Project report Algorithmic Game Theory

Emanuele Conforti - 252122
Jacopo Garofalo - 252093
Gianmarco La Marca - 252256

January 2025

Contents

1 Introduction

1.1 Assignment

We consider a setting with:

- A set $N = \{1, 2, \ldots, n\}$ of agents.
- A set $S = \{s_1, \ldots, s_m\}$ of skills.

Each agent $i \in N$ is associated with a set of skills $S_i \subseteq S$.

- There is no a-priori specified number of skills that each agent owns. For example, one agent might have just one skill, while another might possess all the available skills.

There is a task t to be completed, and all the skills in S are required to this end. Consequently:

- Agents are required to collaborate with each other.
- A number of strategic issues come into play.

1.2 Example Instance

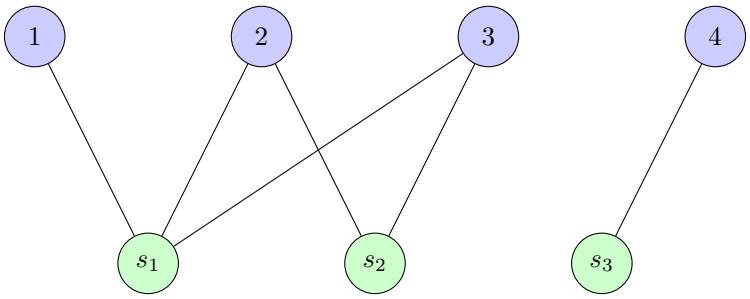
Consider the following example instance:

- $N = \{1, 2, 3, 4\}$ (4 agents).
- $S = \{s_1, s_2, s_3\}$ (3 skills).

The assignment of skills to agents is graphically depicted below. An edge represents the ownership of a skill by the corresponding agent.

Graphical Representation

The instance is depicted as follows:



1.3 Problem analysis

We can consider the game depicted above as a **Coalitional game** or **cooperative game** in which agents can form coalitions to achieve a common goal and collaborate by making *binding agreements* each other.
So in our case, the agents must form a coalition by combining their skills to complete the task.
Since the final reward is given to the coalition as a whole, we can say that we are in the case of a **Transferable utility game**, that is the agents must decide how to distribute the reward among themselves.
Furthermore, the payoff of the coalition only depends on the single agents that are part of it, and not on other coalitions. Thus, this is also a **Characteristic function game**.

In a characteristic function game, we can represent the utility of each agent as a function, called indeed *characteristic function*. This function takes as input a coalition (subset of agents) and returns the payoff that the coalition's agents can earn by working together.
For our instance, the characteristic function v is defined as follows:

$$v = \begin{cases} v(\{1\}) = 0, & v(\{2\}) = 0, \\ v(\{3\}) = 0, & v(\{4\}) = 0, \\ v(\{1,2\}) = 0, & v(\{1,3\}) = 0, \\ v(\{1,4\}) = 0, & v(\{2,3\}) = 0, \\ v(\{2,4\}) = 100, & v(\{3,4\}) = 100, \\ v(\{1,2,3\}) = 0, & v(\{1,2,4\}) = 100, \\ v(\{1,3,4\}) = 100, & v(\{2,3,4\}) = 100, \\ v(\{1,2,3,4\}) = 100. \end{cases}$$

In the code, we decided to generalize the characteristic function v by implementing a function that calculate v from a set of agents, a set of skills, a mapping between agents and skills and a reward:

```
calculate_characteristic_function(agents, skills, agents_skills_map, reward)
```

In this way, we have to calculate all the possible coalitions and their associated reward only one time in the entire project, saving computational power.

2 Point 1

2.1 Task

In the first part, we can assume that completing the task t leads to a reward of 100$. Then, we want to compute the Shapley value associated with the agents as a fair way to distribute that reward among them. And, finally, check whether the Shapley value is in the core of the coalitional game induced by the setting.

2.2 Superadditivity

Before we proceed with the computation of the Shapley value, we check if the game is **superadditive**. A game is superadditive if the following holds for any two disjoint coalitions $S, T \subseteq N$:

$$v(S \cup T) \geq v(S) + v(T).$$

In our case, the game is superadditive, as the following examples show:

$$v(\{2,4\}) \geq v(\{2\}) + v(\{4\})$$
$$v(\{2,4\}) = 100, \quad v(\{2\}) = 0, \quad v(\{4\}) = 0$$
$$v(\{3,4\}) \geq v(\{3\}) + v(\{4\})$$
$$v(\{3,4\}) = 100, \quad v(\{3\}) = 0, \quad v(\{4\}) = 0$$
$$v(\{2,3,4\}) \geq v(\{2,3\}) + v(\{4\})$$
$$v(\{2,3,4\}) = 100, \quad v(\{2,3\}) = 0, \quad v(\{4\}) = 0.$$

In this case, notice that we checked superadditivity using also the **empty coalition**, since there are some coalitions that can be created from the union with the empty coalition, like:

$$v(\{2\}) = v(\{2\}) \cup v(\{\})$$

Superadditivity is a good property for a game, because we can say that the agents have incentives to work together than working alone and a bigger coalition is preferred to a smaller one (the grand coalition, that is the coalition with all the agents, is often the optimal choice for the game).

2.3 Shapley value

We use the **Shapley value** as a way to distribute the reward among the agents. By using the Shapley value, we try to divide the payoff in a *fair way*. The Shapley value distribution is based on **fairness**, meaning that the payoff is divided based on the agents' *average marginal contributions* to the coalition. Furthermore, the Shapley values is the only payoff distribution scheme that satisfies the following properties:

- **Efficiency:** $\varphi_1 + \cdots + \varphi_n = v(N)$
- **Dummy:** if i is a dummy, $\varphi_i = 0$
- **Symmetry:** if i and j are symmetric, $\varphi_i = \varphi_j$
- **Additivity:** $\varphi_i(G_1 + G_2) = \varphi_i(G_1) + \varphi_i(G_2)$

In the *Python notebook* we implemented the Shapley value of agent i with a characteristic function v using the following formula:

$$\phi(i, v) = \sum_{C \subseteq N \setminus \{i\}} \frac{(|N| - |C| - 1)! \cdot |C|!}{|N|!} \cdot [v(C \cup \{i\}) - v(C)]$$

As a result, the Shapley values of the agents in the instance are given as:

$$\phi(1) = 0, \quad \phi(2) = 16.67, \quad \phi(3) = 16.67, \quad \phi(4) = 66.67.$$

As we have done with superadditivity, also in this case we take into account the empty coalition when calculating the Shapley value. We decided to proceed this way because the empty coalition is used in the calculation of the marginal contribution of other coalitions. For example, using the

formula above, in the computation of the Shapley value of agent 4 we use the empty coalition as follows:

$$\text{For } C = \emptyset, \quad v(\emptyset \cup \{4\}) = v(\{4\}) = 0,$$

$$\text{marginal contribution} = v(\emptyset \cup \{4\}) - v(\{4\}) = 0 - 0 = 0.$$

$$\text{Hence,} \quad \frac{(4-0-1)! \cdot 0!}{4!} = \frac{3! \cdot 1}{4!} = \frac{6}{24} = 0.25.$$

Agent 1: Dummy Player

An agent i is **dummy** if for all coalitions $S \subseteq N \setminus \{i\}$, the following holds:

$$v(S \cup \{i\}) = v(S).$$

For agent 1, in the instance we observe:

$$\begin{aligned} v(\{1,2\}) = v(\{2\}) = 0, \quad v(\{1,3\}) = v(\{3\}) = 0, \\ v(\{1,4\}) = v(\{4\}) = 0, \quad v(\{1,2,4\}) = v(\{2,4\}) = 100. \end{aligned}$$

Agent 1 contributes nothing to any coalition, and its Shapley value is $\phi(1) = 0$. Therefore, 1 is a **dummy player**.

Agents 2 and 3: Symmetric Players

Two agents i and j are **symmetric** if for all coalitions $S \subseteq N \setminus \{i, j\}$, the following holds:

$$v(S \cup \{i\}) = v(S \cup \{j\}).$$

For agents 2 and 3, in the instance we observe:

$$v(\{2\}) = v(\{3\}) = 0, \quad v(\{2,4\}) = v(\{3,4\}) = 100, \quad v(\{1,2,4\}) = v(\{1,3,4\}) = 100.$$

Agents 2 and 3 contribute equally to all coalitions, and their Shapley values are equal:

$$\phi(2) = \phi(3) = 16.67.$$

Thus, 2 and 3 are **symmetric players**.

Agent 4: Pivotal Player

An agent i is **pivotal** if its contribution is critical to achieving any nonzero coalition value. That is, for any coalition $S \subseteq N \setminus \{i\}$, we have:

$$v(S \cup \{i\}) > v(S) \quad \text{if and only if } S \cap \{i\} = \emptyset.$$

For agent 4, in the instance we observe:

$$\begin{aligned} v(\{2,4\}) = 100, \quad v(\{2\}) = 0, \\ v(\{3,4\}) = 100, \quad v(\{3\}) = 0, \\ v(\{2,3,4\}) = 100, \quad v(\{2,3\}) = 0. \end{aligned}$$

Agent 4 is indispensable for achieving any coalition value of 100. Its Shapley value, $\phi(4) = 66.67$, reflects its dominant role in the game. Thus, 4 is a **pivotal player**.

2.4 Computational complexity

The Shapley value can be computed in $O(2^n)$ **time complexity**. While the characteristic function v uses $O(2^n)$ **space complexity**.
This means that the computation of the Shapley value is feasible for small instances, but it becomes computationally expensive as the number of agents increases.
Hence, for large instances, we can use other algorithms, like the **Monte Carlo sampling approximation** to estimate the Shapley value.

2.5 Monte Carlo sampling approximation

Monte Carlo sampling approximation is a method to estimate the Shapley value by randomly sampling permutations of the agents and averaging the marginal contributions of the agents over these samples.

Hence, with this method, we can estimate the Shapley value in a reasonable time for large instances.

In particular, we can compute the Shapley value in **polynomial time** by using Monte Carlo sampling approximation, while the spatial complexity remains the same since it depends on the characteristic function v, that always uses $O(2^n)$ **space complexity**.

Obviously, Monte Carlo sampling is an *approximation algorithm*, so it is not as accurate as the exact computation of the Shapley value. The accuracy of the approximation depends on the number of samples taken. The more samples we take, the more accurate the approximation will be.

As you can notice in the code, we adapted Monte Carlo sampling approximation of the Shapley value for the instance of the project. We used 10000 samples to estimate the Shapley value and used the **mean absolute percentage error** as error function for comparing the approximation to the exact Shapley value computation. In the end, we obtained a good approximation with about 3% of error.

Then, we applied Monte Carlo sampling approximation to a larger instance with 20 agents and 6 skills, building a random mapping between agents and skills and the corresponding characteristic function, and we obtained a good approximation with about 5% of error.

Furthermore, we also tested the sampling approximation with the same 20 players instance but with different sampling (100, 500, 1000, 2000, 5000, 10000, 15000 and 20000). As a result, we observed that the error decreases as the number of samples increases, as expected: from 2000 samples on, the error is quite always under 15% and from 10000 samples on, the error is quite always under 10%. Note that, since the permutations are chosen randomly at each iteration, the error can vary from run to run, depending on the permutations used.

2.6 Core

The **core** of a game is the set of all **stable outcomes**, that is outcomes that no coalition wants to deviate from. To check if an outcome is in the core of the game we have to verify that for every coalition $S \subseteq N$ the following holds:

- **positivity:** the payoff is positive for each agent;
- **efficiency:** all value is allocated;
- **stability:** no coalition wants to deviate.

$$\text{core}(G) = \left\{ \mathbf{x} \;\middle|\; \begin{array}{ll} x_i \geq 0 \quad \forall i \in N & \textbf{(Positivity)} \\ \displaystyle\sum_{i \in N} x_i = v(N) & \textbf{(Efficiency)} \\ \displaystyle\sum_{i \in C} x_i \geq v(C) \quad \forall C \subseteq N & \textbf{(Stability)} \end{array} \right\}$$

In the code, we use the Shapley value as outcome and check if it's in the core of the game. To do so, we implemented the following function:

- `is_in_the_core(outcome, characteristic_function, players)`: takes as input the agents Shapley values (outcome), the characteristic function v and the set of agents N and returns whether the outcome is in the core of the game. It calls the other three functions.
- `is_positive(outcome)`: checks if the payoff is positive for each agent.
- `is_efficient(outcome, characteristic_function, players)`: checks if all the value (reward) is allocated in the outcome by summing up the Shapley values of all the agents.

- `is_stable(outcome, characteristic_function)`: for each coalition C checks if the sum of the Shapley values of the agents in C is greater or equal to the value (in the characteristic function) of the coalition.

As a result, the Shapley value **is not in the core of the game**, even though it's *positive* and *efficient*. But actually, it is not *stable*, as the following counterexample shows:

$$\text{for coalition } \{3,4\} \text{ we have:} \quad \phi(3) + \phi(4) = 83.33 < v(\{3,4\}) = 100.$$

In this special case, the players want to deviate from the Shapley value distribution, as the coalition {3, 4} could earn more by working together.

3 Point 2

3.1 The setting

In the second part, we expand this setting by introducing the possibility for each agent $i \in N$ to decide whether to join the coalition or not. In addition, each agent now is subject to a fixed cost c_i to join the group. This, combined with the assumption that each agent wants to maximize their own utility, in this case to not have a negative revenue, shifts the game setting from a Cooperative to a Non-Cooperative framework. The fixed costs in this setting are:

$$c = \begin{cases} c_1 = 10 \\ c_2 = 10 \\ c_3 = 20 \\ c_4 = 40 \end{cases}$$

Now the revenue divided according to the Shapley values of the induced game must at least cover the participation of the single agent if he decides to join the group.

3.2 Nash Equilibrium

The crucial point is to use the concept of the Nash Equilibrium applied to the participation of the agents in the new setting. In particular, the Nash equilibrium will occur when no agent can improve their outcome by unilaterally changing their strategy, assuming others keep theirs unchanged.

Formally, it is defined as a set of stable actions $a = <a_1, ..., a_n>$, where a_i is the choice of agent i and each agent's choice satisfies the following condition:

$$a_i \in BR(a_{-i})$$

where $BR(a_{-i})$ is the best response of agent i given the set of actions of all other agents, a_{-i}. In this way, no agent has a profitable deviation from their current strategy.

In this setting, finding a (pure) Nash equilibrium involves verifying two main conditions:

- no member of the coalition would prefer to leave, as their received value $\phi(i, v)$ is always greater or equal to their cost c_i.

$$\phi(coalition\_player, v) \geq c_{coalition\_player}$$

- no external player has an incentive to join the coalition, as their cost c_i is always greater than the value they would receive $\phi(i, v)$.

$$c_{external\_player} > \phi(external\_player, v)$$

3.3 Task

Our objective for this task is to compute all the Nash Equilibrium (if any). To achieve this we iterate through all the possible coalitions of players and verify the two conditions above while checking the validity of each new coalition using the characteristic function.

So, with respect to the costs of the single agents and the Shapley value computed in the first part of the project, there will be only two coalitions in a (pure) Nash Equilibrium:

- the coalition (2, 4), where there are only players whose received value is greater or equal to their costs from the very beginning, so they will be always interested to join the coalition. Moreover, the associated skills of the two agents are sufficient to complete the task, so they can be payed back from the beginning. On the other hand, the external players (1, 3) will incur in a negative revenue in all the participation cases, so they will never join the group.

$$revenue = \begin{cases} r_1 = \phi(1, v) - c_1 = 0 - 10 < 0 \\ r_2 = \phi(1, v) - c_2 = 16.67 - 10 = 6.67 \\ r_3 = \phi(1, v) - c_3 = 16.67 - 20 < 0 \\ r_4 = \phi(1, v) - c_4 = 66.67 - 40 = 26.67 \end{cases} \rightarrow partecipation = \begin{cases} N_1 : NO \\ N_2 : YES \\ N_3 : NO \\ N_4 : YES \end{cases}$$

- the coalition (), representing the empty set, is a particular group in which the task isn't completed due to the lack of the required skills from the beginning, so there is no revenue that can be distributed to the agents. This implies that no agent can be payed back, so all the agents will not join the group in any case. Moreover, if every agent try to join the group, the result will be the same because no one can complete the task alone. Finally, the decisions of the agents to not join will not change, so there is a Nash Equilibrium.

$$revenue = \begin{cases} r_1 = \phi(1,v) - c_1 = 0 - 10 < 0 \\ r_2 = \phi(1,v) - c_2 = 0 - 10 < 0 \\ r_3 = \phi(1,v) - c_3 = 0 - 20 < 0 \\ r_4 = \phi(1,v) - c_4 = 0 - 40 < 0 \end{cases} \rightarrow partecipation = \begin{cases} N_1 : NO \\ N_2 : NO \\ N_3 : NO \\ N_4 : NO \end{cases}$$

4 Point 3

4.1 Task

Assume that all agents participate to the setting, but they might cheat on the cost c_i, and consider a setting where a mechanism has to identify a group of agents that is capable of completing the task with the minimum overall cost. Then, compute a payment scheme that provides incentives to truthfully report such costs.

4.2 VCG Auction Setting

In order to enforce agents to report their true costs, the setting has been adapted to a classical "Combinatorial Auction" where we use VCG payments to incentive truthfulness. In particular, in this kind of setting we have:

1. "bidders", which are the agents
2. "item bundle", which is the set of skills each agent has
3. "bidder's valuation on item bundle", which is the cost reported by each agent

4.2.1 Allocation

We define an ***allocation*** as a list of triples of the form:

$$[\ (\text{"}bidder\text{"},\text{"}item\ bundle\text{"},\text{"}bidder's\ valuation\ on\ item\ bundle\text{"}),\ ...\ ]$$

4.2.2 Admissible Allocation

An allocation is said to be ***admissible*** if and only if the union set of all the item bundles in each of its triples is equal to the set S of all the skills.

4.2.3 Point 3 Input Example

Following the given example, the input of point 3 will be a list of the following triples of preference:

- (1, [s1], 10$)
- (2, [s1, s2], 10$)
- (3, [s1, s2], 20$)
- (4, [s3], 40$)

For this example the only admissible allocations are the following:

- [(2, [s1, s2], 10$), (4, [s3], 40$)]
- [(3, [s1, s2], 20$), (4, [s3], 40$)]
- [(1, [s1], 10$), (2, [s1, s2], 10$), (4, [s3], 40$)]
- [(1, [s1], 10$), (3, [s1, s2], 20$), (4, [s3], 40$)]
- [(2, [s1, s2], 10$), (3, [s1, s2], 20$), (4, [s3], 40$)]
- [(1, [s1], 10$), (2, [s1, s2], 10$), (3, [s1, s2], 20$) (4, [s3], 40$)]

4.3 Code description

The code uses 2 main functions, which are the following:

1. "compute_winner", which identifies a group of agents (that we call *winners* of the auction) that is capable of completing the task with the minimum overall cost
2. "VCG_auction", which computes the VCG payment for each *winner* of the auction

4.3.1 compute_winner

This method takes as input the list of preferences of the form:

$$[\ (\text{"}bidder\text{"},\text{"}item\ bundle\text{"},\text{"}bidder's\ valuation\ on\ item\ bundle\text{"}),\ ...\ ]$$

And outputs the allocation with minimum overall cost. To do so, we generate all the admissible allocations (through the function "generate_only_admissible", explained later) and then we apply a linear search algorithm to find the allocation with minimum overall cost. The pseudo-code for this function is the following:

```
function compute_winner(list_of_triples) {
    optimal_allocation = new Array()
    optimal_value = MAX_INTEGER

    for each allocation in generate_only_admissible(list_of_triples) {
        val = compute_value(allocation)
        if (val < optimal_value)  {
            optimal_allocation = allocation
            optimal_value = value
        }
    }

    return (optimal_allocation, optimal_value)
}
```

4.3.2 generate_only_admissible

This method implements backtracking to generate all the admissible allocations for a given set of preferences. The input is a set of preferences of the form:

$$[\ (\text{"}bidder\text{"},\text{"}item\ bundle\text{"},\text{"}bidder's\ valuation\ on\ item\ bundle\text{"}),\ ...\ ]$$

To implement the backtracking algorithm, an auxiliary bitmap that maps each bit to a triple in the input set of preferences has been used.
Internally, the bitmap keeps track of which triple to include (bit 1) or exclude (bit 0) from the currently generated *allocation*. By generating all permutations of such bitmap (from 0 to 2^n, with n the number of triples in input), we are generating one by one all the *allocations* in the powerset of the input.
In this way, for each *allocation* we can check at generation time if it's admissible, and thus whether to store it or not.
The pseudo-code for this function is the following:

```
funciton generate_only_admissible(list_of_triples) {
    ret = new Array()

    bitmap = new Array(list_of_triples.length) // inizialized to all 0s

    while has_next_solution(bitmap) {
        solution, bitmap = next_solution(bitmap, list_of_triples)
        if (is_admissible(solution)) {
            ret.add(solutiom)
        }
    }

    return ret
}
```

4.3.3 VCG_auction

This method takes as input a set of preferences, which is a list of the form:

$$[\ (\text{"}bidder\text{"},\text{"}item\ bundle\text{"},\text{"}bidder's\ valuation\ on\ item\ bundle\text{"}),\ ...\ ]$$

And computes a payment scheme that incentives to truthfully report the costs given in input as "bidder's valuation on item bundle".
It precisely implements the VCG payment formula, which is the following:

$$p_i(\omega^*) = \min_{\omega \in \Omega} \sum_{j \neq i \in N} v_j(\omega) - \sum_{j \neq i} v_j(w^*)$$

Where the optimal allocation ω^* is defined as follows:

$$\omega^* = argmin_{\omega \in \Omega} \sum_{i \in N} v_i(\omega)$$

The implementation of this function internally invokes the function *compute_winner* to find the optimal allocation ω^* and in the computation of the first term of the VCG payment formula.

After we compute the payment for each agent, if the sum of the payments exceeds the budget value (global constant), the coalition is rejected and the methods prints the error.

The pseudo-code for this function is the following:

```
function VCG_auction(list_of_triples) {
    optimal_allocation, total_value = compute_winner(list_of_triples)
    ret = new Array()
    total_payment = 0

    for each (bidder, itemset, value) in optimal_allocation {
        list_copy = deep_copy(list_of_triples)
        list_copy.remove(bidder)    // current bidder doesn't partecipate

        sum_values_without_current = compute_winner(list_copy)[1]
        sum_values_other_bidders = total_value - value

        payment = sum_values_without_current - sum_values_other_bidders

        total_payment += payment
        ret.add( (bidder, itemset, payment) )
    }

    if (total_payment > budget) {
        print("Total payment exceeds the budget.")
        system.exit()
    }

    return ret
}
```

4.3.4 Pivotal Agent Problem

As described before, we define a ***pivotal agent*** as a player whose participation to the coalition determines the coalition value to be nonzero.

In the VCG auction setting the presence of such players creates a problem when computing the first term of the VCG payment, which is the following:

$$\min_{\omega \in \Omega} \sum_{j \neq i \in N} v_j(\omega)$$

This term represents the value of the optimal allocation (the one with minimum cost) if the agent "i" **does NOT participate** to the auction.

It is clear that, if the agent "i" appears to be a *pivotal agent*, all the allocations have coalition value equal 0 and thus none of them is admissible.

To solve this issue, we decided that if an agent "i" appears to be a *pivotal agent* then:

$$p_i(\omega^*) = c_i$$

The VCG_auction function pseudo-code is now modified as follows:

```
function VCG_auction(list_of_triples) {
    optimal_allocation, total_value = compute_winner(list_of_triples)
    ret = new Array()
    total_payment = 0

    for each (bidder, itemset, value) in optimal_allocation {
        list_copy = deep_copy(list_of_triples)
        list_copy.remove(bidder)    // current bidder doesn't partecipate

        sum_values_without_current = compute_winner(list_copy)[1]
        sum_values_other_bidders = total_value - value

        payment = 0

        if (pivotal_agent(bidder)) {
            payment = value
        } else {
            payment = sum_values_without_current - sum_values_other_bidders
        }

        total_payment += payment
        ret.add( (bidder, itemset, payment) )
    }

    if (total_payment > budget) {
        print("Total payment exceeds the budget.")
        system.exit()
    }

    return ret
}
```

The output returned by this function for the given example:

- (1, [s1], 10$)
- (2, [s1, s2], 10$)
- (3, [s1, s2], 20$)
- (4, [s3], 40$)

Is the following:

$$[\ (2,\ [s2,\ s1],\ 20),\ (4,\ [s3],\ 40)\ ]$$

Thus we have $p_2 = 20$ and $p_4 = c_4 = 40$.

5 Conclusion

- **Shapley value and core**: we first verified that the game was superadditive and so it was. Hence, the players would have incentive to collaborate creating coalitions.

  Then we computed the Shapley value for the agents of the instance and obtained the following results: $\phi(1) = 0$, $\phi(2) = 16.67$, $\phi(3) = 16.67$, $\phi(4) = 66.67$. We observed that agent 1 is a dummy player, agents 2 and 3 are symmetric players and agent 4 is a pivotal player.

  We also compared the exact computation of the Shapley value with Monte Carlo sampling approximation and we noticed that for larger instances Monte Carlo approximation is way better than the classic Shapley value computation and the approximation's accuracy increases as the number of samples increases.

  Eventually we checked if the Shapley value was in the core of the game. We observed that the Shapley value is not in the core of the game, as it is not a stable outcome, since the coalition $\{3, 4\}$ could earn more by working together. Anyway, the Shapley value is still positive and efficient.

- **Nash equilibrium**: due to the change to the setting, we switch to a non-cooperative game framework, so we had to explore strategic stability among self-interested agents. The brute-force methodology used to identify Nash equilibrium can work, but if we have a larger setting, advanced optimization techniques or strong assumptions can be necessary, like using specific coalitions with certain agents or excluding coalitions where the principal task is not completed from the beginning, even if the empty set will be excluded with this approach.

- **VCG auction**: in order to compute a payment scheme to incentive agents in reporting their true costs, we changed the setting into a **VCG auction**. Mainly two functions were used to complete the task: ***compute_winner*** (that finds the optimal allocation for an input list of triples) and ***VCG_auction*** (that computes the VCG payment for each bidder in the optimal allocation). The **admissible allocations** were generated with backtracking and the **pivotal agent problem** was solved giving as payment for pivotal agents the cost they declared. The overall **time complexity** is exponential in the number of agents since the problem is NP-Hard.