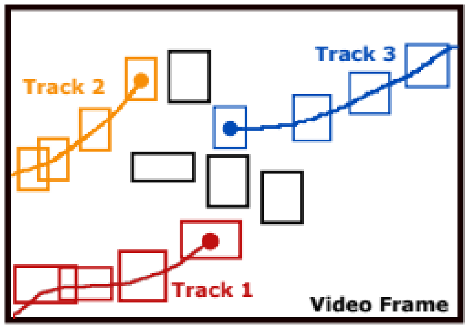
1. [40 points] (Integer Programming) Multi-object tracking within optical video is a common problem in machine learning. Tracking is extremely difficult in general as the number, sizes, and dynamics of objects can be large. Further, objects can by occluded by other objects and scenery, causing variable time gaps between one detection and the next.



igure 1: A tracking system that has three active tracks and has received four detections (black boxes) from the etector. Note that the time intervals between detections assigned to tracks varies, as does the shape of their ounding boxes.

Tracking systems require several key components:

- **Detector**– A model that signals when an object of interest is apparently present, and returns a bounding box in pixel coordinates.
- **Tracker**– Software module that stores information regarding active tracks, and assigns object detections to tracks using a similarity score matrix produced by the *Evaluator*. Optimally, the *Tracker* assigns detections to tracks in a manner that maximizes the sum of the similarity scores from each track/detection pair.
- **Evaluator**– Given N active tracks and M detections, produces a score matrix $\boldsymbol{S} \in \mathbb{R}^{M \times N}$ such that element $s_{ij} \in \boldsymbol{S}$ indicates the similarity of detection i to track j under some similarity measure. Note that this definition leaves the actual definition of "track" fairly abstract. In the simplest case, for example, the *Evaluator* may define s_{ij} as the Euclidean distance between the center of the final detection within track j and the new detection i. More sophisticated models may uses multiple past detections within a track, or model the dynamics of an objects (e.g., with a Kalman filter).

In this problem, assume that you'd like to produce a novel tracking system. The *Detector* and *Evaluator* have already been created, and you must now describe the *0-1 integer linear program* that the *Tracker* will solve to assign detections to tracks. Integer linear programs are constraint problems where the variables are restricted

to be integers, and are typically NP-hard. In particular, 0-1 integer linear programs restrict the variables further by requiring them to be Boolean (i.e., zero or one).

Assume that you are provided with the score matrix $\boldsymbol{S}$, detections can only be matched to a single track, and that tracks cannot be assigned more than once.

(a) [5 points] What do the variables in this problem represent? How many are there?

(b) [15 points] Define the objective for this 0-1 integer linear program.

(c) [20 points] Define the entire 0-1 integer linear program, including constraints, in standard form. How many constraints are there in the program, total?

a) Given N tracks and M detections, there are MxN variables. Each of the MxN variables will be either a 1 or a 0, and will indicate which detections match with what track. A 0 will indicate no match, while a 1 at detection i and on track j will indicate that we would like to match detection i with track j. We can call this 2D array of variables for our problem B.

b) The objective of this problem, based on the given score matrix, is to maximize the matching score for assigning object detections to tracks. Each detection can be assigned to only one track, and each track can likewise hold only one detected object. There may be more detections than tracks, however each track will be matched to a detection (this is assuming that all values in S are positive, if values can be negative it is possible that a track will not be matched to any detection). Thus, mathematically the objective function will look like:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} S[i][j] * B[i][j]$$

All this is saying is that we are maximizing the total matching between our Boolean 2D array that holds the True/False indications for the matching with the score matrix S. By maximizing this objective function we will get the best possible match for all N tracks and M detections based on score matrix S.

c) To define the problem in standard form we must first define the constraints for the problem. Informally, the constraints of the problem are simply that each track must match with exactly one detection. For a single track, we can write this mathematically for track j:

$$\sum_{i=1}^{M} B[i][j] = 1$$

Since we are using a 0 to indicate no match and a 1 to indicate a match, writing out the summation is easy and indicates that each track will match with only one of the M detections. However, standard form dictates that we may not have an equality constraint as a part of our constraints. We can get around this by specifying a pair of constraints

$$\sum_{i=1}^{M} B[i][j] \leq 1$$

$$-\sum_{i=1}^{M} B[i][j] \leq -1$$

Thus, written in standard form, the linear program will be set to maximize our equation in part b subject to the two constraints of the previous equation for each track j in N.

It follows from there that there are twice as many constraints as there are tracks for this problem, thus there are 2N constraints.

2. [40 points] ***Collaborative Problem***: In the course content, we explained how we can solve two-player, zero-sum games using linear programming. One of the games we described is called "Rock-Paper-Scissors." In this problem, we care going to examine this game more closely. Suppose we have the following "loss" matrix for Player 1 (i.e., we are showing how much Player 1 loses rather than gains, so reverse the sign):

$$A = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix}.$$

(a) [10 points] What is the expected loss for Player 1 when Player 1 plays a mixed strategy $x = (x_1, x_2, x_3)$ and Player 2 plays a mixed strategy $y = (y_1, y_2, y_3)$?

(b) [10 points] Show that Player 1 can achieve a *negative* expected loss (i.e., and expected gain) if Player 2 plays any strategy other than $y = (y_1, y_2, y_3) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

(c) [10 points] Show that $x = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ and $y = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ form a Nash equilibrium.

(d) [10 points] Let $x = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ as in part (c). Is it possible for (x, y) to be a Nash equilibrium for some mixed strategy $y' \neq (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$? Explain.

a) The expected loss for player 1 when playing a mixed strategy x=(x1, x2, x3) and player2 plays mixed strategy y=(y1, y2, y3) can be written as a matrix multiplication where

$$x = [x1, x2, x3]$$

$$y = \begin{matrix} [y1, \\ y2, \\ y3] \end{matrix}$$

And the expected loss is:

$$expected\ loss = x * A * y = [x1, \quad x2, \quad x3] * \begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix} * \begin{matrix} [y1, \\ y2, \\ y3] \end{matrix}$$

$$= [-x2 + x3, x1 - x3, -x1 + x2] * \begin{matrix} [y1, \\ y2, \\ y3] \end{matrix}$$

$$= y1 * (x3 - x2) + y2 * (x1 - x3) + y3 * (x2 - x1)$$

We can check our work. If y=[0, 0, 1] (y plays scissors all the time), and x=[1, 0, 0] (x plays rock all the time, we would expect x to win every time, thus have an expected loss of -1 (i.e. expected gain of 1). According to the previous equation, we have the expected loss:

$$expected\ loss = 0 * (0 - 0) + 0 * (1 - 0) * 1 * (0 - 1) = -1$$

Which is exactly what we would expect, such that x will win every time.

b) For any strategy for y other than y=[1/3, 1/3, 1/3] there is a mirrored strategy in x that produces a negative expected loss (expected gain) for player x. We will assume that player x will play a strategy where x=[y3, y1, y2]. This means that whatever probability player 2 assigns to rock, we will assign that value to paper. Likewise, whatever probability player 2 assigns to paper, we will assign to scissors, and whatever probability player 2 assigns to scissors, we will assign to rock. Thus, if player 2 plays scissors every time (like in part a), player 1 will play rock every time. If player 2 plays scissors and rock each half of the time, we will play rock and paper each half of the time. Our expected loss utilizing this strategy is:

$$\begin{aligned} expected\ loss &= y1 * (y2 - y1) + \ y2 * (y3 - y2) + y3 * (y1 - y3) \\ &= (y1 * y2) + (y2 * y3) + (y3 * y1) - y1^2 - y2^2 - y3^2 \end{aligned}$$

The expected loss equation will evaluate to negative for any values such that y1 != y2 or y1 != y3 (meaning at least one of the values in y1, y2, y3 is different from the other two). Given also logically that y1 + y2 + y3 =1, we know that the only scenario where the expected loss equation will evaluate to 0 is when y1 = y2 = y3 -> y1, y2, y3 = 1/3. All other values for y1, y2, y3 will yield a negative expected loss, thus expected gain for player 1.

There it follows that player 1 can achieve a negative expected loss (and thus expected gain) if player 2 plays any strategy other than y=(1/3, 1/3, 1/3) using the given strategy.

c) We have shown in part b) that if player B plays any strategy other than y=(1/3, 1/3, 1/3), player A can play a strategy which gives a negative expected loss. However, if player B plays y=(1/3, 1/3, 1/3), then there is no strategy that player A can play that gives negative expected loss (in fact, if player B plays y=(1/3, 1/3, 1/3) then no matter what strategy player A plays, they will have an expected loss of 0, or break even). The same applies for player B if player A plays x=(1/3, 1/3, 1/3), such that there is no strategy which will give a better payoff for player B. A Nash equilibrium occurs if there is a strategy for all players such that neither player can increase their payoff by choosing an action different from the current action. Thus, the point x=(1/3, 1/3, 1/3), y=(1/3, 1/3, 1/3) forms a Nash equilibrium.
d) We know from part b) that player A can achieve a negative expected loss for any strategy played by player B that is not y=(1/3,1/3,1/3). We also know that the Nash equilibrium point indicates a point where neither player can increase their payoff by choosing an action different from the current action. If player B does not play y=(1/3,1/3,1/3), then player A has incentive to change to a strategy which increases their payoff. Thus any point other than y=(1/3,1/3,1/3) will not be a Nash equilibrium point.

3. Consider the following problem. There is a set U of nodes, which we can think of users (e.g., these are locations that need to access a service, such as a web server). You would like to place servers at multiple locations. Suppose you are given a set S of possible sites that would be willing to act as locations for servers. For each site s ∈ S, there is a fee fs ≥ 0 for placing a server at that location. Your goal will be to approximately minimize the cost while providing the service to each of the customers.

So far, this is very much like the Set Cover Problem: The places s are sets, the weight of a set is fs, and we want to select a collection of sets that covers all users. But there is one additional complication. Users u ∈ U can be served from multiple sites, but there is an associated cost dus for serving user u from site s. As an example, one could think of dus as a “distance” from the server. When the value dus is very high, we do not want to serve user u from site s, and in general, the service cost dus serves as an incentive to serve customers from “nearby” servers whenever possible.

So here is the question, which we call the Facility Location Problem. Given the sets U and S with associated costs d and f, you need to select a subset A ⊆ S at which to place the servers (at a cost of P s∈A fs) and assign each user u to the active server where it is cheapest to be served, mins∈A dus. The goal is to minimize the overall cost

$$\sum_{a \in S} f_s + \sum_{u \in U} \min_{s \in A} d_{us}.$$

Give a ρ(n)-approximate algorithm for this problem.

We can follow an approach similar to the greedy strategy used for set cover from section 35.5 of the textbook. Given just one open site, we can calculate the cost associated with serving each user from the given site. When opening a new site, we can associate the site with the subset of users which would be served in a cheaper manner from the new site. Calculating the amount saved for all users who are served in a cheaper manner with the new site minus the cost of the new site will give the total amount saved building the new site. Similar to how set cover greedily selects the set which covers the most unassigned nodes, we can select the site which reduces the cost to serve the users the most. We can do this until there are no site locations that reduce the total cost. This will be the greedy solution which meets the goal of minimizing the cost. Since the algorithm runs in the same time as the greedy set cover problem, this problem runs in ρ(n)-approximate time. Pseudocode would look something like this for the given problem:

```
def greedy_facility_loc(U, S)                    # function input is users U and possible site locations S
 sites = []                                      # this is a list that will store where to build sites
 user_costs = {}                                 # dictionary to store the cost for each user
 for user in U:
  user_costs[user.id] = big_number               # start by setting some big cost for each user (not inf.)
 while(true):
  best_site, savings = best_savings(U, S, user_costs)     # get the best site and associated savings
  if savings < 0:
   break                                         # if best savings negative, no more sites reduce cost
  sites.append(best_site)                        # append the best site to sites
  S.pop(site)                                    # remove site from S so that we don't have to check it
  for user in U:
   if best_site.cost_for_user(user) < user_costs.get(user.id):     # check if the new site is better for user
    user_costs[user.id] = best_site.cost_for_user(user)            # set the new cost for the user
 # end while loop
 return sites                                    # return the sites which minimize cost

def best_savings(U, S, user_costs)               # compute which remaining site will give best savings
 best_save = -inf                                # store the value of the best savings
```

```
savings = 0                                          # variable to store savings for a site
for site in S:
  for user in U:
    if site.cost_for_user(user) < user_costs.get(user.id):   # check if new site is better for user
      savings += site.cost_for_user(user) – user_costs.get(user.id)# add up the savings
  savings -= site.cost                               # factor in the cost of the site
  if savings > best_save:                            # check if this is the new best site
    best_site = site                                 # save the new best site
  savings = 0                                        # reset savings to 0
return (best_site, savings)                          # return the best site and the associated savings
```

Section 35.3 of the book makes the argument that the greedy set cover is not too much larger than an optimal set cover. Along these lines, the greedy minimum cost is not too much larger than the optimal minimum cost. Thus this algorithm meets the requirement of the goal being to minimize cost.

The best_savings() helper function iterates over each site in S and each user in U and performs a finite set of operations within the loops before returning the most optimal site to select to reduce costs based on the sites and users. Thus the best_savings() helper function runs in O(U*S) time. The best_savings() helper function is run within a while loop in the main function. The while loop will run no more than O(S) times, since the worst case scenario is that it iterates through each of the sites in S. Thus, the runtime of the whole algorithm is O(S^2*U), which fits the requirements of a ρ(n)-approximate time algorithm, just like the greedy set cover problem laid out in the book.