Implementation of Mixed Manna Framework

Ishita Chakravarthy
University of Massachusetts, Amherst
ichakravarth@umass.edu

1 Introduction

The project focuses on implementing an algorithm for the Mixed Manna problem involving goods and chores as detailed in Viswanathan and Zick [2023]. The algorithm allocates items based on agents' preferences, where goods are valued at $c > 1$, chores are valued at -1, and indifferent items are valued at 0. Specifically, for the class of Order-Neutral submodular (ONSUB) valuation functions over $c, 0, -1$, the algorithm computes a complete leximin allocation.

The implementation of the algorithm could have several applications in the real world- dividing chores among roommates, committees across faculty members. In these case, an individual might want to do a certain chore/committee, when another individual negatively values it. With an implementation of the algorithm, further extensions of creating an interface to input preferences can be explored.

For example, consider a scenario where two roommates, Alice and Bob, need to divide five household tasks between them: taking out the trash, cleaning the kitchen, cleaning the bathroom, vacuuming, and taking the dog for a walk. Alice enjoys cleaning, including the kitchen, bathroom, and vacuuming, valuing them at c. She is indifferent to taking the dog out for a walk (0) and hates taking out the trash (-1). Similar to Alice, Bob enjoys cleaning the kitchen and vacuuming (c), and, dislikes taking out the trash (-1). However, Bob finds cleaning the bathroom to be a chore (-1), but loves taking the dog for a walk (c). With this example, we see how the algorithm allocates the tasks between them, while ensuring that Alice and Bob are happy with the final allocation.

Through this project, some of my main contributions have been working on building an implementation of the algorithm with Python. Section 3 talks about the different stages of the implementation, and mentions individual contributions to each of the sections.

1.1 Related Work

Bérczi et al. [2024] presents results on the fair allocation of mixed goods and chores for a variety of valuation function classes, which represent preferences of agents over items. Hosseini et al. [2022] study the problem of fair allocation for a restricted subclass of additive valuations. Bhaskar et al. [2020] present an algorithm to compute allocations under doubly monotone valuations where each item is a good or a chore. As a natural extension to this, the current work explores order-neutral submodular valuation.

Existing tools for allocating mixed resources are limited in scope. KajiBuntan (Igarashi and Yokoyama [2024]), an implementation of Aziz et al. [2018], is an application for dividing goods and chores among agents. However, it can only handle two agents with additive valuation functions. This motivates the current implementation with larger number of agents, addressing this limitation, and covering a different class of valuation functions. It builds on the theoretical foundations of Yankee Swap established in Viswanathan and Zick [2023], and borrows implementation details from Navarrete Díaz et al. [2024].

2 Preliminaries

We have a set of $n$agents $N = [n]$and a set of m items $O = \{o_1, o_2, \ldots, o_m\}$. Each agent $i \in N$ has a valuation function $v_i : 2^O \to \mathbb{R}$; $v_i(S)$ denotes the value of the set of items S according to agent i. Given a valuation function v, we let $\Delta v(S, o) = v(S + o) - v(S)$ denote the marginal utility of adding the item o to the bundle S under v.

An allocation $X = (X_0, X_1, \ldots, X_n)$ is a $(n + 1)$-partition of the set of items O. Here, X_i denotes the set of items allocated to agent i, and X_0 denotes the set of unallocated items. The allocation is denoted with the matrices X^0, X^c, X^{-1} matrices of size $n + 1 \times m$. The matrix X^i is the allocation matrix under the valuation of i. Each row represents an item and the columns represent the agents. The last column of the matrices denote the unallocated items.

The exchange graph of an allocation X is a directed graph G(X) =(G, E) on the set of goods G. If a good g is in Xj for some agent $j \in N + 0$, then an edge exists from g to some other good $g' \in /Xj$ if $v_j(X_j - g + g') = v_j(X_j)$. In other words, there is an edge from g to g' if the agent who owns g can replace it with g' with no loss to their utility.

This algorithm addresses the allocation of items that are valued as chores (value -1), items that agents are indifferent to (value 0), and goods that an agent wants (value c). Adding any item to a bundle of an agent gives a marginal utility of {-1,0,c}. The valuation framework considers decreasing marginal gains, as agents receive more items, a good initially valued at c may lose its positive value, becoming neutral (0) or have negative value (-1). It is possible that an item initially valued at 0 may become a chore. This valuation also has a property of adding more of one item provides decreasing marginal gain (submodular). The order in which the agent receives items does not affect the valuation of the bundle (Order Neutral). We define this valuation class as $\{-1, 0, c\}$- Order Neutral submodular(ONSUB) valuations.

Our goal is to compute complete – allocations where $X_0 = \emptyset$ – and fair allocations. Here, the notion of fairness used is a leximin allocation- which is an allocation that maximizes the utility of the agent with the least utility; subject to that, it maximizes the second-least utility, and so on. Since the class of items also includes chores, it is important that all items are allocated, even if that reduces the total valuation of agents.

3 Implementation

Given a set of items and agents with $\{-1, 0, c\}$-ONSUB valuations, the algorithm outputs a complete leximin allocation of the items. It can be broken down into 3 main stages: Yankee Swap, Path Augmentation and Greedy Allocation. Initially, Yankee Swap stage allocates all items valued at 0 or c. Next, path augmentation refines the allocation by redistributing items and assigning items valued at c. Finally, the remaining chores are assigned using a greedy approach to complete the allocation. In each stage, we will look at the example of roommates and household tasks we defined in 1.

3.1 Stage 1: Yankee Swap

For a given set of agents and items, three allocation matrices X^0, X^c and X^{-1} are initialized to represent all items that are unallocated.

Initially, Yankee swap is run on the set of items which are valued at 0 or c, i.e. these items give the agents a marginal gain of 1. Yankee Swap is an algorithm for allocating items to agents based on binary submodular preferences, and outputs a leximin allocation. In each iteration, the agent with the lowest utility can either pick a new item from the remaining unallocated items or steal an item from another agent. The item can be stolen only if there is a way for the agent who is being stolen from to recover their utility, else the initial agent gets kicked out of the game. If an item is stolen, the agent who loses the item gets a turn, where they can either select a new item or steal an item from another agent. This process repeats until all items are allocated, or until no agent can increase their utility without making another agent worse off.

On running Yankee swap with the agents and preferences, the allocation matrix X^0 gets updated. At this stage, some items allocated in the X^0 matrix have a true valuation of c and not 0.

Yankee swap has been implemented in Viswanathan and Zick [2023] and the existing yankee swap algorithm is built on top of course allocation framework, which considers multiplicity of items. There were certain updates needed to be made to simplify the update allocations and exchange graph functions. Also, these updates also had to be dynamic, to ensure that the functions could be reused for further stages of the algorithm, specifically stage 2.

Looking at the example, Alice values cleaning (bathroom and kitchen), vaccuming, and taking the dog at 0 or above. Bob enjoys cleaning the kitchen and vacuuming , and taking the dog for a walk (c) at 0 or above. This is considering that both Alice and Bob have no household tasks initially to begin with. Running Yankee swap with these tasks, Alice gets assigned bathroom and vacuuming, whereas Bob gets assigned kitchen and taking the dog for a walk.

3.2 Stage 2: Path Augmentation

By the end of Phase 1, all items which can be positively valued have been allocated in the X^0 matrix. These items need to be redistributed between the X^0 and X^c matrices. The main goal of this step is to allocate as many items as possible with a valuation of c.

Initially, the allocation allocation matrices X^c and X^{-1} are initialized, with all items unassigned under both valuations. An exchange graph is initialized with items valued at c as nodes. A sync node of X_c^0 is added which has an edge to every node in the graph.

In stage 2A, a node is added for an agent chosen at random, and edges are added to every item that gives the agent a marginal gain of c. The min-weight pareto improving path from the agent to X_0^c is found, and the allocation is augmented along this path. The X^0 and X^c allocation matrices are updated and the weighted directed exchange graph is updated to reflect the new allocation. For two items in an agents bundle, if an edge exists from an item in its X^0 bundle to X^c bundle, it is weighted at 0.5. The remaining edges are given a weight of 1. The agent is removed from the exchange graph, and a new agent is chosen at random to add to the exchange graph. This process repeats until for all agents, there is no path which exists to the sync node of X_0^c. In the implementation, the agent is chosen in sequential order, whereas Cousins et al. [2023] chooses items in a sequential order.

Stages 2B and 2C address two distinct problems that might arise after Stage 2A. Only one of these stages is executed in a single iteration. Once Stage 2A is run, it is possible that one agent has been assigned significantly higher items, as they got chosen randomly more often. Stage 2B aims to tackle this problem. A min-weighted path from an item an agent with lower valuation wants to an item in the bundle of a higher valued agent is found. Stage 2C ensures that if two agents have a bundle size difference of 1, the agent with a lower index has higher priority. Similar to stage 2B, a min-weighted path is found from an item in the bundle of an agent with lower index(higher priority) to an item in the bundle of a higher index (lower priority) agent is found. Either one of stage 2B or 2C is run, the allocation is augmented along the path found in the one of the stages, and the allocation matrices are updated. Details on sequential allocation and its impact on Stage 2B and 2C are mentioned in Section 6.

If any paths were augmented i.e. if any items were exchanged in 2A/2B/2C, Stage 2 is re-run. If no paths are augmented, the algorithm proceeds to Stage 3.

The implementation of Stage 2A mainly borrowed from the previously implemented methods for Yankee swap. There were minor changes which needed to be made, including adding weights to exchange graphs, and updating both X^0 and X^c matrices. To ensure that the code remained readable and clean, various flags were passed to existing functions to add in minor updates. Stage 2B and 2C had to be implemented from scratch, and these were the most challenging phases to optimize, since they required computing all agents bundles, and picking the right pair of agents to run a path exchange.

In the example, so far, Alice was assigned bathroom and vacuuming, whereas Bob was assigned kitchen and taking the dog for a walk. However, all of these tasks were assigned with a value of 0. Now, redistributing and looking at sequential allocation, Alice picks a c valued item, which in this case would be bathroom. Bob picks kitchen, followed by Alice picking vacuuming, and bob finally picks walking the dog. though the allocation remained same after running phase 2, all of these items are now valued at c instead of 0. Since no more c valued items are left to be assigned we move on to the next phase.

3.3 Stage 3: Greedy Allocation

At the end of Phase 2, all items with values of 0 or c have been allocated. The remaining items have a marginal utility of -1 to every agent. These items are allocated greedily to agents to ensure a fair allocation. The current valuations of the agents are calculated, and items are assigned to the agent with the highest valued bundle. This process is repeated until all items valued at -1 are assigned. The greedy allocation ensures that no agent has a significantly higher valued bundle while another agent is allocated all chores.

Implementing the greedy algorithm was straightforward, since this did not require exchange graphs or any complicated computation. Initially, valuations of all agents bundles was found, and this was followed by assigning items greedily, while continuously updating the valuations.

In the example, so far, Alice was assigned bathroom and vacuuming, whereas Bob was assigned kitchen and taking the dog for a walk all with a value of c. Both Alice and Bob have a value of 2c for their assigned tasks. Now, the only task left which neither want is taking out the trash which is assigned to Bob. So, the final allocation is Alice is assigned bathroom, vacuuming and Bob is assigned kitchen, dog walking and trash. With this allocation, the valuations of the roommates of their respective bundles are 2c and 2c-1.

4 Code Example Run

The code takes input of a list of agents, each of class Agent, (list[Agent]) and list of items (string). For each Agent, the valuations are defined by a list of {desired items c} as well as {desired items 0}. Each Agent also has a maximum capacity for the number of items they value at c. The code only models additive valuations, further details about valuation function and its implementation limitations are detailed in 5.

On running the mixed-manna allocation algorithm, the output is the allocation matrices X^0, X^c and X^{-1}. The output allocation has the guarantee of a complete leximin allocation.

1 shows the run for the example discussed through this report, with the three allocation matrices. The item mapping is given by {0: Kitchen, 1: Bathroom, 2: Vacuuming, 3: Dog Walk, 4: Trash} with Agent 1 as Alice and agent 2 as Bob.

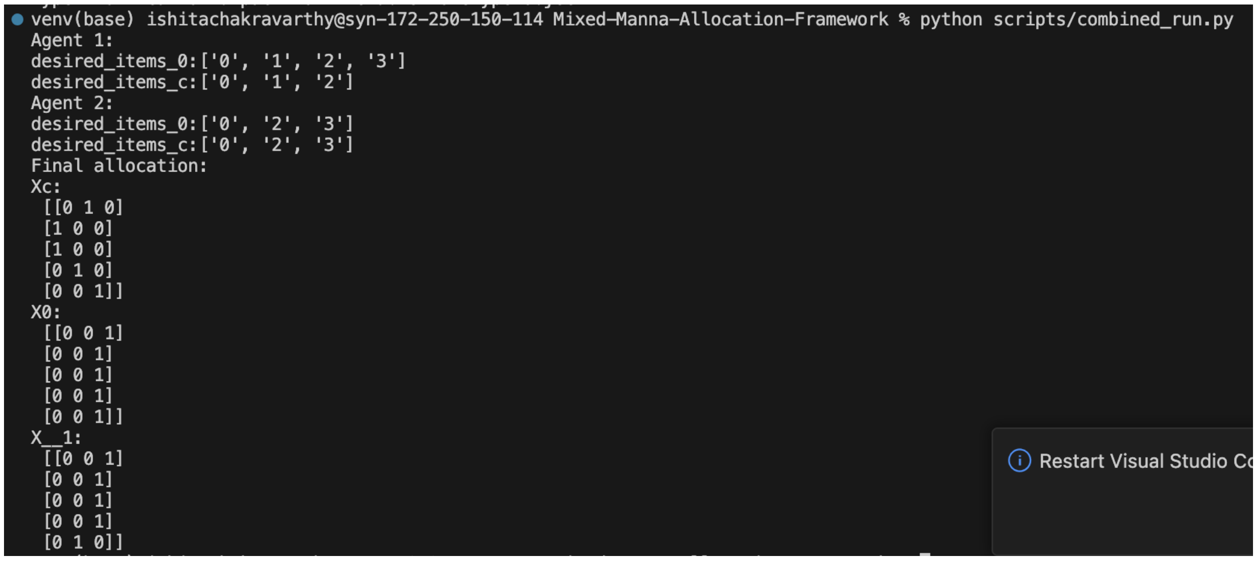
```
venv(base) ishitachakravarthy@syn-172-250-150-114 Mixed-Manna-Allocation-Framework % python scripts/combined_run.py
Agent 1:
desired_items_0:['0', '1', '2', '3']
desired_items_c:['0', '1', '2']
Agent 2:
desired_items_0:['0', '2', '3']
desired_items_c:['0', '2', '3']
Final allocation:
Xc:
 [[0 1 0]
 [1 0 0]
 [1 0 0]
 [0 1 0]
 [0 0 1]]
X0:
 [[0 0 1]
 [0 0 1]
 [0 0 1]
 [0 0 1]
 [0 0 1]]
X__1:
 [[0 0 1]
 [0 0 1]
 [0 0 1]
 [0 0 1]
 [0 1 0]]
```

Figure 1: Example run with 2 agents (Agent 1 and Agent 2) and 5 items ('0' to '4')

5 Valuation function

The mixed manna allocation algorithm takes as input items and agents with ONSUB valuations over the items. A challenge in implementing this algorithm was defining valuation functions of agents. For the implementation, we cannot assume that the valuation function is an oracle, since this would require intractable computation of the valuation for every agent over every potential bundle.

A naive implementation of the valuations encompasses additive valuations by using two lists- desired items 0 and desired items c. This covers a simple class of additive valuation functions. For the example discussed, it can been seen that Alice's valuation function can be defined by two lists desired items 0: [0: Kitchen, 1: Bathroom, 2: Vacuuming, 3: Dog Walk] and desired items c: [0: Kitchen, 1: Bathroom, 2: Vacuuming]. However, this restricts the change in value of items. Items that have a value of c will always have a value of c to an agent, regardless of the agent's bundle.

The below alternative to valuation functions aims to address that items might have different valuations, where items can switch between c and 0. Valuations can be defined with the help of a bipartite graph for 0 and c valued items along with a list for chores (refer Figure 2). Every agent has a list of slots, each of which is matched to one item in the final allocation. A bipartite matching is found between the slots and the items, and the items in slots are valued at c. The unallocated items are valued at 0. This implementation ensures that items initially valued at c can be valued at 0 based on a different agent bundle. However, it does not consider the shift of items valued at 0 shifting to being valued at -1.

This is still an open problem, of how we can design valuation functions that fully capture the dynamics of ONSUB preferences. Alternative approaches suggested include introducing additional graph structures, and having two bipartite graphs. Developing such models would require an elegant construction, since increasing the complexity of the valuation function would significantly increase runtime of the algorithm.

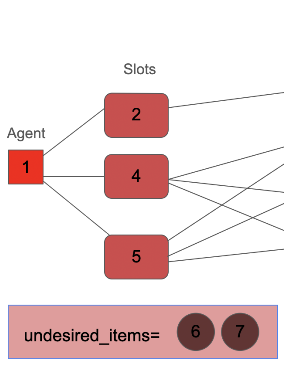

Figure 2: Proposed agent valuation function with bipartite graph and undesired_items list

6 Future Work

6.1 Sequential allocation

Pareto improving paths done in Stage 2A of the algorithm is very similar to Yankee swap. To find pareto improving paths, agents are chosen at random. This could lead to one agent consecutively picking items, and thus having a significantly higher valued bundle. Phase 2B works to fix this inequality in allocation. However, similar to Yankee swap, if agents pick items sequentially, this problem could potentially be avoided. Intuitively, if agents pick items sequentially, the only case in which an agent would have significantly higher valuation is when they value more items than other agents. Though this has been observed through a variety of examples, it has not been proven formally.

6.2 Implementation of the valuation function

Currently, only additive valuations are implemented with the help of lists as detailed in Section 5. Future work would include finding an approach to include the 0 to -1 switch, along with implementation of the bipartite graph structure. This would ensure that agent preferences are, in fact, {-1,0,c} ONSUB valued and enable more robust modeling of agent preferences, particularly for chores with negative utility.

References

Haris Aziz, Ioannis Caragiannis, Ayumi Igarashi, and Toby Walsh. Fair allocation of combinations of indivisible goods and chores, 2018.

Umang Bhaskar, A. R. Sricharan, and Rohit Vaish. On approximate envy-freeness for indivisible chores and mixed resources, 2020.

Kristóf Bérczi, Erika R. Bérczi-Kovács, Endre Boros, Fekadu Tolessa Gedefa, Naoyuki Kamiyama, Telikepalli Kavitha, Yusuke Kobayashi, and Kazuhisa Makino. Envy-free relaxations for goods, chores, and mixed items. *Theoretical Computer Science*, 1002:114596, June 2024. ISSN 0304-3975. doi: 10.1016/j.tcs.2024.114596. URL `http://dx.doi.org/10.1016/j.tcs.2024.114596`.

Cyrus Cousins, Vignesh Viswanathan, and Yair Zick. The good, the bad and the submodular: Fairly allocating mixed manna under order-neutral submodular preferences, 2023.

Hadi Hosseini, Sujoy Sikdar, Rohit Vaish, and Lirong Xia. Fairly dividing mixtures of goods and chores under lexicographic preferences, 2022.

Ayumi Igarashi and Tomohiko Yokoyama. Kajibuntan: A house chore division app. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(13):16449–16451, Jul. 2024. doi: 10.1609/aaai.v37i13.27075. URL `https://ojs.aaai.org/index.php/AAAI/article/view/27075`.

Paula Navarrete Díaz, Cyrus Cousins, George Bissias, and Yair Zick. Deploying fair and efficient course allocation mechanisms. 2024.

Vignesh Viswanathan and Yair Zick. Yankee swap: a fast and simple fair allocation mechanism for matroid rank valuations. In *Proceedings of the 22nd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2023.