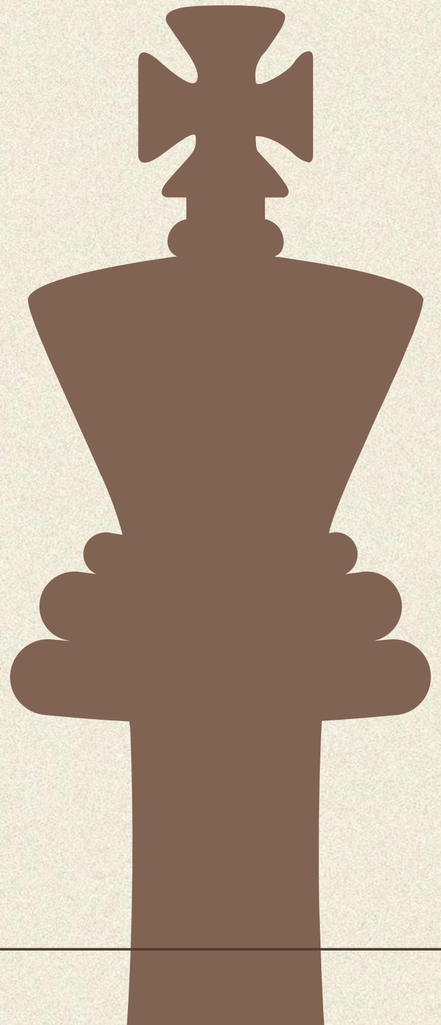


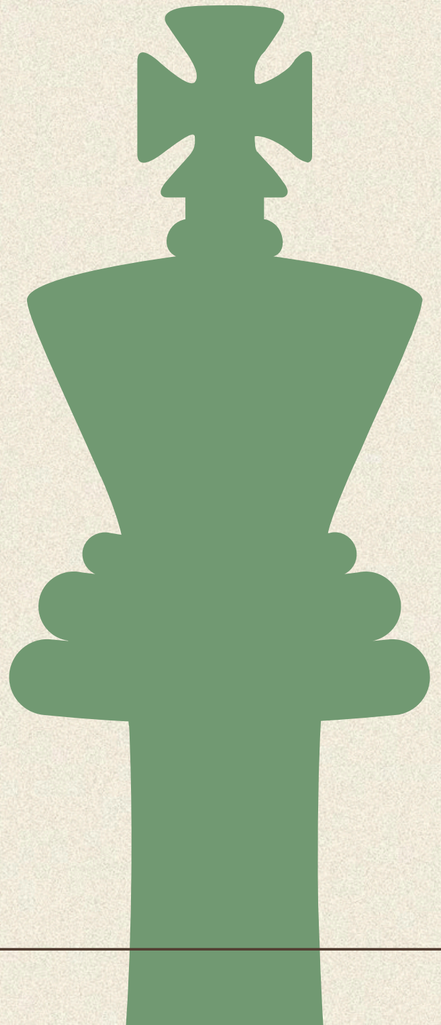
AGT Project

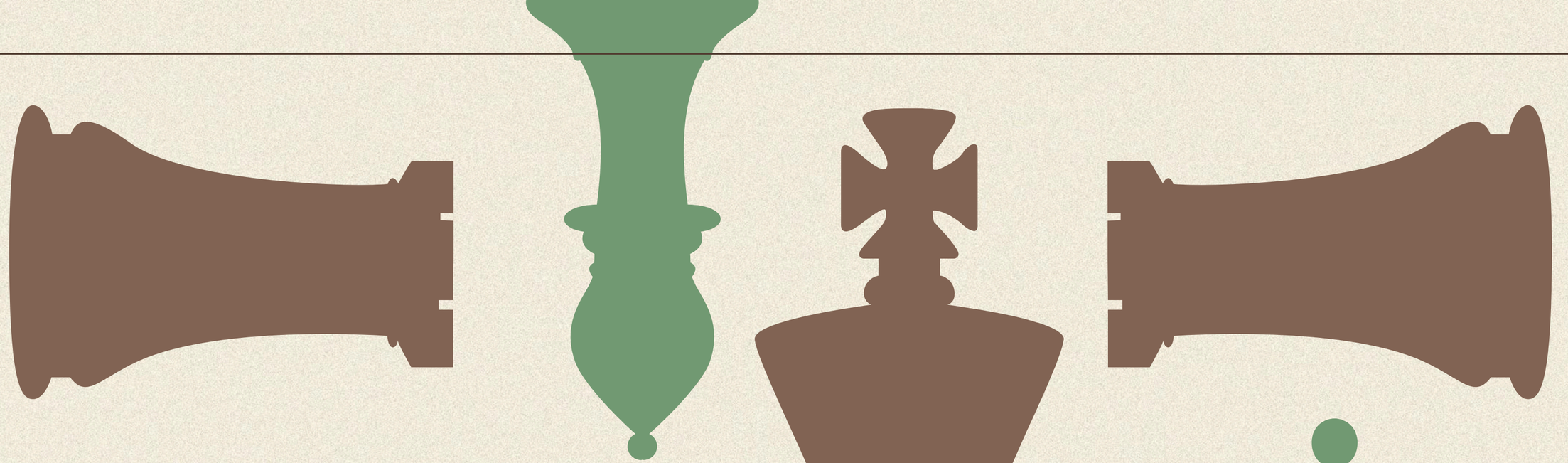
Students:

Emanuele Conforti

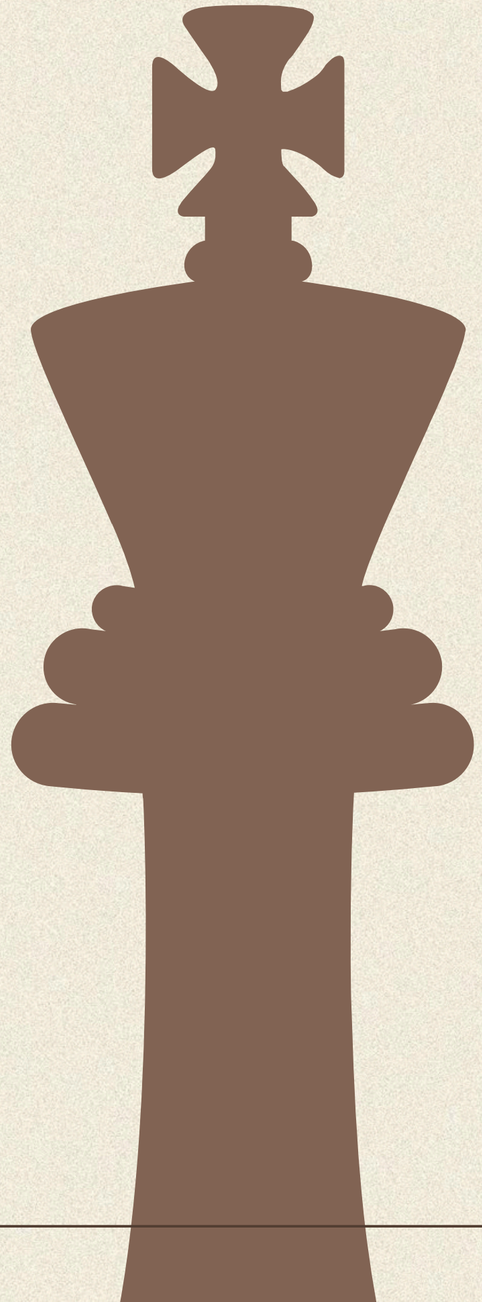
Jacopo Garofalo

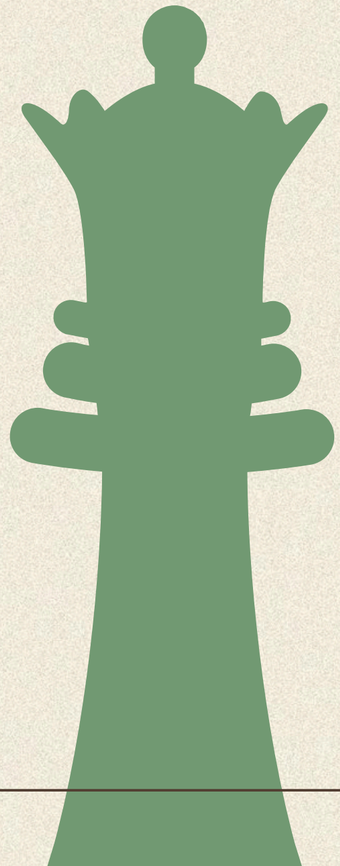
Gianmarco La Marca





Point 1





Assignment

Set $N=\{1,2,....,n\}$ of agents

Set $S = \{s_1,....,s_m\}$ of skills

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

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

Agents are required to collaborate with each other

Design choices

- **powerset(List)**: generate the powerset of List (with the empty set)
  - Useful for bruteforcing
  - Some variants implemented

- **calculate_characteristic_function(agents, skills, agents_skills_map, reward)**: generate the characteristic function *v* from a set of agents, a set of skills, a mapping from agents to skills and a reward

Task

- Task t leads to a reward of 100$
- Compute the Shapley value associated with the agents
- Check whether the Shapley value is in the core of the coalitional game

Superadditivity

- A game is **superadditive** if the following holds for any two disjoint coalitions S, T ⊆ N:

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

- **Why checking superadditivity?**
  - Because we can say that the agents have incentives to work together than working alone

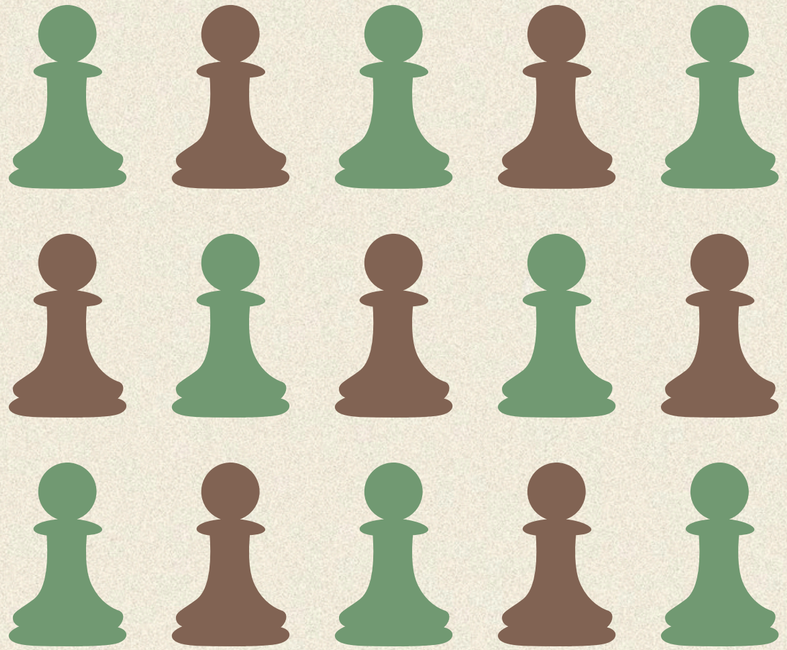
- The game is superadditive, as the following examples show:
  - $v(\{2, 4\}) \geq v(\{2\}) + v(\{4\})$
  - $v(\{2, 4\}) = 100$, $v(\{2\}) = 0$, $v(\{4\}) = 0$

  - $v(\{3, 4\}) \geq v(\{3\}) + v(\{4\})$
  - $v(\{3, 4\}) = 100$, $v(\{3\}) = 0$, $v(\{4\}) = 0$

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

Superadditivity

```
def is_superadditive(v, players):
    power_set = powerset(players)

    for i in power_set:
        for j in power_set:
            # check that the two coalitions i and j are disjoint
            if len(i.intersection(j)) == 0:
                sum = v[i] + v[j]
                union = v[i.union(j)]
                if union < sum:
                    return False

    return True
```



Shapley value

Classic computation

Using an exact formula, computed in **exponential time**

Monte Carlo sampling approximation

Approximation, so the result is not exact but it’s computed in **polynomial time**

Classic Shapley value

- **Shapley value** is a solution concept in coalitional games
- It *fairly* distributes the total value of a game among players
- It’s based on the players’ marginal contributions

```
def shapley_value(players, player, v):
  power_set = powerset(players - {player})
  shapley = 0

  for c in power_set:
    first_part = factorial(len(c)) * factorial(len(players) - len(c) - 1)
    second_part = first_part / factorial(len(players))
    third_part = v[c.union({player})] - v[c]
    shapley += second_part * third_part

  return round(shapley, 2)
```

Formula

$$\varphi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(n - |S| - 1)!}{n!} (v(S \cup \{i\}) - v(S))$$

Monte Carlo sampling

- It’s 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

Pseudo code

def monte_carlo_shapley(players, v, num_samples):
shapley_dict = {player: 0 for player in players}

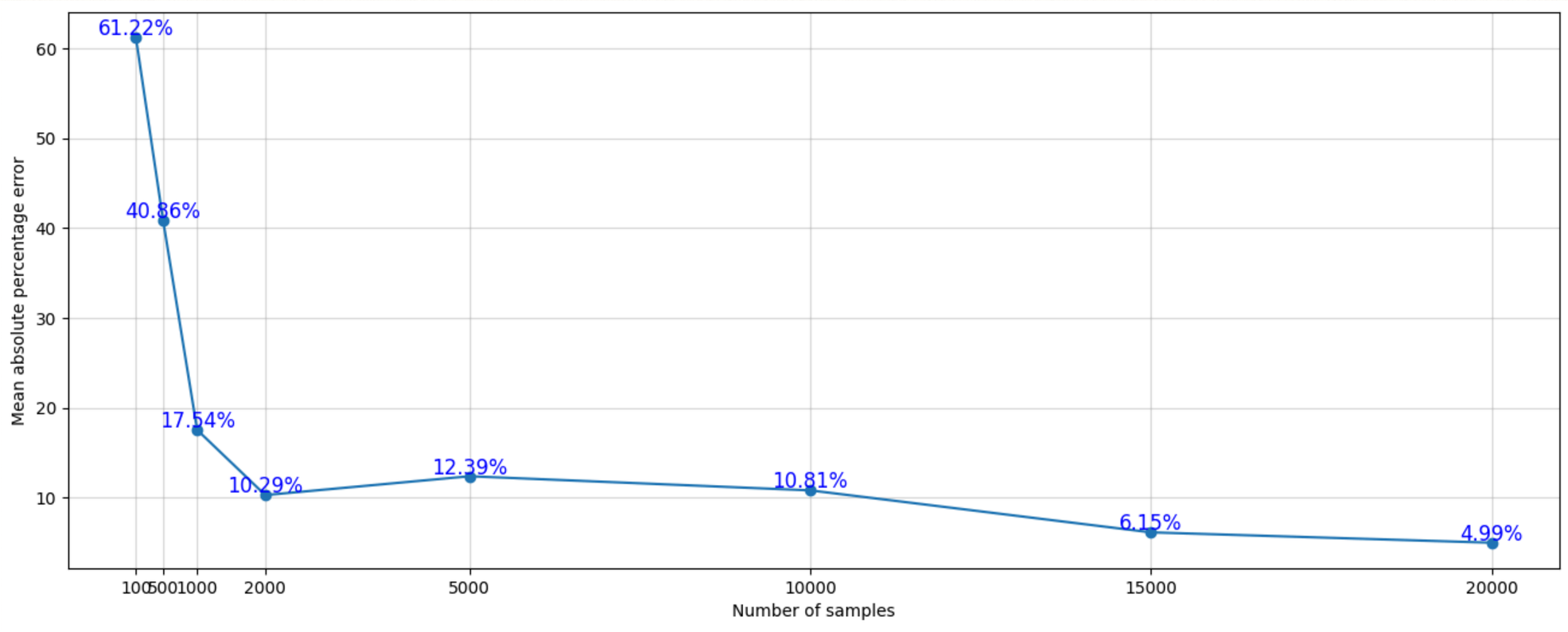
for num_samples:
- take a random permutation of the players
for i, player in enumerate(perm):
- sum up the marginal contribution for each player
marginal_contrib = v[perm[:i].union(player)] - v[perm[:i]]
shapley_dict[player] += marginal_contrib

divide the shapley values of each player with num_samples (compute the average of the marginal contributions)
for player in shapley_dict.keys():
shapley_dict[player] /= num_samples

return shapley_dict

Monte Carlo sampling

- We used **MAPE** as function to calculate error between classic Shapley value computation and Monte Carlo sampling approximation
- The more samples we take, the more accurate the approximation will be
- Analysis made on the same 20 players instance



Core

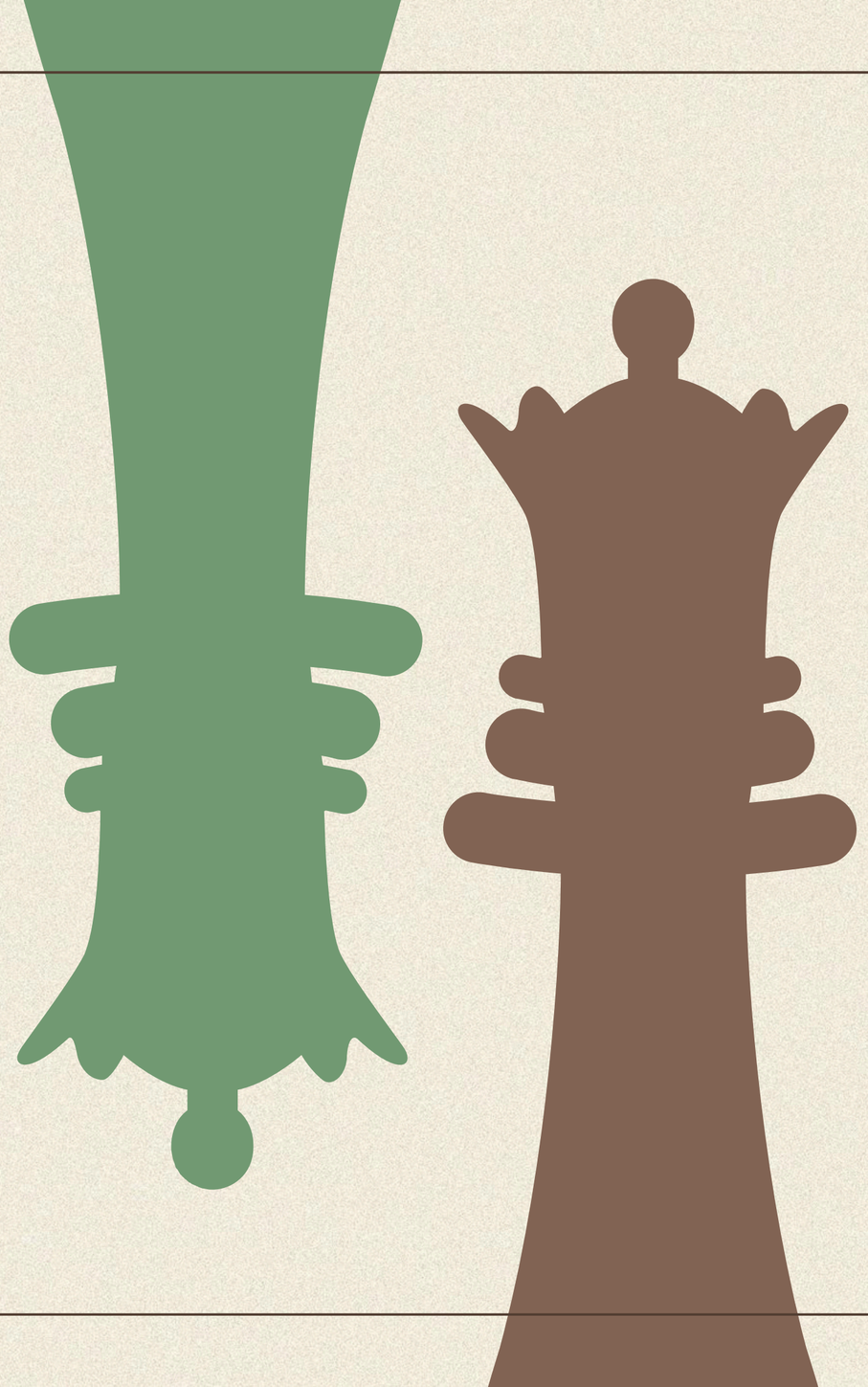
- The **core** of a game is the set of all **stable outcomes**
- For every coalition S ⊆ N check the following properties:
  - □ **positivity**: the payoff is positive for each agent;
  - □ **efficiency**: all value is allocated;
  - □ **stability**: no coalition wants to deviate.

def is_in_the_core(outcome, v, players):
return is_positive() and is_efficient() and is_stable()

Results

- Shapley value for the 4 players instance:
  { '1': 0.0, '2': 16.67, '3': 16.67, '4': 66.67}
- Agent 1 is **dummy**
- Agent 2, 3 are **symmetric**
- Agent 4 is **pivotal**

- The Shapley values are positive, efficient, but not stable. Example:
  for coalition {3, 4} we have: $\phi(3) + \phi(4) = 83.33 < v(\{3, 4\}) = 100$
- The Shapley value **is not in the core** of the game
- So the players want to deviate from the Shapley value distribution



Point 2

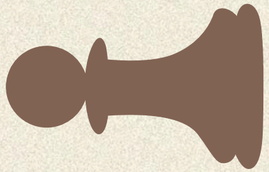
The new setting

- Each agent freely decide to join or not.
- There is a fixed cost that every agent has to join the group and everyone would like to cover the expense.
- Revenue divided according to the Shapley value of the coalitional game induced by the agents that join the group
- Every agent is selfish interested = Non-Cooperative Game

The task

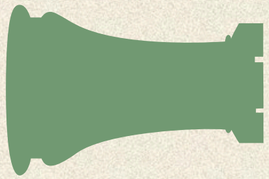
Check whether the resulting strategic setting, where each agent has two actions (join, not join), admits a **pure Nash equilibrium** and computes one, if any.

The Nash equilibrium for the setting



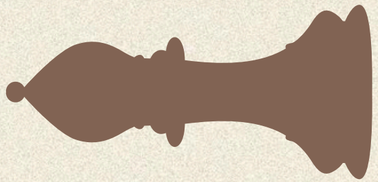
Best choice for every agent

Every agent decides to join or be external to the group with the respects of their incomes and their costs



Players in the group stay

Task completed, everyone’s costs are covered, so they don’t want to leave



Players outside the group don’t join

If each agent decides to join **independently** of the decisions of the other external agents, the task will not be completed or their cost is greater than their Shapley revenue, so they will have a negative revenue

EXAMPLE: COALITION [2,4]

Fixed costs = [10,10,20,40]

Player situation at the beginning:

- 1 is not participating
- 2 is participating
- 3 is not participating
- 4 is participating

Players' decisions:

- Player 1: 0 - 10 < 0 → NO
- Player 2: 16.67 - 10 > 0 → YES
- Player 3: 16.67 - 20 < 0 → NO
- Player 4: 66.67 - 40 > 0 → YES

Player situation at the end:

- 1 stay outside
- 2 stay in the group
- 3 stay outside
- 4 stay in the group

→ Nash equilibrium found!

Pseudo-code

def compute_nash_equilibrium_coalitions(characteristic_function, shapley_values, agents_costs):

```
    good_coalitions = []
    for coalition in powerset(agents):
        if verify_nash_equilibrium(coalition, characteristic_function, shapley_values, agents_costs):
            good_coalitions.append(coalition)
    return good_coalitions
```

Results: (2,4) and empty set ()

Point 3

Task

Assume that all agents participate to the setting, but they might cheat on the cost, 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.

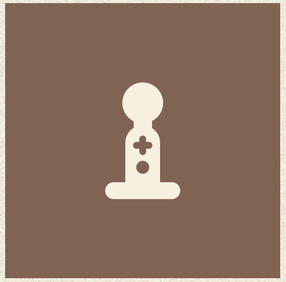
VCG Auction Setting

Bidders

The agents of the exercise

Item bundle

The skills set each agent has



Bidders' valuation on item bundle

The cost reported by each agent

Definitions

List of triples of the form:
[(”bidder”, ”item bundle”, ”bidder′s valuation on item bundle”), ...]

Allocation

Admissible Allocation

The union set of all the item bundles in each triple is equal to the set S of all the skills

Main Functions

compute_winner

input: the bid of each bidder
output: the allocation with the minimum overall cost (*optimal allocation*)

VCG_auction

input: the bid of each bidder
output: VCG payment for each bidder of the optimal allocation

Auxiliary Function

generate_only_admissible

Implements **backtracking**
to generate all the
admissible allocation from
a given set of bids.
It is internally used by
compute_winners

VCG Payment Formula

Once we find the optimal allocation (the one with the minimum overall cost), for each bidder “i” in it we compute the VCG payment as follows:

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

Where the optimal allocation ω* is defined as follows:

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

Pivotal Agent Problem

Pivotal Agent

Player whose participation to the coalition determines the coalition value to be nonzero

Problem

How to compute the first term of the VCG payment for a pivotal agent?

Solution

If the player “i” is a pivotal agent, then we enforce that $p_i = c_i$

VCG_auction pseudo-code pt.1

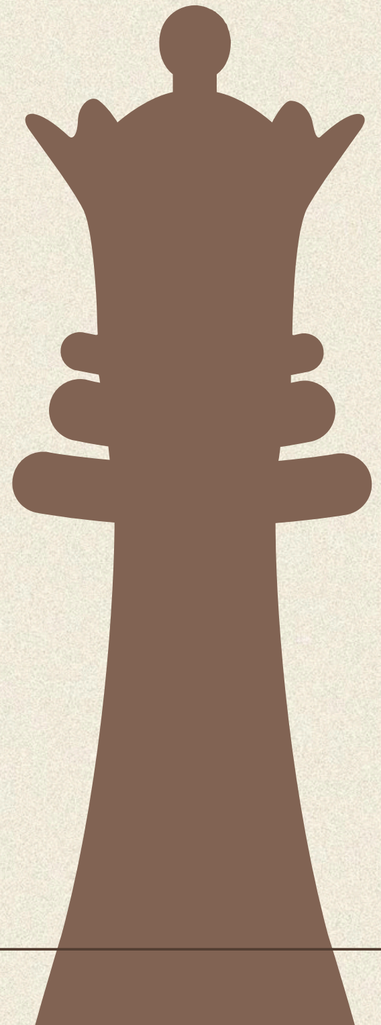
```
function VCG_auction(list_of_triples) {
  ret = new Array()
  total_payment = 0
  optimal_allocation, total_value = compute_winner(list_of_triples)

  for each (bidder, itemset, value) in optimal_allocation {
    payment = 0
    if (is_pivotal(bidder)) {
      payment = value
    }
    else {
      list_without_current = copy(list_of_triples).remove(bidder)
      ...
```

VCG_auction pseudo-code pt.2

```
      ...
      sum_values_without_current = compute_winner(list_without_current)
      sum_values_other_bidders = total_value - value
      payment = sum_values_without_current - sum_values_other_bidders
    } // else
    total_payment += payment
    ret.add( (bidder, itemset, payment) )
  } // for
  if (total_payment > reward) {
    print("Total payment exceeds the budget.")
    system.exit()
  }
  return ret
} // VCG_auction
```

Results

Bidder	Offer	Belongs to optimal allocation	Payment
1	[s1], 10	no	0
2	[s1, s2], 10	yes	20
3	[s1, s2], 20	no	0
4	[s3], 40	yes	40



THANKS

DO YOU HAVE ANY QUESTIONS?