MPCS 55001 Algorithms Autumn 2023
Homework 8

Zhiwei Cao zhiweic

Collaborators: Student A, Student B

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

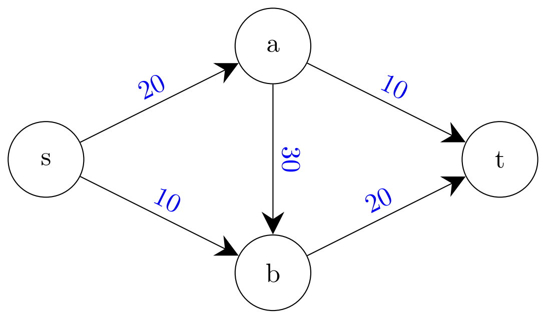
1 Binding Edges (14 points)

Let G be a flow network with integer edge capacities and a positive max flow. An edge in G is *upper-binding* if increasing its capacity by 1 also increases the value of the maximum flow in G. Similarly, an edge is *lower-binding* if decreasing its capacity by 1 also decreases the value of the maximum flow in G.

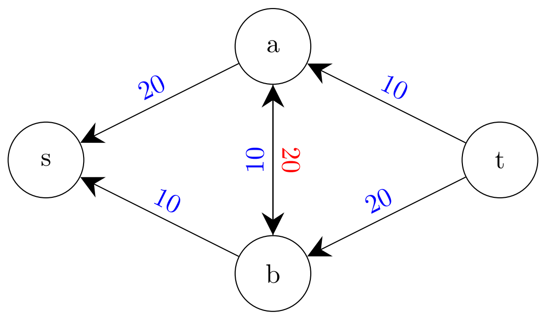
(a) (2 points) Does every flow network G have at least one upper-binding edge? If so, prove it. If not, give a counterexample.

No. By applying the theorem of the flow network: the value of min-cut in the flow network equals the max flow of the flow network. It indicates that there is one edge or edges are fully utilized and the flow equals the capacity by the min-cut max-flow idea. If there is only one min-cut in a flow network, it means that any other cut must be greater (>) than the min-cut by at least 1. In this case, any edge(s) in the min-cut set is the upper-binding edge(s). Is there are multiple min-cuts in a flow network, the only possible case that the flow has at least one upper-binding edge is that the edge is a part of all min-cuts, and the edge is fully utilized. Otherwise, there are no upper-binding edges existing in a flow network.

Counter Example: There are two min-cut sets, but if we select any edge from them and increase the capacity by 1, the general max flow of the network doesn't change. Since it still constricts by the min-cut value.



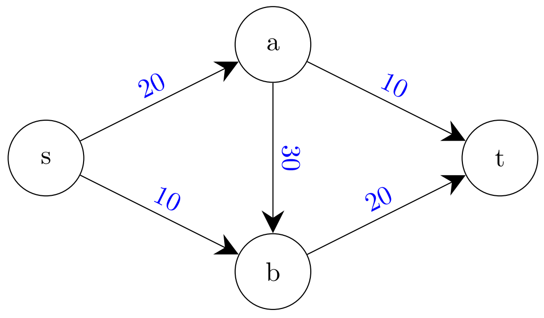
In other words, based on the residual graph of G, no matter which edge increases the capacity by 1, there is no path from s to t.



(b) (2 points) Does every flow network G have at least one lower-binding edge? If so, prove it. If not, give a counterexample.
Yes. As long as a flow network contains a min-cut(s). It must have at least one lower-binding edge. Since the max-flow of the G equals the min-cut of the G, then any edge in the min-cut set can be the lower-binding edge. Since any edge of any min-cut set(s) decreased by 1, which means the value of the min-cut is decreased by 1 as well. If the value of any min-cut of the total min-cut(s) changes, it means there is a new and unique min-cut (smaller by 1) that exists in the flow network. At the same time, the new (smaller) min-cut indicates the new (smaller) max-flow of the flow network.

Recall that an edge is **saturated** under a flow f if $f(e) = c(e)$.

(c) (2 points) Prove or disprove: Given a flow network G, if an edge e is saturated under a max flow f, then e is an upper-binding edge.
Disprove and Counterexample:



Keep using this flow network as a counterexample to disprove this statement. By the definition of a flow network, the max flow of the flow is 30, and there is no augment path from s to t in G'_f. As a result, there are four edges that are saturated which are (s, a), (s, b), (a, t), and (b, t) in this flow. Take any of these four edges, for example (s, a), and add 1 to the capacity, which indicates that there is an edge with a capacity of 20 from a to s and an edge with a capacity of 1 from s to a in G_f assume that the edge (s, a) (or any other edges that saturated in this case) is an upper-binding edge. However, there is no augment path in the G_f from s to t in this case, and the value of max flow doesn't change as well. It constructs the contradiction. Then disprove the statement that "Given a flow network G, if an edge e is saturated under a max flow f, then e is an upper-binding edge."

(d) (2 points) Prove or disprove: Given a flow network G, if an edge e is an upper-binding edge, then e is saturated under any max flow f.
Prove If an edge $e = (u, v)$ is an upper-binding edge, which means that increasing its capacity by 1 also increases the value of the maximum flow in G by definition. Assume that the edge e is not saturated which indicates that the flow of the edge is less than the capacity of the edge. In this case, the flow of the edge e doesn't take all the capacity of the edge. As a result, 1 more capacity still cannot make the flow of the edge e equal the capacity of e, since the flow into the u equals the flow out of u, the flow of the edge e in this case, by the flow conservation. Then the max flow of the network doesn't change. Then it constructs the contradiction. As a result, the edge e is saturated under any max flow f by the contradictions.

(e) (2 points) Write **pseudocode** for an algorithm which takes a flow network G, maximum flow F, and edge e as input and returns a boolean indicating whether e is an upper-binding edge. Your algorithm should run in $O(V + E)$ time.

Algorithm 1 1 (e): IsAUpperBindingEdge

```
1:  function IsAUpperBindingEdge($G$, $F$, $e$)
2:  Input: $G = (V, E, c)$ where $c$ is the capacity of the edge, $F$ is the max flow graph which functions $f$ to access
    the flow value of the edges, and an edge $e = (u, v)$.
3:  Output: Boolean result that if the $e$ is an upper-binding edge.
4:  $residual\_graph \leftarrow$ a new empty graph (u, v, value)
5:  for each vertex v in $F$ do
6:      add v to $residual\_graph.v$
7:  end for
8:  for each edge $e' = (u, v) in F$ do
9:      $flow \leftarrow F.getFlow(e')$ // get the flow of the edge $e$ from maximum flow network F
10:     $capacity \leftarrow G.getCapacity(e')$ // get the capacity of the flow network G
11:     $residual\_value = capacity - flow$
12:     if $flow > 0$ then
13:         add the edge $(v, u, flow)$ with the $flow$ in $residual\_graph$
14:     end if
15:     if $residual\_value > 0$ then
16:         add the edge $(u, v, residual\_value)$ with the $residual\_value$ in $residual\_graph$
17:     end if
18: end for
19: if e.getValue then
20:     // if there is a path from u to v exists on the edge $e = (u, v)$ in the $residual\_graph$
21:     // indicates that there is some capacity is not utilized in the edge,
22:     //no matter how many more capacities are added to the edge.
23:     // It doesn't affect the value of the max flow
24:     return False
25: end if
26: add the edge $(v, u, 1)$ to $residual\_graph$ by attribute of upper-binding edge
27: $ifPath \leftarrow$ applies the BFS to check if there is a path from $s$ (source) to $t$ sink.
28: if $ifPath$ then
29:     return True // there is a new augment path in the new $residual\_graph$
30: else
31:     return False
32: end if
```

(f) (4 points) Write **pseudocode** for an algorithm which takes a flow network G and a maximum flow F as input and computes a list of all upper-binding edges in G in $O(V + E)$ time.

Algorithm 2 1 (f): AllUpperBindingEdge

```
Reference: https://www.cl.cam.ac.uk/teaching/1415/Algorithms/micro7.pdf
function AkkUpperBindingEdge($G$, $F$)
Input: $G = (V, E, c)$ where $c$ is the capacity of the edge, $F$ is the max flow graph which functions $f$ to access the flow value of the edges.
Output: Boolean result that if the $e$ is an upper-binding edge.
$residual\_graph \leftarrow$ a new empty graph (u, v, value)
for each vertex v in $F$ do
    add v to $residual\_graph.v$
end for
for each edge $e' = (u, v) in F$ do
    $flow \leftarrow F.getFlow(e')$ // get the flow of the edge $e$ from maximum flow network F
    $capacity \leftarrow G.getCapacity(e')$ // get the capacity of the flow network G
    $residual\_value = capacity - flow$
    if $flow > 0$ then
        add the edge $(v, u, flow)$ with the $flow$ in $residual\_graph$
    end if
    if $residual\_value > 0$ then
        add the edge $(u, v, residual\_value)$ with the $residual\_value$ in $residual\_graph$
    end if
end for
$res \leftarrow anemptysettostoreallupperbindingedges$
$connet\_to\_source \leftarrow anemptysettostoreallverticesthatarereachablefromsource$
$color \leftarrow BFS(residual\_graph, s)$
for each v in color do
    add v to the set $connet\_to\_source$
end for
$connet\_to\_sink \leftarrow$ an empty set to store all vertices that are reachable from sink
$color \leftarrow BFS(residual\_graph, t)$
for each v in color do
    add v to the set $connet\_to\_sink$
end for
for each edge $e = (u, v)$ in $residual\_graph$ do
    if (u in $connet\_to\_source$ and v in $connet\_to\_sink$) or opposite then
        add the edge e to $res$
    end if
end for
return res
```

2 Final Exam Scheduling (13 points)

MPCS Admin has hired you to write an algorithm to schedule the MPCS final exams. There are n different classes, each of which needs to schedule a final exam in one of k classrooms during one of t different time periods. At most one class's final exam can be scheduled in each classroom during each time period; in addition, classes cannot be split into multiple classrooms or multiple time periods. Furthermore, each exam must be proctored by one of p Ph.D. students. Each Ph.D. student can proctor at most one exam at a time; each Ph.D. student is available for only certain time periods; and no Ph.D. student is allowed proctor more than 3 exams total.

The input to this scheduling problem consists of three arrays:

- An integer array $R[1..n]$, where $R[i]$ is the number of students registered in the ith class.
- An integer array $S[1..k]$, where $S[j]$ is the number of seats in the jth classroom. The ith class's final exam can be held in the jth classroom if and only if $R[i] \leq S[j]$.
- A boolean array $B[1..t, 1..p]$, where $B[\ell, m] =$ `True` if and only if the mth proctor is available during the ℓth time slot.

Give an algorithm that returns a list of n tuples where each tuple contains a class, a classroom, a time period, and a Ph.D. student proctor. If no schedule is possible your algorithm should return `NIL`. Your algorithm should involve constructing a flow network, finding the max flow, and interpreting the max flow. Your algorithm must run in $O(nN^2)$ time where $N = (n + k + t + p)$.

(a) (5 points) Write **pseudocode** for your algorithm.

(b) (4 points) Prove your algorithm is correct. Reminder: you need to prove two directions.
Start from the graph construction: In order to build a flow network for the question, we add a source and a sink into graph G. If the algorithm can find a path from the source to the sink, it indicates there is an exam on schedule. Since the purpose of the question is to assign n classes' exam, the graph contains n classes vertices, and the source connects to them with the capacity of 1 (each class only has one final exam). There are k available classrooms in the university so assign t vertices stands for each classroom. In order to assign a class's final exam to a classroom, we need to ensure that the classroom has enough space to hold students who take the class. As a result, we need to ensure that $R[n'] \leq S[t']$ first, and then connect n' to k' with a capacity of 1 (since each classroom can only hold for one class exam). For each classroom, there may be several different time periods t available for the exam, so assign t vertices as the time periods for each classroom. If the classroom k' is available for some time periods t', then connect k' to $t'(s)$ with the capacity 1 (since each time period can hold one class's exam). Assign p vertices to stand for p Ph.D. student will proctor the exam. Since we need to ensure the Ph.D. p' is available during some time periods t, so we only connect the vertices t' to p' where $B[t', p'] = True$ which the capacity 1 (each Ph.D. can proctor at most one exam at a time). At last, based on the problem description "no Ph.D. student is allowed to proctor more than 3 exams total", so the capacity of the edges from each Ph.D. p' to the sink is 3. The flow network construction is done.
Proof of termination: The capacity of the network is finite, so there is only a finite capacity out of source, $\sum_{v \in V} f(s,v) \leq \sum_{v \in V} c(s,v)$ by the capacity constraint. The capacity value in this case is greater or equal to 1, then $C_f(p) \geq 1$, so the smallest flow increases by one on each iteration. As a result, the flow of the network is increased in discrete increments. Since the capacities are finite and flow increases are discrete, there cannot be an infinite number of augmenting paths, so the algorithm will reach a state where no augmenting path exists in the G'. Then based on the algorithm condition, the algorithm will stop and terminate.
Proof of max flow: Since the algorithm will terminate, it means there is no augmenting path in the flow network. It indicates that every path from the source to the sink is blocked by some edges that are fully utilized or there is no flow that can exist in the residual graph. As a result, if G_f has an augmenting path p, then could add flow from the source to get $f + G_f(p)$ with value $|f'| + C_f(p) > |f|$, so f is not the max flow by the contradiction. Then suppose G_f has no augmenting path. Define a cut $S = \{s\} + \{v\}$ where $v : \exists$ path from s to v in G_f with $C_f(p) > 0$, and $T = V - S$. Then (S, T) is a s-t cut. Then $f(S,T) = \sum u \in S \sum v \in T f(u,v) - \sum v \in T \sum u \in S f(v,u) = \sum u \in S \sum v \in T c(u,v) = c(S,T)$. For an ideal flow network in this question, the min-cut should cut all edges between the source and all classes vertices where S = $\{s\}$ and T = $\{v \in n\}$, because there are n classes need to take final exams and they just need to take once. At last, it is known that $|f| \leq c(S,T)$ for all cuts S, T. $|f| = c(S,T)$ for some cut, which indicates that $|f|$ is the max flow.

Algorithm 3 2 (a): examScheduling

```
function examScheduling($n, k, t, p$,$R[1 \dots n]$, $S[1 \dots k]$,$B[1..t, 1..p]$)
Input: $n$ different classes; $k$ classrooms; $t$ different time periods; $p$ Ph.D. students; An integer array $R[1..n]$, where $R[i]$ is the number of students registered in the $i$th class; An integer array $S[1..k]$, where $S[j]$ is the number of seats in the $j$th classroom; A boolean array $B[1..t, 1..p]$, where $B[\ell, m] = \texttt{True}$ if and only if the $m$th proctor is available during the $\ell$th time slot.
Output: returns a list of $n$ tuples where each tuple contains a class, a classroom, a time period, and a Ph.D. student proctor.
$G \leftarrow theflownetwork$
// consider $n'$ classes, $k'$ classrooms, $t'$ time periods, $p'$ Ph.D are vertex
add $source$, $sink$ to $G$
// connect source to all classes
for each vertex $n$ in $n'$ do
    add $n$ to $G$
end for
for each vertex $n$ in $G$ do
    add $n$ to $G.adj[source]$
    $c(source, n) \leftarrow 1$
end for
// connect class to classroom only $c(n, k) \leftarrow 1$
for each vertex $k$ in $k'$ do
    add $k$ to $k$
end for
for each vertex $n$ in $G$ do
    for each vertex $k$ in $G$ do
        if $R[n] \leq S[k]$ then
            add $k$ to $G.adj[n]$
            $c(n, k) \leftarrow 1$
        end if
    end for
end for
// connect classroom to time $t$, if the classroom is available during time $t$
for each vertex $t$ in $t'$ do
    add $t$ to $G$
end for
for each vertex $k$ in $G$ do
    for each vertex $t$ in $G$ do
        if classroom $k$ is available during $t$ then
            add $t$ to $G.adj[k]$
            $c(k, t) \leftarrow 1$
        end if
    end for
end for
// connect time $t$ with Ph.D $p$ if $B[\ell, m] = \texttt{True}$
for each vertex $p$ in $p'$ do
    add $p$ to $G$
end for
for each vertex $t$ in $G$ do
    for each vertex $p$ in $G$ do
        if $B[t, p] = \texttt{True}$ then
            add $p$ to $G.adj[t]$
            $c(t, p) \leftarrow 1$
        end if
    end for
end for
```

Algorithm 4 2 (a): examScheduling

```
// connect all p to sink with **capacity 3**
for each vertex $p$ in $G$ do
    add $sink$ to $G.adj[p]$
    $c(p, sink) \leftarrow 3$
end for
$schedule\_res = []$
// apply Ford-Fulkerson Algorithm
// Reference CLRS p.724
for each edge $e = (u, v) \in G.E$ do
    $(u, v).f \leftarrow 0$
end for
while there exist a path $p$ from $source$ to $sink$ in the residual network $G_f$ do
    $C_f(p) \leftarrow \min(C_f(u, v) : (u, v)$ is in p)
    add the vertices of the augmenting path p as a tuple to the $schedule\_res$ with the order of [class, classroom, period, Ph.d] (without source and sink)
    for each edge $(u, v)$ in $p$ do
        if $(u, v) \in E$ then
            $(u, v).f \leftarrow (u, v).f + C_f(p)$
        else
            $(v, u).f \leftarrow (v, u).f - C_f(p)$
        end if
    end for
end while
return $schedule\_res$
```

(c) (1 point) Analyze the time complexity of your algorithm and argue that it runs in $O(nN^2)$ time.
Running time: since the source connects to each class with a capacity of 1, then the max flow $|f|$ cannot exceed the number of classes which is n. Then there are at most $n + nk + kt + tp + p$ edges in $G.E$ which is less than N^2. Then the running time is $O(nN^2)$

Sad news! Based on your algorithm's output, MPCS Admin realizes it is able to schedule a final for every MPCS course **except MPCS 55001**. Determined to solve this problem, Professor Brady wants to allow one Ph.D. student to proctor up to 4 exams instead of 3.

(d) (3 points) Describe (2–3 sentences is fine) an algorithm to determine whether there exists a Ph.D. student such that permitting this Ph.D. student to proctor 4 exams would allow MPCS Admin to schedule a final for every MPCS course. If so, the algorithm should return the index (between 1 and t) of the lucky Ph.D. student selected to proctor 4 exams. Otherwise, return `NIL`.
Naive Approach: Add one iteration of p Ph.D. students. In each iteration, add 1 capacity value from p to sink, so the capacity from current p to sink will be 4 in total (means the current Ph.D. student will proctor 4 exams). If the size of $schedule\_res == n$ (from question a), then return the index of the p. If cannot find such a Ph.D. student, then return NIL.
Better Approach: apply the idea of finding the all upper binding edges of the flow network in this question. After that, check each edge $e = (u, v)$ in the upper binding edge list, if $u \in$ Ph.D. vertices and $v \in$ sink, then find the Ph.D. and return.

3 Paintball Strategies (15 points)

Konstantinos and Andrew have decided to take you out for paintball as a reward for all of your hard work! Konstantinos is the Blue squad leader and Andrew is the Red squad leader. Each of them has devised 3 tactics, and being perfect paintball strategists, they know the probability of victory for each pair (a blue tactic, a red tactic). Every pair of tactics gives the Blue squad either an increase or a decrease in their probability of winning. The Blue squad has the following tactics (B_1, B_2, and B_3):

- B_1: Hold back and send two members to flank
- B_2: Rush the Red squad
- B_3: Capture the bunker to the right

The Red squad has devised three tactics (R_1, R_2, and R_3):

- R_1: Post a sniper on their outlook tower and defend around it
- R_2: Rush the Blue squad
- R_3: Flank from the left

Before the tactics are chosen, both squads win with equal probability. Table 1 shows the change in the Blue squad's winning probability (referred to as Blue's advantage *adv*) for each pair of tactics:

Red / Blue	R_1	R_2	R_3
B_1	-0.2	+0.3	-0.1
B_2	+0.4	+0.0	-0.2
B_3	+0.25	-0.2	+0.0

Table 1: Blue squad's win probability change for each strategy pair

(a) (1 point) Use Table 1 to show that if the Blue squad always picks the same tactic, the Red squad can always pick a tactic to put the Blue squad at a disadvantage (i.e., having probability less than 50% to win).
If the Blue Squad chooses Tactics 1, the Red Squad could always choose Tactics 1 to decrease the Blue's winning probability by 0.2 each round, then the winning probability of the Blue will be 40%, less than 50%. If the Blue Squad chooses Tactics 2, the Red Squad could always choose Tactics 3 to decrease the Blue's winning probability by 0.2 each round, then the winning probability of the Blue will be 40%, less than 50%. If the Blue Squad chooses Tactics 3, the Red Squad could always choose Tactics 2 to decrease the Blue's winning probability by 0.2 each round, and then the winning probability of the Blue will be 40%, less than 50%.

(b) (2 points) Suppose that the Red squad and the Blue squad choose their tactics based on the probabilities in Table 2. For example, the Blue squad chooses B_1 with probability 0.3. Compute the probability of the Blue squad winning.// Blue Winning Prob = P(B1)*P(R1)*adv(B1,R1) + P(B1)*P(R2)*adv(B1,R2) + P(B1)*P(R3)*adv(B1,R3) + P(B2)*P(R1)*adv(B2,R1) + P(B2)*P(R2)*adv(B2,R2) + P(B2)*P(R3)*adv(B2,R3) + P(B3)*P(R1)*adv(B3,R1) + P(B3)*P(R2)*adv(B3,R2) + P(B3)*P(R3)*adv(B3,R3) + 0.5 = 0.4295. // Since there are only two teams, each team has a 50% chance to win. Based on the 50%, calculate the winning probability with the different tactics.

Blue squad		Red squad	
B_1	0.3	R_1	0.1
B_2	0.6	R_2	0.2
B_3	0.1	R_3	0.7

Table 2: Probabilities for each squad to choose each tactic

Konstantinos believes that the Blue squad can do better, so he wants you to choose new probabilities for each tactic. Your task is to determine $p_{B_1}, p_{B_2}, p_{B_3}$ that maximizes the advantage *adv* (defined above) regardless of which tactic the Red squad chooses.

(c) (4 points) Write a Linear Program that determines how to choose $p_{B_1}, p_{B_2}, p_{B_3}$ to maximize *adv*.

Hint: In order to maximize *adv* regardless of what tactic the Red squad chooses, your linear program should somehow consider the expected advantage of your strategy ($p_{B_1}, p_{B_2}, p_{B_3}$) against each of the Red squad's tactics (R_1, R_2, R_3). You need to choose $p_{B_1}, p_{B_2}, p_{B_3}$ to maximize the minimum of the three expected advantages associated with the three Red squad tactics.

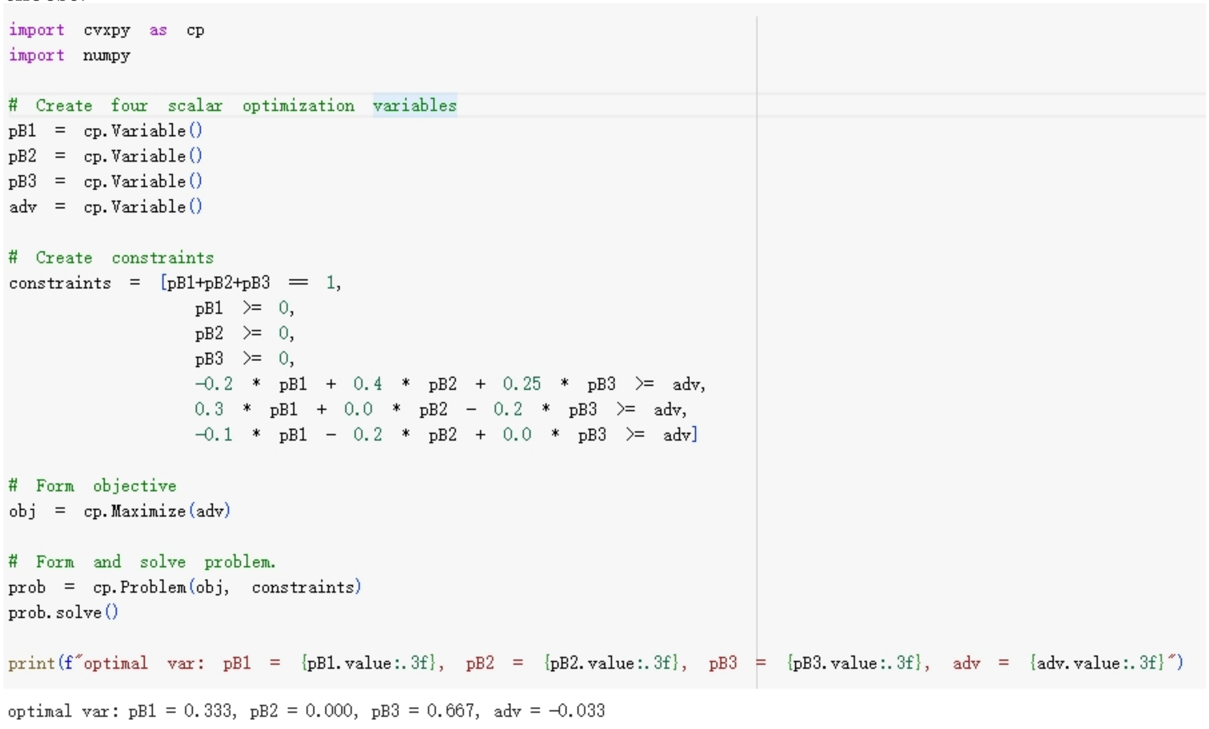
objective max *adv*

Constraints

$adv(B1, R1) * p_{B_1} + adv(B2, R1) * p_{B_2} + adv(B3, R1) * p_{B_3} \geq adv$
$adv(B1, R2) * p_{B_1} + adv(B2, R2) * p_{B_2} + adv(B3, R2) * p_{B_3} \geq adv$
$adv(B1, R3) * p_{B_1} + adv(B2, R3) * p_{B_2} + adv(B3, R3) * p_{B_3} \geq adv$
$p_{B_1} + p_{B_2} + p_{B_3} = 1$
$0 \leq p_{B_1}, p_{B_2}, p_{B_3}$

(d) (2 points) There are various LP solvers online. Pick one of them, and have it solve your linear program from (c). You might consider trying this site first, but feel free to try other options. You can also use python's CVXPY library if you want.

Solve the LP, and state the largest possible *adv*, and the probabilities $p_{B_1}, p_{B_2}, p_{B_3}$ that Konstantinos should choose.



```
import cvxpy as cp
import numpy

# Create four scalar optimization variables
pB1 = cp.Variable()
pB2 = cp.Variable()
pB3 = cp.Variable()
adv = cp.Variable()

# Create constraints
constraints = [pB1+pB2+pB3 == 1,
               pB1 >= 0,
               pB2 >= 0,
               pB3 >= 0,
               -0.2 * pB1 + 0.4 * pB2 + 0.25 * pB3 >= adv,
               0.3 * pB1 + 0.0 * pB2 - 0.2 * pB3 >= adv,
               -0.1 * pB1 - 0.2 * pB2 + 0.0 * pB3 >= adv]

# Form objective
obj = cp.Maximize(adv)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve()

print(f"optimal var: pB1 = {pB1.value:.3f}, pB2 = {pB2.value:.3f}, pB3 = {pB3.value:.3f}, adv = {adv.value:.3f}")
```

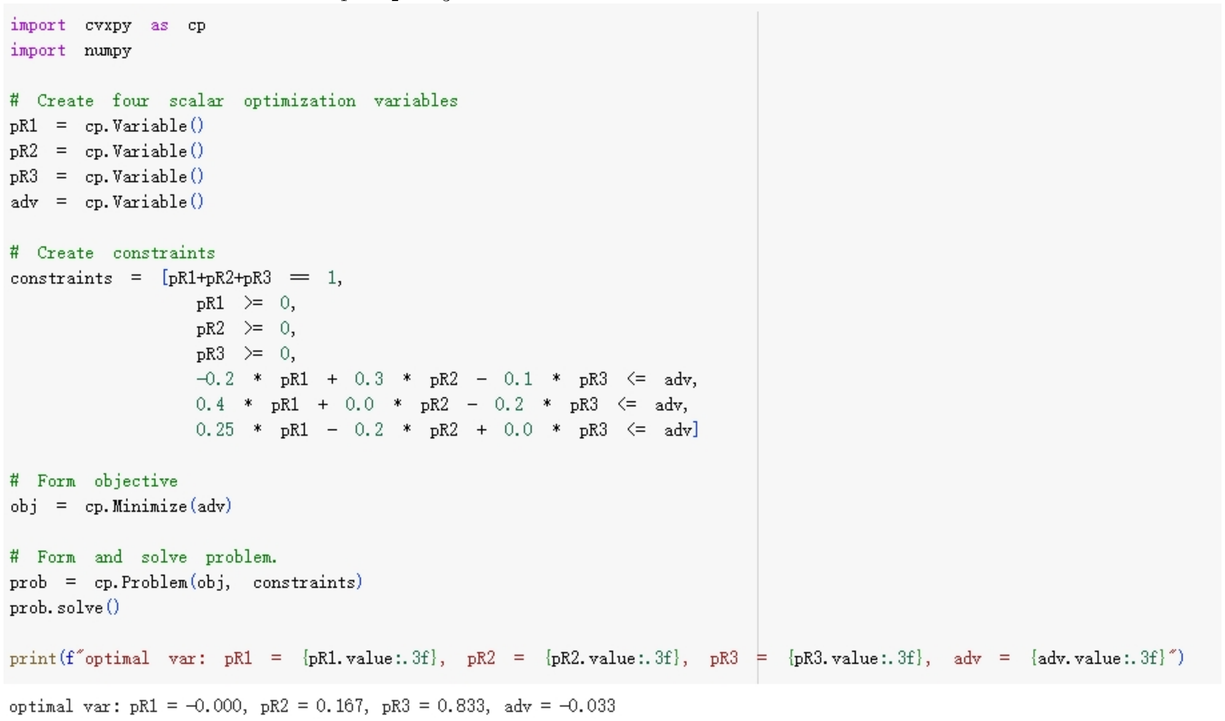
```
optimal var: pB1 = 0.333, pB2 = 0.000, pB3 = 0.667, adv = -0.033
```

$p_{B_1} = 0.333 = 1/3, p_{B_2} = 0.000, p_{B_3} = 0.667 = 2/3, adv = -0.033$

For the next round, you have been transferred to Red squad. After Andrew has seen his winning chances significantly reduced from last time, he asks that you optimize his strategy selection as well.

(e) (4 points) Write a Linear Program to determine the probabilities $q_{R_1}, q_{R_2}, q_{R_3}$ to **minimize** the advantage adv of the Blue squad regardless of which tactic the Blue squad chooses.
objective min adv
Constraints
$adv(B1, R1) * p_{R_1} + adv(B1, R2) * p_{R_2} + adv(B1, R3) * p_{R_3} \leq adv$
$adv(B2, R1) * p_{R_1} + adv(B2, R2) * p_{R_2} + adv(B2, R3) * p_{R_3} \leq adv$
$adv(B3, R1) * p_{R_1} + adv(B3, R2) * p_{R_2} + adv(B3, R3) * p_{R_3} \leq adv$
$p_{R_1} + p_{R_2} + p_{R_3} = 1$
$0 \leq p_{R_1}, p_{R_2}, p_{R_3}$

(f) (2 points) Use an online LP solver or CVXPY to solve the linear program from (e). State the lowest possible adv, and the probabilities $q_{R_1}, q_{R_2}, q_{R_3}$ that Andrew should choose.



```
import cvxpy as cp
import numpy

# Create four scalar optimization variables
pR1 = cp.Variable()
pR2 = cp.Variable()
pR3 = cp.Variable()
adv = cp.Variable()

# Create constraints
constraints = [pR1+pR2+pR3 == 1,
               pR1 >= 0,
               pR2 >= 0,
               pR3 >= 0,
               -0.2 * pR1 + 0.3 * pR2 - 0.1 * pR3 <= adv,
               0.4 * pR1 + 0.0 * pR2 - 0.2 * pR3 <= adv,
               0.25 * pR1 - 0.2 * pR2 + 0.0 * pR3 <= adv]

# Form objective
obj = cp.Minimize(adv)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve()

print(f"optimal var: pR1 = {pR1.value:.3f}, pR2 = {pR2.value:.3f}, pR3 = {pR3.value:.3f}, adv = {adv.value:.3f}")
```

```
optimal var: pR1 = -0.000, pR2 = 0.167, pR3 = 0.833, adv = -0.033
```

$p_{R_1} = 0.000, p_{R_2} = 0.167, p_{R_3} = 0.833, adv = -0.033$

4 Programming: Linear Programs in Python and CVXPY (20 points)

Follow this Google Colab template link for an introduction to Linear Programming in Python. **Do not attempt to edit the linked file.** Create your own copy instead by clicking File → Save a copy in Drive.

When you have completed the notebook, download the .ipynb file. **Important**: you must submit this problem separately from .pdf file which contains the rest of your theory solutions. Submit the completed .ipynb file to the Gradescope assignment named “HW 8 - Problem 4 Notebook”.