTea as m. The linear program will be:

$$\max 3.72z + 0.6m$$

$$z \leq 60$$

$$m \leq 40$$

$$0.78z + 4.4m \leq b$$

$$z \geq 0,\ m \geq 0$$

(b)

$$\min 60y_1 + 40y_2 + 6y_3$$

$$y_1 + 0.78y_3 \geq 3.72$$

$$y_2 + 4.4y_3 \geq 0.6$$

$$y_1 \geq 0,\ y_2 \geq 0,\ y_3 \geq 0$$

(c) There are three types of solutions for this part since we can have a budget that is so large that we can afford to make all 60 ZestyJuice and 40 MilkyTea, a limited budget so we maximize ZestyJuices since they make the most profit, and another limited budget where we maximize MilkyTea because they turn a profit.

$$z = \frac{b}{0.78},\ m = 0 \text{ for } b < 46.8$$

$$z = 60,\ m = \frac{b - 46.8}{4.4} \text{ for } 46.8 \leq b \leq 222.8$$

$$z = 60,\ m = 40 \text{ for } b > 222.8$$

3. Mountain pass

(a)

(b)

4. The Hungry Caterpillar

Four-Part Solution:

Main Idea:
The main idea to solve this is to use a graph traversal algorithm to get to k distinct branch points. However, since it costs energy to travel from one branch to the next and we need to start and end at the same point, this means that the caterpillar will be forced to return back to the start point by expending more energy. Since the process of going back on a traversal just costs energy without allowing the caterpillar to eat, this should be avoided as much as possible, except when the caterpillar has to return back to start. Therefore, the caterpillar's path should try to traverse as far down as possible and try to avoid as many backwards movement as possible. To do this, there will be a penalty imposed on each backwards movement the algorithm takes to discourage the caterpillar from traversing backwards in general.

Pseudocode:

```
1  shortestPathsDFS(G, len, s, t):
2    dist(s) = 0
3    # initialize a priority queue starting from s and length of 0
4    Q = stack[(s, 0)]
5    maxCount = 0
6    count = 0
7    minStack = Null
8    while Q is not empty:
9      # u = path and v = dist
10     u, v = Q.pop()
11     if u = t:
12       # 1 + count because we popped one already
13       count += 1
14       if count > maxCount:
15         maxCount = count
16         minStack.add(node)
17       return 1 + count((t, v) left in the queue)
18     for edges(u, v) in E:
19       Q.add((v, val + len(u, v))
20   return minStack
```

Proof of Correctness:
It should be correct because we are just modifying DFS but making it greedy by keeping a counter of the number of back traversals done to discourage the back traversals. □

Run Time:
$O(3n)$.

Justification:
The runtime is $O(3n)$ because the graph traversal will take $O(|V| + |E|)$ time. Since each vertex can either have 2 branches or none, the number of edges will be equal to double the number of

vertices. Since the number of vertices is k and it is upper bounded by n, the number of vertices is upper bounded by n and the number of edges is upper bounded by $2n$. Therefore, the runtime is $O(n + |E|) = O(n + |V - 1|) = O(n + 2|V|) = O(n + 2n) = O(3n)$.

5. Star-shaped polygons in 2D

The idea for this problem is that the point x will be able to see all the points of the polygon, which means that there can be a direct line between each point of the polygon to the point x and none of these points will intersect or be the same equation as each other between each point of the polygon and x. In other words, the linear program needs to show whether there exists a point x that is the intersection of all the lines that go from each point of the polygon to x. This can further "simplified" by having the linear program show whether there exists a point x that is the intersection of all the lines that go from each CORNER of the polygon to x.

Therefore, the linear program can be represented as:

$$\max None$$

$$a_i x + b_i y > c_i \text{ where } i \in \{1, 2, ..., n-1\}$$

6. All Knight-er

Four-Part Solution:

Main Idea:
The main idea is that a knight can only attack in a "L" shape where it needs to move 2 spaces on one axis and 1 space on another axis. So, the idea is that the subproblems will be to find the number of ways we can add more knights given the previous row and the one before that since all the other rows before those two rows will not affect the next row since we just need to care about the previous 2 rows.

Pseudocode:

```
1  function AllKnighter(M, N):
2    state1 = none
3    state2 = none
4    count = 0
5    for all M's:
6      if positionSoFar == 2:
7        count += 1
8        # only one way to put knight when increasing the column size
             by 2
9        return AllKnighterActual(1, 2, state1, state2)
10
11   function AllKnighterActual(sum, positionSoFar, state1, state2):
12     while positionSoFar < N:
13       for all M's:
14         AllKnighterActual(sum, positionSoFar, state0, state1).sum +=
               AllKnighterActual(sum, positionSoFar - 1, state2, state3
             ).sum
15         count += AllKnighterActual(sum, positionSoFar, state0,
             state1).sum
16     return AllKnighterActual(sum, positionSoFar, state0, state1)
17
18   return count
```

Proof of Correctness:
To show that the idea for the pseudocode is correct, I will start with the base case, which is also my first conditional in the for-loop for AllKnighter. This part means that for every size of M, I will increase the count by 1 because in a M by N board, every M by 2 has 1 new valid knight placement assuming that the previous two rows had valid placements since the 2 previous rows limit the new placements that can be be made at the next row. Therefore, to use this idea, I need to be able to keep the previous 2 row's information (which I call state1 and state2) as I am considering the number of possible placements for the new state. Doing so means that for every new row that I look at, I will count the number of possible knight placements in that row and add it to the sum I got when working on the previous row. □

Run Time:
$O(2^{3M}N)$.

Justification:
The runtime is $O(2^{3M}N)$ because the number of times I call AllKnighterActual is $2^{2M}N$ times since the $2M$ comes from the 2 recursive calls for all M in the for loop of all M. Since the while loop has N iterations, there are $2^{2M}N$ subproblems. However, there is also time to compute each occurance of M in the outer function, which takes 2^M time. Therefore, the total runtime is $O(2^{2M}N \times 2^M) = O(2^{3M}N)$.