Proposal for 'New' Course Allocation Policy

Assumptions:

- **Bucket:** Bucket is a virtual container which stores the course preference order for an elective category for a student. Eg: A bucket storing X student's course preference order for open-elective category
- **Bucket Category:** This is an attribute of the bucket. This is the same as the elective category for which the bucket stores the course preference order. Eg: The bucket mentioned in above example has bucket category 'open-elective'
- The number of seats (i.e. totalled value of the number of seats for each course offered under the bucket category) for a bucket category is always higher than the number of students who select the bucket category

---

Method 1 (Preferred): Preference-Order based Course Allocation Algorithm with true Randomisation

The process begins by having students rank the courses they would like to take within a particular category (eg: Humanities, Science, Open, etc), with ***1*** being their top preference and ***m*** being their lowest preference, where ***m*** is the number of courses they select out of the ***n*** courses offered.

Now, for a particular category, the algorithm will proceed by taking each preference rank into consideration, starting from preference rank 1.

Now while going through each of the preference ranks, for each course that has more seats available that the number of students who selected it as their preference for that preference rank, that course will be allocated to all of those students. The algorithm will decrease the number of seats available for that course and remove those students from consideration for further rounds.

For courses that have fewer seats available than the number of students who selected it as their preference for that preference rank (i.e. the course is over-booked), the algorithm will randomly select students to allocate to the course until all the seats are filled. The algorithm will then remove the course from consideration for further rounds and remove the allocated students from consideration for lower preference courses.

Demostration of working of the algorithm by the use of a short example:

Suppose 3 students A, B, and C have filled their preference order for the Humanities bucket, where the courses available and the seats available are as follows:

Course Code	Course Name	Seats Available
101	Exploring Masculinities	1
102	Literature and Ethics	1
103	Crime and Punishment	1
104	Science, Tech and Society	1
105	Intro to Human Sciences	1
106	Values and Ethics in AI	1

And the preferences filled by the students are:

Student Name	Preference 1	Preference 2	Preference 3
A	106	103	102
B	104	102	101
C	104	102	103

Now, the following process will execute to allocate courses:

- First, the program will take all the courses that have been selected at preference rank 1
  - Such courses are couse 106 by A and course 104 by B and C
- Order in which the courses are iterated doesn't have a causations and thus, suppose it first takes the course 106 into consideration
  - As number of students who took it at preference rank 1 (here 1) is equal to the number of seats available (again 1), course 106 is allocated to A
  - Now the course 106 and student A are removed from consideration for further rounds
- Now, it will take course 104 into consideration (as that is the only course left which was chosen as preference 1 by the students)
  - As number of students who took it at preference rank 1 (here 2) are more in number than the number of seats available (here 1), we will select a student at random. Suppose student C is chosen
  - Now the course 104 and student C are removed from consideration for further rounds

- Since the seats for every course that had been selected as preference 1 have been filled, we will look at the preference rank 2
- Courses selected at preference rank 2 are: course 102 by B (other students have already been alloted the best possible course and thus are no longer considered)
- Now, we will take the course B into consideration
    - As number of students who took it at preference rank 2 (here 1) is equal to the number of seats available (here 1), course 102 is allocated to B
- Now, since every student has been allocated a course, the algorithm will stop here

Pseudo-code:

If a bucket category offers ***n*** courses and a student chooses ***m*** courses, then the student will have to assign a preference rank between ***1 to m*** to each of these ***m*** courses. ***1*** would mean first preference (***preference_1***) and ***m*** would mean last preference (***preference_m***). Once the above process is complete for every student, the following flow will be used

(**NOTE:** The following flow is for a single bucket category and can be extrapolated for multiple bucket categories)

- ***n*** == 1
    - Allocate the course to every student who selected the bucket
- ***n*** > 1
    - For ***preference_i*** in (1, 2, ... ***n***)
        - Let number of unique courses chosen by students as ***preference_i*** be ***course_c***
        - Calculate number of students who have a course as ***preference_i*** for each course
        - For ***course_i*** in (1, 2, ... ***course_c***)
            - If number of seats in ***course_i*** > number of students who chose ***course_i*** as ***preference_i***
                - Allocate the course to every student
                - Decrease the number of seats for ***course_i*** by the number of students who chose it
                - Push the details for these students to Allocated DB and remove them from the Temporary DB (thus students who have been allocated this course will not be considered for further rounds involving lower preferences)
            - If number of seats in ***course_i*** < number of students who chose ***course_i*** as ***preference_i***
                - Use ***random.shuffle()*** function from the ***random*** python library on the list of students

- Sequentially pick students from this shuffled list till number of students chosen becomes equal to the number of number of seats in ***course_i***
- Remove the course from the Temporary DB (thus this course can no longer be allocated to more students)
- Push the details for these students to Allocated DB and remove them from the Temporary DB (thus the students will no be considered for further rounds involving lower preferences)

Method 2 (Alternate): Preference-Order based Course Allocation Algorithm with Hyperbolic Discounting

This process is a modification of the previous one. While the basic structure remains the same, it differs from the above algorithm in the fact that in case of an overbooking, rather than choosing the students completely at random, we assign weights to the students in order to introduce the element of First Come, First Serve, so that even the selection remains random, the student who fill the form early, and thereby has a higher weight assigned to them, will have a higher chance to be selected by the algorithm.

This assinging of weight is based on the mathematical model of Hyperbolic Discounting. Hyderbolic Discounting states that the rewards received earlier are more valued than rewards of more valued received later. This concept is used to assign a higher weight (value) to the student who fills in the preferences earlier as opposed to the one who fills them later. These weights influence the algorithm as the student with a higher weight will have a stronger effect on the algorithm and thus make the algorithm susceptible to favouring the student.

The process begins by having students rank the courses they would like to take within a particular category (e.g. Humanities, Science, Open, etc.), with **1** being their top preference and **n** being their lowest preference. Weights will be assigned to the students in the same order that they had filled the form, that is, the student who fill the form first will be assigned the highest weight, the student who filled the form second will be assigned the second-highest weight, and so on. This order of filling the form is saved so as to assign the weights later for each iteration and each round.

Now, for a particular category, the algorithm will proceed by taking each preference rank into consideration, starting from preference rank 1.

Now while going through each of the preference ranks, for each course that has more seats available than the number of students who selected it as their preference for that preference rank, that course will be allocated to all of those students. The algorithm will decrease the number of seats available for that course and remove those students from consideration for further rounds.

For courses that have fewer seats available than the number of students who selected it as their preference for that preference rank, the algorithm will randomly select students while keeping their order of filling the form in mind to allocate to the course until all the seats are filled. To achieve this, weights are assigned to each student. The student among the students being considered for the round (i.e. allocation of this specific course) is assigned a weight equal to the number of students being considered. The weight assigned reduces by one for each student iterated according to the order in which they filled the form. Now, a number is selected randomly (using ***random.randint()*** function from the ***random*** python library) between 1 and the 'total of the weights of students being considered'.

Note that everytime the course changes or a student is removed from the list, the weights for each student and thus the total weight will be recalculated to accomodate the changes in the students being considered for allocation.

Then, we iterate through the students list in order of their weights (higher to lower) and keep on substracting their weight from this random number. As soon as the number reduces to 0 or below, the iteration is stopped and the student whose weight caused this drop is allocated the course. The student is also removed from the list so that the student does not influence further iterations.

This process is then again repeated (i.e. selection of a random number and then iteration) to select students till number of selected students becomes equal to the number of seats in the course. In the case that the selected random number does not reduce to 0 or below and the list has been iterated, the iteration is repeated till this condition is satisfied, without changing the randomly selected number (i.e. taking the final value after the iteration). This ensures that the student with a higher weight has a higher chance of reducing the selected number to 0 or below and thus the allocation is susceptible to favour the student. Though, this may not be always the case as can be seen from the below stated example and thus has a component of randomness to it. The algorithm will finally remove the course from consideration for further rounds.

Demostration of working of the algorithm by the use of a short example:

Suppose 3 students A, B, and C have filled their preference order for the Humanities bucket, with A filling the form first, B filling the form second, and C filling the form third. Therefore, the order of weights assigned to A, B and C is that A has the highest weight, B has the second-highest weight and C has the lowest weight. Let’s suppose the courses available and the number of seats available for them are:

Course Code	Course Name	Seats Available
101	Exploring Masculinities	1
102	Literature and Ethics	1
103	Crime and Punishment	1
104	Science, Tech and Society	1
105	Intro to Human Sciences	1
106	Values and Ethics in AI	1

And the preferences filled by the students are:

Student Name	Preference 1	Preference 2	Preference 3	Order of filling the form
A	106	103	102	1
B	104	102	101	2
C	104	102	103	3

Now, the following process will execute to allocate courses:

- First, the program will take all the courses that have been selected at preference rank 1
  - Such courses are course 106 by A and course 104 by B and C
- Suppose it first takes course 106 into consideration
  - As number of students who took it at preference rank 1 (here 1) is equal to the number of seats available (again 1), course 106 is allocated to A
  - Now course 106 and student A are removed from consideration for further rounds
- Now, it will take course 104 into consideration
  - As number of students who took it at preference rank 1 (here 2) is more than the number of seats available (here 1), we will select a student at random while keeping their weights in mind.
  - Since B filled the form earlier than C, B will have a weight of 2 and C will have a weight of 1 (as only 2 students are being considered).
  - Now, let us assume that the number chosen between 1 and the total of the weights of the students being considered (3 here) is 3.
  - The algorithm will iterate the list of students being considered in order of their weights and B will be encountered first. The number will be reduced to 1 after substracting the

weight of B (3 - 2 = 1).
    - The C will be encountered and the number will be reduced to 0 (1 - 1 = 0). Thus, C will be allocated the course.
    - An alternate case would have been that the number chosen could have been 1 or 2 and in that case B would have been allocated the course. Thus the probability of B getting the course was 2/3 or 0.66 (as probability would be count of numbers B can reduce upon the total count of numbers possible). Hence, naturally B has a higher chance of getting the course, though, as portrayed above C can get the course and thus the allocation remains fair.
    - Now course 104 and student C are removed from consideration for further rounds
- Since the seats for every course that had been selected as preference 1 have been filled, we will look at the preference rank 2
- Courses selected at preference rank 2 are: course 102 by B
- Now, we will take the course 102 into consideration
    - As the number of students who took it at preference rank 2 (here 1) is equal to the number of seats available (again 1), course 102 is allocated to B
- Now, since every student has been allocated a course, the algorithm will stop here

Pseudo-code:

Procedure for preference selection remains the same as above. If a bucket category offers ***n*** courses, students will have to assign a preference rank between ***1 to m*** to each of the ***m*** courses they select. ***1*** would mean first preference (***preference_1***) and ***m*** would mean last preference (***preference_m***).

Now, along with the preference rank students assign, we will store the position ranging from ***1 to student_c*** at which the student completes the procedure. Here ***student_c*** is the count of the students who filled the form.

In case of overbooked courses, we reuse ***student_c*** to mean the count of the students being considered for the iteration, we will allot every student a weight between ***1*** and ***student_c***, where ***student_c*** is the highest weight, in order of them filling the form. We will also calculate the total of the weights assigned to each student, i.e. ***weight_t***, using the formula: ***student_c * (student_c + 1) * 1/2***. This formula allows us to calculate the total without manually iterating over the students list and adding weight assinged to each student.

The process is detailed below:

(**NOTE:** The following flow is for a single bucket category and can be extrapolated for multiple bucket categories)

- ***n*** == 1
    - Allocate the course to every student who selected the bucket

- ***n*** > 1
    - For ***preference_i*** in (1, 2, ... ***n***)
        - Let number of unique courses chosen by students as ***preference_i*** be ***course_c***
        - Calculate number of students who have a course as ***preference_i*** for each course
        - For ***course_i*** in (1, 2, ... ***course_c***)
            - If number of seats in ***course_i*** > number of students who chose ***course_i*** as ***preference_i***
                - Allocate the course to every student
                - Decrease the number of seats for ***course_i*** by the number of students who chose it
                - Push the details for these students to Allocated DB and remove them from Temporary DB (thus students who have been allocated this course will not be considered for further rounds involving lower preferences)
            - If number of seats in ***course_i*** < number of students who chose ***course_i*** as ***preference_i***
                - Make a temporary list of students who have chosen ***course_i*** as ***preference_i*** and name it ***tempList***. This list should be sorted in descending order according to the order in which the student filled the form
                - While number of students allocated < seats in ***course_i***
                    - Calculate and assign the weights to the students
                    - Let ***seed*** = random.randint(1, ***weight_t***)
                    - For student in ***tempList***
                        - ***seed*** = ***seed*** - weight of ***student***
                        - If ***seed*** <= 0
                            - Push the details for ***student*** to Allocated DB and remove him/her from Temporary DB (thus the students will not be considered for further rounds involving lower preferences)
                            - Remove the student from the ***tempList*** and the Temporary DB and push

        the details to the Allocated DB.
        - break
    - If ***student*** is last entry in ***tempList***, reset ***student*** pointer to the first entry in ***tempList*** (i.e. reiterate through the list till condition for ***seed*** is satisfied)
- Remove the course from Temporary DB (thus this course can no longer be allocated to more students)

Proposal for 'New' Add/Drop Policy

Resource Allocation Algorithm:

This method is used to allocate courses during add/drop. If there are n courses, students will have to assign a preference rank to the courses they want to add instead of the course previously allocated to them, 1 would mean top preference (preference_1) and k would mean the lowest preference (preference_k, k<= n) (lower the rank, higher the preference) for that particular student . If a student does not get the course chosen by him /her during add/drop, then the course allocated to that student during course allocation will remain the same.

During add/drop, there is a possibility of occurrence of deadlock. An example is given below:

person_1 is allocated course A but wants course B and person_2 was allocated course B but wants course A. This is a clear situation of deadlock.

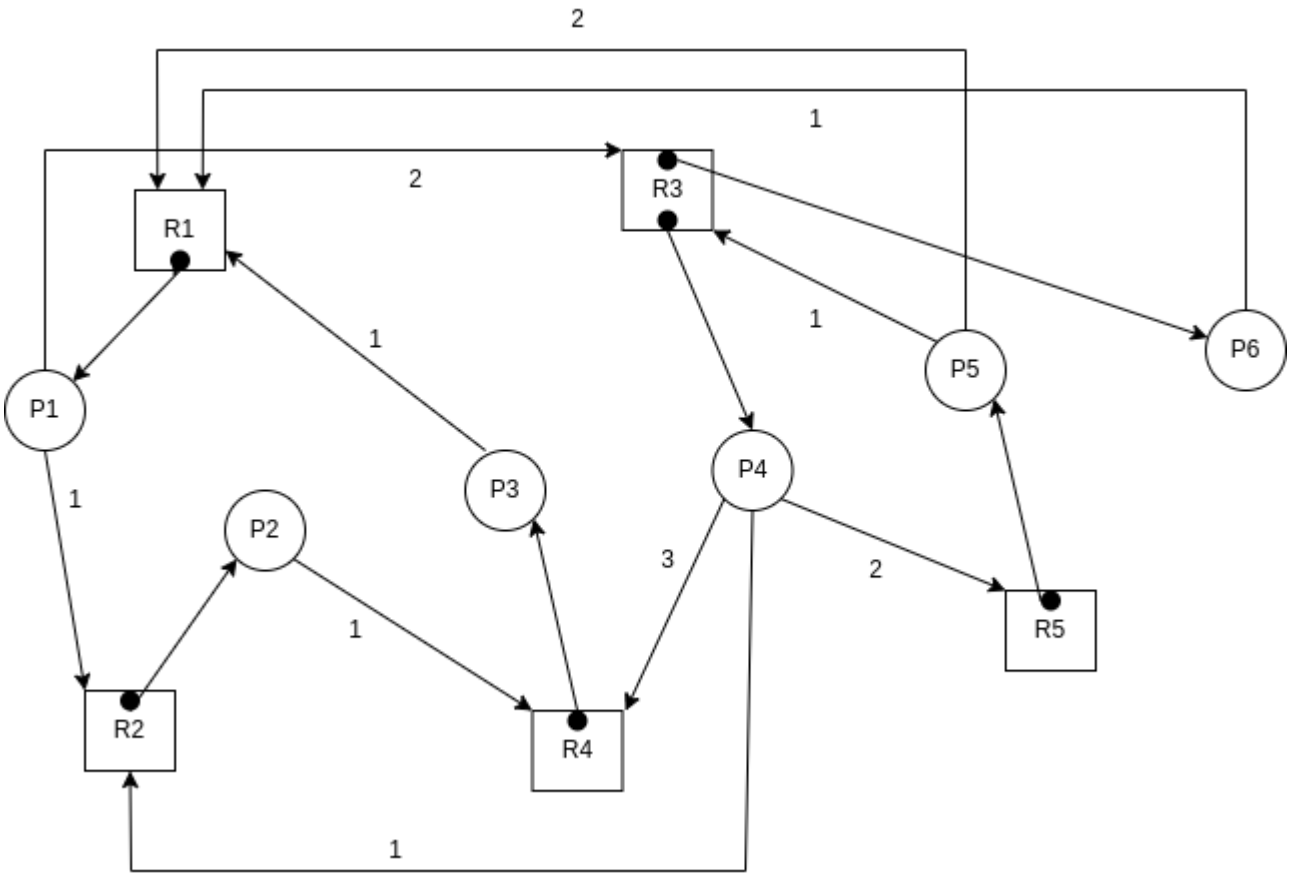
These types of problems can be solved by detecting cycles in the resource allocation graph (here, resource = course)

Assumptions for this example:

- Every course is overbooked
- Every course except R3 has only 1 seat available for add/drop whereas R3 has 2 seats available for add/drop
- Each student can add/drop atmost one course

Student	Previously Allocated	Preference 1	Preference 2	Preference 3
P1	R1	R2	R3	-
P2	R2	R4	-	-
P3	R4	R1	-	-
P4	R3	R2	R5	R4
P5	R5	R3	R1	-
P6	R3	R1	-	-

The resource-allocation graph for the above preferences will be as follows:



Pi -> Rj edge represents the request that person_ i has made for resource_j in the preference order. Rj -> Pi edge represents course Rj which was previously allocated to person Pi.

We have assigned a weight to every Pi -> Rj edge, the weight being the preference of Person_ i for Resource_ j. We assign a weight of 0 to every Ri -> Pj edge (as the course is already allocated to that particular person). Weight is assigned to an edge only when a student requests an overbooked course.

The algorithm will proceed as follows:

- We will find cycles in the graph. There are 2 cycles in the above graph.

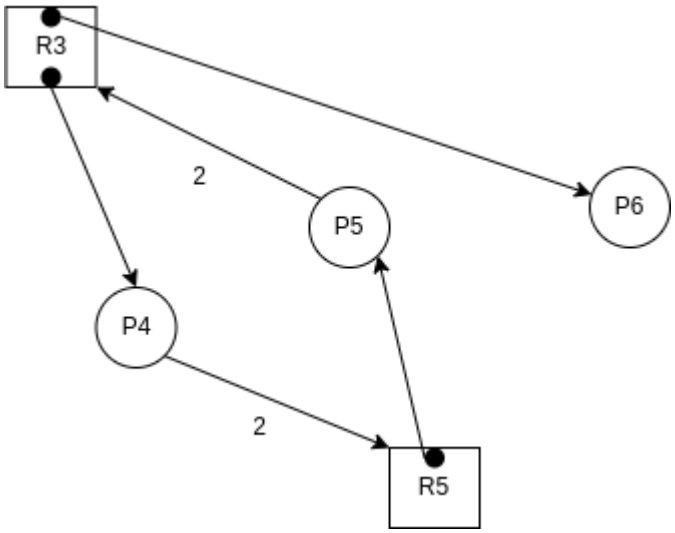
  R1 -> P1 -> R2 -> P2 -> R4 -> P3 -> R1 -- 1

  R3 -> P4 -> R5 -> P5 -> R3 -- 2

- We will find cycles that have edges with Weights 1 / 0. Cycle 1 satisfies the property.

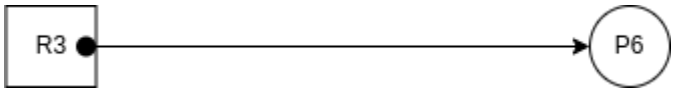
  Now we allocate R2 to P1, R4 to P2, R1 to P3 and remove the nodes P1, P2, R3 and R1, R2, R4 and their corresponding edges (as they have only one instance of resource available and that is occupied now).

- Now we replace the weight of edges with weight 1 by 2. The above graph changes as follows:



- We will find cycles that have edges with Weights 2/ 0. Cycle 2 satisfies the property.

  Now we allocate R5 to P4 and R3 to P5 and remove the nodes P4, P5 and resource R5.



- Now as the graph contains no Pi -> Rj edge. No more allocation is possible. P6 does not get a new course in add/Drop.

Pseudo-code:

// Non-overbooked courses

- For ***preference_i*** in (1, ***max_preference*** + 1)
  - For each student // the sequence order for students would be randomised
    - If ***course_i*** is not overbooked
      - Allocate course to student and remove him/her from database
      - Decrease the number of seats for ***course_i*** by 1

// Overbooked courses

- Consider, number of students who opted for add/drop = ***n***
  - Number of courses available for add/drop = ***m***
  - Number of preferences opted by all the students in total = ***p***
- Construct a graph with ***n*** + ***m*** vertices and ***p*** + ***n*** edges
- Draw a directed edge from ***Pi*** to all the courses the student opted for, with weight being the preference

- Draw a directed edge from resource to the person (course previously allocated to that person) with weight being 0

// Depth First Search (DFS) function to find cycles

- function dfs(vertex, visited, path)
    - visited[vertex] = True

      // mark vertex as visited
    - for each adjacent node u of vertex
        - if visited[u] == True
            - if u == path[0]

              // if u is the first vertex in the path, we have found a cycle
                - print path + [u]

                  // print the cycle
                - arr.push(path)

                  // store the cycle in an array of arrays which stores the path of a cycle
        - else
            - dfs(u, visited, path + [u])

              // continue DFS on adjacent node u, passing visited array and updated path
    - visited[vertex] = False

      // mark vertex as unvisited after DFS is complete

// function to allocate courses that are deadlocked

- function checkCycles(graph)
    - checkWeight = 1
    - While there is an edge from ***P_i*** to ***R_j***

- printAllCycles(graph)
- brr = []
- For cycle in arr
    - If weight of each edge in cycle == 0 or checkWeight

      // checking for cycles with equal edge weights and pushing them

        - brr.push(cycle)
- Option 1:
    - Sort brr with respect to decreasing order of their weights

      // allocating course to the max number of people with higher preference
- Option 2:
    - random.shuffle(brr)

      // we can allocate randomly as randomness sometimes ensures fairness
- For ***allocate_cycle*** in brr:
    - Allocate courses to the students present in ***allocate_cycle*** and remove the allocated vertices (students) from graph

      // allocating courses to all the disjoint cycles in brr

◦ For edges in graph

- If weight of edge == ***checkWeight***
    - Weight = ***checkWeight*** + 1

      // decreasing the priority of highest priority edge of 1
    - ***checkWeight***++
