1 Introduction

We study how to allocate m indivisible items among n agents in a fair way. Because the items are indivisible, it is not possible to allocate the items as in the cake-cutting problem, so *maximin share (MMS) allocation* is used for this situation. The *maximin share* of an agent is the maximum she can guarantee herself if she can choose a partition of items into n bundles (one for each agent), with the condition that all other agents get to choose a bundle before her. In this summer, we studied 1/2-MMS allocation, 2/3-MMS allocation [1], and 3/4-MMS allocation [2] and implemented these algorithms in Java.

Next, we formally define these notions.

Definition 1.1 (MMS Allocation)**.** *The maximin share (MMS) of an agent is the maximum she can guarantee herself if she is allowed to choose a partition of items into n bundles (one for each agent), with the condition that all other agents get to choose a bundle before her. An MMS allocation provides each agent a bundle worth at least her maximin share.*

Definition 1.2 (α-MMS)**.** *Agent i's maximin share (μ_i) is the maximum value she can guarantee herself if she is allowed to choose the allocation A, on the condition that she receives her least preferred bundle. Formally, an allocation is α approximate MMS (or simplify α-MMS) if each agent i receives a bundle A_i worth at least: $v_i(A_i) \geq \alpha\mu_i$, for some $\alpha \in (0,1)$. For instance, 1/2-MMS, 2/3-MMS, 3/4-MMS.*

2 Key Functions

In this section, we define the key functions of the implementation.

2.1 MMS class

For all three 1/2-MMS, 2/3-MMS, 3/4-MMS program, we build an MMS class first. This is a class that we use to present the whole problem system and use for the following calculations. This class makes the following calculations more convenient and it has some built-in functions inside this class. The variables we need for an MMS object is the number of the agents(n) and the number of items(m) and an $n \times m$ matrix. The number of agents and number of items are *int* and the matrix is composed of *double* numbers. In the matrix, each row of the matrix represents values of all items for one agent. For instance, *matrix[0][0] = 10.5* means that the first agent values the first item as 10.5. In the constructor, we have four to five variables need to be initialized by the three information above: *agentNumber*, *itemNumber*, *agentAllNumber*, *values*, *Unitem*. In the constructor, *agentNumber* and *agentAllNumebr* are both equal to the number of agents at first. The differences between them are that *agentAllNumber* will stay the same in the whole program, which is convenient for looping over all items and agents in the program, while *agentNumber* records the number of agents who are not assigned yet. The *itemNumber* represents and equals to the number of items. *Unitem* is a *list* which includes the numbers(in the form of *int*) from 0 to *itemNumebr*. *Unitem* is used in 3/4-MMS program for us to count the order of unassigned items.

2.2 Normalize

For problem instance $I = (N, M, V)$, if agent i's valuation function satisfies:

$$v_i(M) = \sum_{j \in M} v_{ij} = |N| \ .$$

Then $\mu_i \leq 1$. Normalizing agents' valuations provides a convenient upper bound on i's without affecting performance guarantees. After every agent leave the system, we need to repeat calling the function of *normalize*.

2.3 Order

In each row, to make that agent's valuation ordered from largest to smallest. This function is important for 2/3-MMS and 3/4-MMS allocation.

2.4 RandomArray

To generate a random double array of size $n \times m$. This is used in the main function to generate random MMS object for us to test our functions and programs.

2.5 IfS1, IfS2, IfS3, IfS4

Those functions are used in 3/4-MMS program to check if the sum certain order items of any agent is larger than 3/4. *IfS1* is to check the highest value. *IfS2* is to check the bundle with n^{th} and $(n+1)^{th}$ highest value items. *IfS3* is to check the bundle with $(2n-1)^{th}$, $2n^{th}$, $(2n+1)^{th}$ highest value items. *IfS4* is to check if the bundle with 1^{st} and $(2n+1)^{th}$ highest value items.

2.6 Allocation

The main allocation functions. The actual and most important allocation process is used by calling the allocation function. After we find the index of items and index agent to assign, the followings need to do:

1. Put the index of the items to the according agent's list in final results list.
2. *agentNumber* needs to minus one. This is to make sure the normalization is correct. Also, when the *agentNumber* equals to 0, it means the allocation can stop.
3. Add the index of the agent to the list of *FilledAgent*. This is to record the agents who have already been assigned and avoid duplicate assignment.
4. Add the index of the item to the list of *FilledItem*. This is to record the items who have already been assigned and avoid duplicate assignment.
5. In the *values* matrix, all the positions in the row of the assigned agent and the column of the assigned items need to be changed to zero. We do not change the index of items and agents during the allocation, but we make their rows and column to zero to mark them as leaving the system. So it would not affect normalization and future calculations.
6. For 3/4-MMS program, we also need to remove the item from unassigned agent item's list.
7. Call the normalize function. As long as there are agents leave the system, we need to repeat normalization.

3 Allocation methods

3.1 Bag Filling

All the program will use bag filling at the end to deal with low-value items. The bag filling algorithm is as follows: For α-MMS allocation, we add one, arbitrary item at a time to a bag S until an agent k value the bag at least α. We use two loops, one outside loop and one inner loop, to realize the bag filling in our algorithm. The outside loop is for adding new items to the bag. Each loop, if the item is unassigned, the item will be put in our temporary bag, which is a *list*. The inner loop is to check if the bag item satisfies α-MMS for an agent. If yes, we assign the items in the bag to that agent and clear the bag, then use "break" to leave the inner loop and continue the outside loop to fill a new bag.

3.2 1/2-MMS allocation

1/2-MMS is the most basic one among the three. The process is as follows:

1. Normalize the MMS.
2. Divide the items into the high-value item and low-value items. If the value is larger or equal to 1/2, it is a high-value item. If lower than 1/2, it is a low-value item.
3. Count the number of high-value items. If there are any high-value items, use the *matching* to assign the high-value item. *Matching* means, if the valuation of the agent is larger than or equal to 1/2, we assign the item to the agent and then the agent can leave the system. Then recount the number of high-value items. It is necessary to recount the high-value items because there might have one item that is considered as the high-value item for more than one agent.
4. If there are no more high-value items, use bag-filling to assign the low-value item.

3.3 2/3-MMS allocation

2/3-MMS allocation is more complicated than 1/2-MMS allocation. We also need to order the items and at the end, find their original index for final results. The process is as follows:

1. Make a copy of a original matrix.
2. Normalize the matrix and order the matrix.
3. Divide the items into the high-value item,low-value items, and medium-value item. If the value is larger or equal to 2/3, it is a high-value item. If lower than 1/3 for any agent, it is a low-value item. If neither higher than/equals to 2/3 nor lower than 1/3, it is a medium-value item.
4. Count the number of high-value items. If there are any high-value items, use the *matching* to assign the high-value item. Until there are no more high-value items unassigned.
5. Count the number of the medium-value items and compare with the number of the agents in the system.

6. If the number of medium-value items is larger or equal to the number of the agents in the system, we greedily assign the two least preferred items of the medium-value items to an arbitrary agent i with valuation $v_i(S) \geq 2/3$.

7. If the number of medium-value items is smaller than the number of the agents in the system. We simply initialize the bag S using one arbitrary medium-value item and then fill the bag with low-value items until some agent i values S at least $v_i(S) \geq 2/3$. It is a modified bag-filling process.

8. For low value items, use bag-filling.

9. Change back to original order. Now can have the ordered-results from the ordered matrix. And we still need to find the unordered-results of the unordered matrix: The the ordered results decide the order of the agent to choose their items according to their valuation from the original matrix. So, if one agent is assigned with the first item in ordered-results, he can first choose his highest-value item from unassigned items. This is to say, the ordered-results decide the order to choose. Every agent at his turn can choose his highest-value item from the unassigned items.

3.4 3/4-MMS allocation

The 3/4-MMS includes four main part: fixed assignment, tentative assignment, update upper bound and bag filling. The functions of *IfS1,IfS2,IfS3, IfS4* are needed here. Let $S1 = \{1\}$ (the highest value item for all agents), $S2 = \{n, n+1\}$ (the bundle with n^{th} and $(n+1)^{th}$ highest valued items for all agents), and $S3 = \{2n-1, 2n, 2n+1\}$ (the bundle with $(2n-1)^{th}$, $(2n)^{th}$ and $(2n+1)^{th}$ highest valued items for all agents) S4 = $\{1,\, 2n+1\}$ the highest value item and $(2n+1)^{th}$ highest valued items.

1. Order and normalize the matrix.

2. Fixed assignment. Recursively removes high value items using valid reduction.Allocate $S1, S2, S3$ bundle if any exists.

3. Keep some data backup here. If there exists a *type2B* agent in the system, we need to update the upper bound and go back to this place. The backup variables are current *agentNumber*(the current agent number still in the system), *Unitem* (the indexes of unassigned items), *values*, *Filledagent*(the agents who are assigned), *Filleditems* (the items which are assigned).

4. Tentative assignment. Allocate $S1$, $S2$, $S3$, and $S4$ bundle if any exists. After removing an agent i with $S4$, it may later trigger a valid reductions with $S1, S2, S3$.

5. Check if there is any *type2* agent.

6. If there are *type2* agent, check if there is any *type2B* agent.

7. If exists *type2B* agent, calculate the new upper bound.

8. Use the new upper bound for tentative assignment.

9. Bag-filling for low-value items.

4 Summary

I test the code by generating random MMS objects and the three programs can run well after taking randomly generate array numbers. It would worth exploring more examples and the use those programs to test some specific examples. Also, we can extend to 4/5-MMS allocation and even more. This fair allocation can be utilized in the real industry. This is a good combination of Computer Science and Game Theory.

References

[1] Jugal Garg, Peter McGlaughlin, and Setareh Taki. 2018. Approximating Maximin Share Allocations. In 2nd Symposium on Simplicity in Algorithms (SOSA 2019).

[2] Jugal Garg, Setareh Taki. 2019. An Improved Approximation Algorithm for Maximin Shares. In arXiv:1903.00029.