Elijah Chou

CS 526

Michelangelo Grigni

9/24/2023

HW 3

1. A) I should choose p1 = 0.15789473684210537, p2 = 0.21052631578947376, p3 = 0.6315789473684208, and p4 = 0.0. Choosing these values, P = 1.4210526315789473.  
     
   B) My adversary should choose q1 = 0.21052631578947378, q2 = 0.3157894736842105, q3 = 0.4736842105263156, and q4 = 0.0. Choosing these values, Q = 1.4210526315789473.  
     
   Below is the payoff matrix I calculated based on the question, the explicit LPs for both A and B, the code I used to calculate these values, and also the output I got from running the code.  
     
   A close-up of a whiteboard with equations

   Description automatically generated  
     
   A white paper with blue writing on it

   Description automatically generated

Code:  
  
from ortools.linear\_solver import pywraplp

def LinearProgrammingProblemA():

solver = pywraplp.Solver.CreateSolver("GLOP")

if not solver:

return

p1 = solver.NumVar(0, 1, "p1")

p2 = solver.NumVar(0, 1, "p2")

p3 = solver.NumVar(0, 1, "p3")

p4 = solver.NumVar(0, 1, "p4")

P = solver.NumVar(-solver.Infinity(), solver.Infinity(), "P")

print("Number of variables =", solver.NumVariables())

solver.Add(-3 \* p1 - (3/2) \* p2 - p3 - p4 + P >= 0)

solver.Add(-p1 - 2 \* p2 - (4/3) \* p3 - (4/3) \* p4 + P >= 0)

solver.Add(-p1 - p2 - (5/3) \* p3 - (5/3) \* p4 + P >= 0)

solver.Add(-p1 - p2 - p3 - 2 \* p4 + P >= 0)

solver.Add(p1 + p2 + p3 + p4 == 1)

print("Number of constraints =", solver.NumConstraints())

# Objective function: 3x + 4y.

solver.Minimize(P)

# Solve the system.

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:

print("Solution:")

print("Objective value =", solver.Objective().Value())

print("p1 =", p1.solution\_value())

print("p2 =", p2.solution\_value())

print("p3 =", p3.solution\_value())

print("p4 =", p4.solution\_value())

else:

print("The problem does not have an optimal solution.")

print("\nAdvanced usage:")

print("Problem solved in %f milliseconds" % solver.wall\_time())

print("Problem solved in %d iterations" % solver.iterations())

def LinearProgrammingProblemB():

solver = pywraplp.Solver.CreateSolver("GLOP")

if not solver:

return

q1 = solver.NumVar(0, 1, "q1")

q2 = solver.NumVar(0, 1, "q2")

q3 = solver.NumVar(0, 1, "q3")

q4 = solver.NumVar(0, 1, "q4")

Q = solver.NumVar(-solver.Infinity(), solver.Infinity(), "Q")

print("Number of variables =", solver.NumVariables())

solver.Add(-(3 \* q1 + q2 + q3 + q4) + Q <= 0)

# Constraint 1: 3x - y >= 0.

solver.Add(-(3/2 \* q1 + 2 \* q2 + q3 + q4) + Q <= 0)

# Constraint 2: x - y <= 2.

solver.Add(-(q1 + (4/3) \* q2 + (5/3) \* q3 + q4) + Q <= 0)

solver.Add(-(q1 + (4/3) \* q2 + (5/3) \* q3 + 2 \* q4) + Q <= 0)

solver.Add(q1 + q2 + q3 + q4 == 1)

print("Number of constraints =", solver.NumConstraints())

# Objective function: 3x + 4y.

solver.Maximize(Q)

# Solve the system.

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:

print("Solution:")

print("Objective value =", solver.Objective().Value())

print("q1 =", q1.solution\_value())

print("q2 =", q2.solution\_value())

print("q3 =", q3.solution\_value())

print("q4 =", q4.solution\_value())

else:

print("The problem does not have an optimal solution.")

print("\nAdvanced usage:")

print("Problem solved in %f milliseconds" % solver.wall\_time())

print("Problem solved in %d iterations" % solver.iterations())

LinearProgrammingProblemA()

LinearProgrammingProblemB()  
  
Output:   
A computer screen shot of a black screen with white text

Description automatically generated

1. First, let us start off with the setup discussed in lecture6.pdf. In my algorithm, I will include every set S in the set cover with probability xS, independently; then the expected cost is E[cost] = . This partial cover also has the property that for every single element a, Pr[a not covered] < 1/e.  
     
   Now we want a random algorithm, for fixed ε>0 and large enough n, that can find a cover with cost at most (1+ε)(ln n)OPT.  
     
   Given the random procedure above, we can repeat it D times, and take the union of all the selected sets to get a random cover C. Let cost(C) denote the total cost of the sets in C. We then have the following inequalities:

* E[cost(C)] ≤ D·OPTf
* For any single element a (one of the n elements), Pr[a is not covered] < (1/e)^D

In the lecture notes, D was defined to be d ln n, but for this problem we will define it as D = c + ln n, where c depends on ε.  
  
To get a complete set cover C’, we should independently select c + ln n such subcollections so that Pr[a is not covered by C’] ≤ (1/e)^(c + ln n) = (1/e)^c \* (1/n). Then if we define c = ln(4/ε), we get that Pr[a is not covered by C’] ≤ (1/e)^(ln (4/ε)) \* (1/n) = 4/ε \* 1/n = ε/4n.  
  
Summing over all elements a, we get Pr[C’ is not a valid set cover] ≤ n \* ε/4n = ε/4. By applying union bounds, we get that Pr[C’ covers every element] ≥ 1- (ε/4), for an expected cost of E[cost(C’)] = (c + ln n)\* OPTf = (ln n – ln (ε/4)) \* OPTf = ln (4n/ε) \* OPTf.  
  
Earlier, we defined D as c + ln n, which we then further defined c as ln (4/ε). We can further simplify D = ln (4/ε) + ln n = ln (4n/ε). To simplify the bounding of the expected cost, we can derive D ≤ (1 + ε/4) \* ln n, through which we can also bound the expected cost E[cost(C’)] ≤ (1 + ε/4) \* ln n \* OPTf.  
  
Since (1 + ε/2) \* (1 + ε/4) < (1 + ε), we can bound the probability of the cost of C’ being greater than (1 + ε) \* E[cost(C’)] as such:  
  
Pr[cost(C’) > (1 + ε/2) \* E[cost(C’)] < Pr[cost(C’) > (1 + ε) \* E[cost(C’)] < 1/(1 + ε/2) < 1 – (ε/3).

By union bounding, we get that the probability of the two “bad” events is ≤ ε/4 + 1-(ε/3) = 1 - (ε/12), and therefore the Pr[success] ≥ ε/12.  
  
With these in mind, if we repeat the process 12/ε times, we should eventually succeed with a good enough C’ with probability of at least ½. This is because   
Pr[never succeeds] ≤ (1 - ε/12)^(12/ε) ≤ (e^(-ε/12))^(12/ε) ≤ (e^(-ε/12) \* (ε/12) ≤ e^-1 < ½. We thus get Pr[succeeds at least once] > ½.  
  
We know that this should still run in polynomial time because we know we can test the cover C’ is good enough in polynomial time. It should be poly(n, m, 1/ε) total time. This should be true for small enough ε > 0.

1. A) To verify that the function f is submodular, we need to show that for any two subsets Si and Sj where i,j ∈ {1, 2, ..., m}, the following inequality holds:

f(Si) + f(Sj) ≥ f(Si ∪ Sj) + f(Si ∩ Sj)

Let’s first assume that the x points covered in Si are all different from the y points covered in Sj. Then we would get that x + y ≥ x + y + 0, which is true. Let’s then assume that the x points covered in Si are all the same as the y points covered in Sj. This also means that x = y. Then we would get that x + y ≥ x + x since the union of Si and Sj would be the same number of points as x, and the intersection of Si and Sj includes all points in Si and Sj (x points total). We can then simplify it to x + x ≥ x + x, which is true. Let’s finally assume that there are “z” points that are unique to Sj and not in Si and that there are “a” points that are unique to Si and not in Sj such that x - a = y – z = intersection of Si and Sj. We would get x + y ≥ x + y – (x – a) + x – a = x + y – (y – z) + y – z. We can then simplify this to x + y ≥ x + y, which still holds true. Since f(Si) + f(Sj) ≥ f(Si ∪ Sj) + f(Si ∩ Sj) holds for any set Si, Sj among the Sm finite sets, we can conclude that function f is submodular.

A more formal proof of the above is as follows. We know that S(Si ∪ Sj) = S(Si) ∪ S(Sj) and that S(Si ∩ Sj) ⊆ S(Si) ∩ S(Sj). Then we know that f(Si) + f(Sj) = |S(Si)| + |S(Sj)| = |S(Si) ∪ S(Sj)| + |S(Si) ∩ S(Sj)| ≥ |S(Si ∪ Sj)| + |S(Si ∩ Sj)| = f(Si ∪ Sj) + f(Si ∩ Sj).  
  
B) The following greedy algorithm iteratively selects the set that provides the maximum marginal gain in terms of increasing the union size until it has selected k sets.  
 I = [] # Initialize an empty set I

while len(I) < k:

best\_set = None

best\_increase = -1

for i in range(1, m + 1):

if i not in I:

I\_candidate = I + [i] # Add i to the current set I\_candidate

increase = f(I\_candidate) - f(I) # Calculate the marginal gain

if increase > best\_increase:

best\_set = i

best\_increase = increase

if best\_set is not None:

I.append(best\_set) # Add the best set to the solution

return I  
  
C) The greedy algorithm makes a series of iterations until it selects k sets. In each iteration, it evaluates the marginal gain by calling the function f. In the worst case, it may need to evaluate f for each of the m sets (m iterations), and this process repeats until k sets are selected. Therefore, the algorithm calls the function f a maximum of O(m \* k) times.  
  
D) For i from 0 to k, let si be the size of my union of i sets, after i greedy steps. Clearly s0=0, and sk ≤ OPTk. Let di=OPTk-si. So d0=OPTk, and we would like to show dk is small. Here I will show that di+1 ≤ (1-1/k) di. Let U be the set of points covered by the optimal k-cover, but not covered by mu greedy cover (after i steps). We know that |U| is at least di because if the set that was chosen in the ith step are all points that are covered in OPTk, then |U| would become di = OPTk – si since now those points covered in both si and the optimal k-cover and thus do not belong in U anymore. On the other other hand, |U| can only get larger than di if the set chosen in step i contains points not covered in the optimal k-cover.   
  
Now, let’s argue that there is a set in the unknown optimal k-cover that covers at least |U|/k points of U. In the extreme case where the points in |U| are spread out equally in k sets, choosing any set would return a set that would cover |U|/k points. Considering all other cases where there may be some overlap between sets in the optimal k-cover, by the pigeonhole principle we would be covering at least |U|/k points by choosing any set in the optimal k-cover since the covers will either contain |U|/k points OR more than |U|/k points. They could contain more than |U|/k points since, if the cover contains points that are covered in other sets in the optimal k-cover, then that cover will actually represent a bigger fraction of points covered by the optimal k-cover.  
  
On step i+1, my greedy algorithm should select a set which covers at least |U|/k new points based on the arguments above. We can then infer that di+1 ≤ (1-1/k) di because di+1 = |U| - |U|/k ≤ di – di/k = (1-1/k) di. With this inequality, we can get that dk ≤ (1-1/k) dk-1 ≤ (1-1/k)^2 dk-2 ≤ … ≤ (1-1/k)^k d0 = (1-1/k)^k OPTk. We also know that dk = OPTk – f(I), since dk is the distance of the returned I set after k steps of the greedy algorithm from the actual optimal k-cover. We can then get OPTk – f(I) ≤ (1-1/k)^k OPTk 🡪 **f(I) ≥ OPTk – (1-1/k)^k OPTk = [1 – (1 – 1/k)^k] OPTk.**

1. A) The following is the formulation of the Hollywood problem as an integer linear program:  
     
   Let xi be a binary variable that takes the value 1 if actor i is chosen, and 0 otherwise for i=1,2,…,n. Let yj be a binary variable that takes the value 1 if investor j is chosen, and 0 otherwise for j=1,2,…,m. The objective function is to maximize the profit, which is the sum of payments from investors minus the payments to actors:

We also need to enforce the following constraints to ensure that the integer linear program is representative of the problem. The following constraint ensures that yj can only be 1 if all the actors in Lj are selected.

The next two constraints are simply to indicate that the values for x and y are integers and can only be 0 or 1.

B) We can relax the above integer linear program to a linear program by simply replacing the last two constraints in part A with the constraints below.

To show that this linear program still has an integral optimal solution, we can treat this problem as a flow network where actors are sources, investors are sinks, and the funding conditions are capacity constraints on edges.

Suppose we have the following problem. 3 available actors (A1, A2, A3) with costs (s1, s2, s3) = (2, 3, 4). 2 available investors (I1, I2) with investments (p1, p2) = (6, 5).

Funding conditions:

I1 requires (A1, A2).

I2 requires (A2, A3).

Step 1: Start with a fractional solution.

x1 = 0.6 (A1 is partially chosen).

x2 = 0.7 (A2 is partially chosen).

x3 = 0.4 (A3 is partially chosen).

y1 = 0.8 (I1 is partially chosen).

y2 = 0.3 (I2 is partially chosen).

Step 2: Create the Flow Network.

Construct a flow network with source S, sink T, and intermediate nodes representing actors and investors.

S connects to actors (A1, A2, A3) with capacities (0.6, 0.7, 0.4).

Investors (I1, I2) connect to T with capacities (0.8, 0.3).

Now, connect actors to investors based on the fractional solution:

Connect A1 to I1 with capacity 0.6.

Connect A2 to both I1 and I2 with capacities 0.7 and 0.7.

Connect A3 to I2 with capacity 0.4.

Step 3: Find a Max Flow.

Apply a max flow algorithm (e.g., Ford-Fulkerson or Edmonds-Karp) to find the maximum flow in the network.

In this example, you'll find that the maximum flow is 1.0. The flow through each edge will indicate the integer values for xixi​ and yjyj​ that will lead to a feasible solution:

Flow on the edge (S, A1) = 0.6 (x1 = 1).

Flow on the edge (S, A2) = 0.7 (x2 = 1).

Flow on the edge (S, A3) = 0.4 (x3 = 0).

Flow on the edge (A1, I1) = 0.6 (y1 = 1).

Flow on the edge (A2, I1) = 0.6 (y1 = 1).

Flow on the edge (A2, I2) = 0.1 (y2 = 0).

Flow on the edge (A3, I2) = 0.4 (y2 = 1).

Flow on the edge (I1, T) = 0.8 (y1 = 1).

Flow on the edge (I2, T) = 0.3 (y2 = 0).

Step 4: Convert to an Integer Solution.

Now, we have a feasible integer solution:

x1=1x1​=1

x2=1x2​=1

x3=0x3​=0

y1=1y1​=1

y2=0y2​=0

You can see that by starting with a fractional solution, creating a flow network, finding the maximum flow, and then adjusting the values based on the flow, we have obtained a fully integer solution.

1. B) To compute the fattest s-t path in a graph, you can use a variant of Dijkstra's algorithm known as the Capacity Scaling Algorithm. This algorithm is used to find the maximum flow in a network and can be adapted to find the fattest path. Here's how it works:
2. Initialize a flow variable to 0.
3. Run Dijkstra's algorithm with a modification:
   1. Instead of using the shortest path based on the sum of edge weights, choose the path with the highest capacity. You can achieve this by using a Maximum Priority Queue in Dijkstra's algorithm to prioritize paths with higher capacity. One possible structure that will have this desired effect is to use a binary heap. This can be done by assigning a priority to each path based on the minimum capacity of its constituent edges.
   2. Keep track of the current maximum capacity found so far.
4. While there exists a path from s to t in the residual graph (the original graph minus the flow), repeat the following steps:
   1. Find the path with the highest capacity using the modified Dijkstra's algorithm.
   2. Augment the flow along this path by the minimum capacity of its constituent edges.
   3. Update the residual capacities of the edges.
5. Repeat step 4 until no s-t path exists in the residual graph.
6. Return the maximum capacity found during the process as the capacity of the fattest s-t path.

This algorithm works because it repeatedly finds the s-t path with the highest capacity, effectively finding the fattest path.

C) A simple argument for this question is that, in the worst case scenario, all of the edges in the graph form a “single, straight path” from s to t. This also means that, in the worst case, the fattest and only path in the graph G involves all E edges in the graph, which ultimately indicates that we can write the flow as a sum of at most |E| path flows.

Below is a more elaborated answer to answer this question, where we still argue that, when given a flow f, we can write it as a sum of at most |E| path flows. We will also argue that these paths all use their edges in the same direction as f, so there is no cancellation between these path flows.

First, suppose f is an st-flow in G, let E(f) be the edges used by f, i.e. E(f) = {e: f(e)>0}. We will argue that if f has positive flow value, then you can find a path P in E(f) from s to t. Assuming that f has a positive flow value, this means there exists a flow of positive value from the source s to the sink t in the flow network G. To find a path from s to t in E(f), you can use a standard graph traversal algorithm like Depth-First Search (DFS) or Breadth-First Search (BFS) on the residual graph G\_f (which is the graph representing the remaining capacity after the flow f is applied). These algorithms will allow you to find a path from s to t, consisting of edges with positive flow values.

Next, let fP be the flow on path P, with value min{f(e): e on P}, which is positive. Let f' = f-fP, think of f' as the "remaining flow" after removing fP. For edges in P, f'(e) = f(e) - fP(e) = f(e) - min{f(e): e on P} ≥ 0, since fP(e) is the minimum flow value on path P. For edges not in P, f'(e) = f(e) - 0 = f(e) > 0, since there was no flow on these edges in fP. Therefore, E(f') is a subset of E(f) because for each edge in E(f'), either its flow value remains non-negative or it was not part of the path P.

Finally we will argue that the process eventually decomposes the original f into not-too-many path flows, with no cancellation. At each step, you find a path P from s to t in E(f), and the flow fP is subtracted from f. Since fP is positive, it decreases the flow on some edges in the original flow f. This process continues until there are no more paths from s to t in E(f). At this point, the remaining flow f' is such that there are no paths from s to t in E(f') because you've removed all paths. The flow f has been decomposed into path flows, and there is no cancellation because each flow fP on a path P is subtracted from the original flow f, ensuring that the flow values only decrease.

In conclusion, the process described decomposes the original flow f into not-too-many path flows with no cancellation, where each path flow corresponds to a path from s to t in the flow network G.  
  
D) Let’s say that f’’ is an optimal flow of the graph, and we know that the value of the flow is F. Then let’s say f is the current Ford Fulkerson flow which had an initial value of 0. Let’s define f’ = f’’ – f, and it is a flow that should exist within the residual graph Gf of the Ford Fulkerson algorithm. Based on the proof of part c above, we know that the fattest path p has flow value greater than or equal to that of all previous iterations of the path, which is also ≥ 1/|E| \* the flow value of f’ = 1/|E| \* flow value of f’’ – flow value of f = 1/|E| (F – flow value of f).

Using induction and initial f = 0, we can show that after k iterations of adding the fattest paths, we get that 0 ≤ F – flow value of f ≤ (1 – 1/|E|)^k \* F. If k = |E| ln F, we can see that 0 ≤ (1 – 1/|E|)^k \* F = F \* (1 – 1/|E|)^(|E| ln F) = F \* [(1 – 1/|E|)^(|E|)]^(ln F) ≤ F \* (1/e)^(ln F) = F/F = 1. In this case, the flow value of f must equal to F since 0 ≤ F – flow value of f ≤ 1. From this inequality, we can see that by the kth iteration, the current flow value of f is equal to the optimal flow value of F, which means that by |E| ln F iterations (at most O(|E| ln F) time), Ford Fulkerson should terminate.