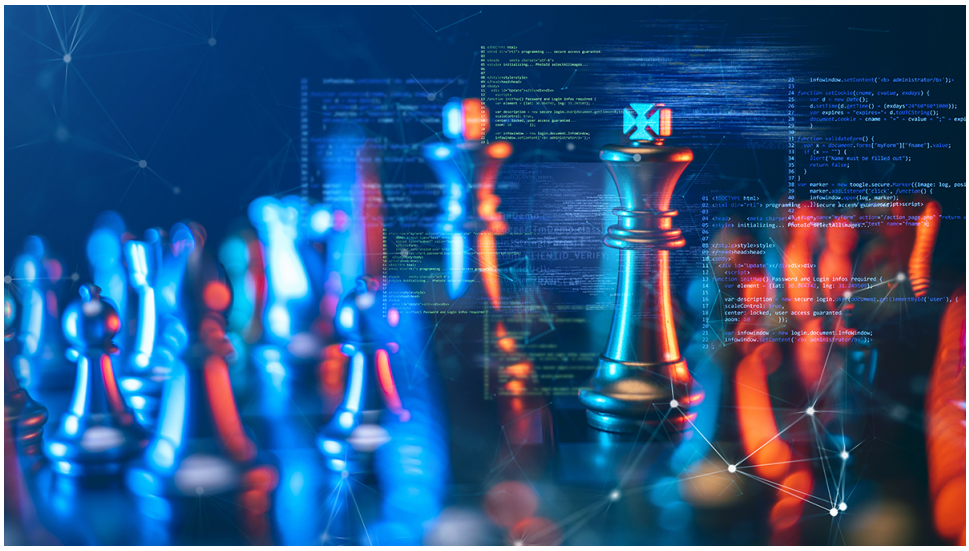
UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA E INFORMATICA**



Group 4 Project

Professors:

Prof.Gianlugi Greco

Prof.Sebastiano Piccolo

Students:

Pierpaolo Sestito 242707

Cristian Ciriaco Campagna 242604

Academic Year 2023/2024 - Algorithmic Game Theory

Contents

Chapter 1

Introduction

1.1 Assignment Overview

Consider a setting with an undirected graph G=(N,E), where N= 1,2,. . . ,n} U {s,t} is the set of nodes and s and t are two distinguished source and target nodes, respectively. The objective is to define a path connecting s and t, via the nodes in 1,2,. . . ,n with each of them being controlled by an agent. Hereinafter, such nodes are therefore transparently viewed as the corresponding agents. For each of the following questions, implement in Python a method that can provide results for any possible graph G. Report then the results obtained over the specific graph instance depicted below.

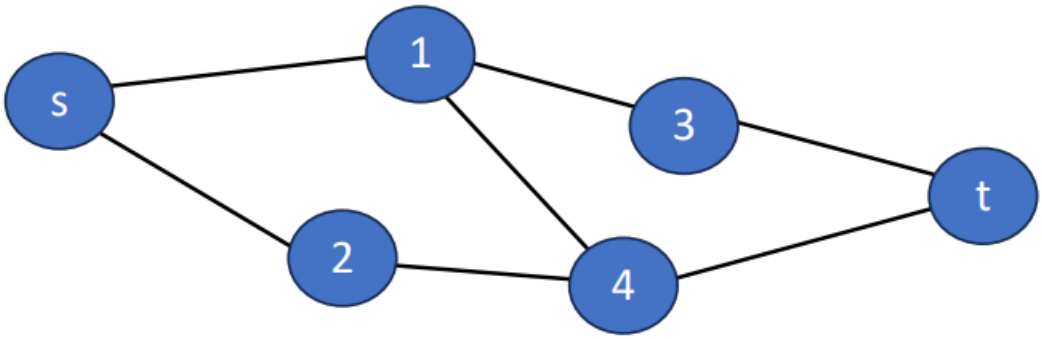


Figure 1.1: Assignment Graph

1.1.1 Objectives

1. Assume that forming a path connecting s and t leads to a reward of **100$**. Then, compute the **Shapley value** associated with the agents in {1,2,...,n} as a fair way to distribute that reward among the agents in {1,2,...,n}, which in particular encourages cooperation.

2. Assume that each agent in {1,2...,n} might freely decide whether to provide her/his contribution to connect **s** and **t**. Assume, in particular, that each agent is willing to contribute only if at most two of her/his neighbors do so. Then, check whether the resulting setting admits a pure **Nash equilibrium** and compute one, if any.

3. Assume that G has treewidth bounded by some constant and provide again answer to point 2, by exploiting this additional information.

4. Assume that agent i in {1,2,...,n} has some internal utility – say **i x 10$** – for being selected in a path connecting **s** and **t**, and that s/he might cheat in declaring a different utility. Assume moreover that the goal is to form a path with the maximum overall possible utility, and compute a payment scheme that provides incentives to truthfully report such utility values.

Chapter 2

Solutions

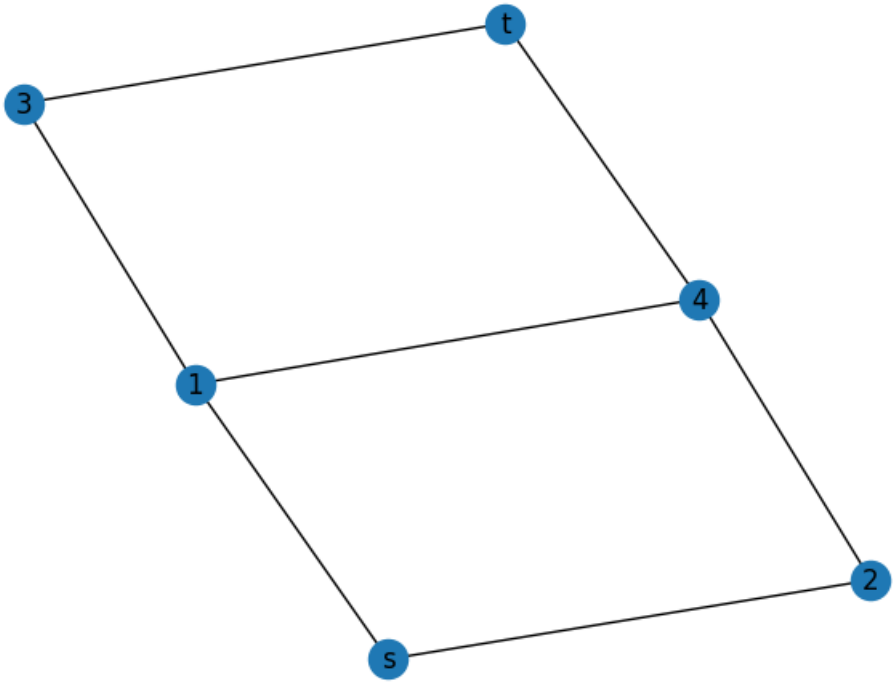


Figure 2.1: Graph implemented using *networkx* library

```
import networkx as nx

graph = nx.Graph()
start_end = ('s','t')
graph_nodes = [1,2,3,4]
graph_edges = [(start_end[0],1),(start_end[0],2),(1,3),(1,4)
    ,(2,4),(3,start_end[1]),(4,start_end[1])]

graph.add_nodes_from([start_end[0],start_end[1]] + graph_nodes)
graph.add_edges_from(graph_edges)

nx.draw(graph, with_labels = True)
```

2.1 Shapley Value

Problem

Assume that forming a path connecting s and t leads to a reward of \$100. Then, compute the Shapley value associated with the agents in $\{1, 2, ..., n\}$ as a fair way to distribute that reward among the agents in $\{1, 2, ..., n\}$, which in particular encourages cooperation.

Problem Solution

One method for equitable reward distribution involves computing the Shapley value for each agent, determined by the following formula:

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

Here, $\phi(i, v)$ represents the Shapley value for agent i, where v denotes the characteristic function.

The characteristic function operates as follows: if the agents in a coalition successfully establish a path from s to t, they share a reward of \$100; otherwise, the reward is \$0.

```
def exam_characteristic_function(graph: nx.Graph, coalition:
    frozenset):
     return 100 if exists_path_between_nodes(graph, coalition)
    else 0
```

To ascertain whether a coalition C forms such a path, we devised a function that examines whether a path exists from s to t within the subgraph induced by $C \cup \{s, t\}$.

We devised a method to compute the characteristic function utilizing a bottom-up approach. Beginning with the empty coalition, we iteratively determined its value and then recursively calculated the worth of its supersets by gradually adding agents. Notably, we optimized the computation by bypassing the evaluation of supersets of coalitions already worth \$100. This optimization capitalizes on the inherent property that if a path is formed within a coalition, it will persist across all its supersets. Consequently, the worth of any superset containing a path from s to t remains \$100, eliminating the necessity for redundant path evaluations.

```
def get_characteristic_function(graph, graph_nodes: frozenset,
    coalition, check_path, characteristic_function):

     if coalition in characteristic_function:
         return
     characteristic_function[coalition] =
    exam_characteristic_function(graph, coalition) if check_path
     else 100
     check_path = False if characteristic_function[coalition] ==
     100 else True

     graph_nodes_without_coalition = graph_nodes.difference(
    coalition)
     supersets = [coalition.union([j]) for j in
    graph_nodes_without_coalition]

     for c in supersets:
         get_characteristic_function(graph, graph_nodes, c,
    check_path, characteristic_function)
```

At the end, the Shapley Value is calculated as follows (using the equation described before):

```
def shapley_value(player, characteristic_function):
    player_list = max(characteristic_function)
    player = set([player])
    N = len(player_list)
    coalitions = powerset(player_list)
    shapley_val = 0
    for coalition in coalitions:
        if len(coalition) != 0:
            S = len(coalition)
            marginal_contribution = characteristic_function[
  coalition] - (characteristic_function[coalition - player] if
   len(coalition - player) > 0 else 0)
            shapley_val += ((factorial(N - S) * factorial(S -
  1)) / factorial(N)) * marginal_contribution
    return shapley_val
```

Output:

Player	Shapley Value (%)
1	33.33
2	16.67
3	16.67
4	33.33

Table 2.1: Shapley Values

2.2 Nash Equilibrium

Problem

Assume that each agent in $\{1, 2 \ldots, n\}$ might freely decide whether to provide her/his contribution to connect s and t. Assume, in particular, that each agent is willing to contribute only if at most two of her/his neighbours do so. Then, check whether the resulting setting admits a pure Nash equilibrium and compute one, if any.

Problem Solution

In this scenario, each node, excluding s and t, is regarded as an agent faced with a binary decision: either contribute or abstain. Regardless of the node, every agent adheres to the same utility function, which, contingent upon the actions of its neighbors, influences their preference towards one action over the other. Nodes s and t are exempted from the agent classification and consistently opt to contribute. Enclosed below is a straightforward implementation that exhaustively evaluates all potential action combinations to ascertain a pure Nash equilibrium

```
def check_nash_equilibrium(graph, players, action_profile: dict
    ):
     for player in players:
         sum_ns = 0
         for neighbor in graph.neighbors(player):
               if action_profile[neighbor] == "y":
                 sum_ns += 1
         if action_profile[player] == 'n' and sum_ns <= 2:
           return False
         if action_profile[player] == 'y' and sum_ns > 2:
           return False

     return True

def find_nash_equilibrium(graph, players: list, action_profile:
     dict, i=0):
     for c in range(2):
         c2 = ""
         c2 = "y" if c == 1 else "n"

         action_profile[players[i]] = c2
         if i == len(players) - 1:
             if check_nash_equilibrium(graph, players,
    action_profile):
                 return action_profile
         elif normal_find_nash_equilibrium(graph, players,
    action_profile, i + 1) is not None:
             return action_profile
     return None
```

Observation

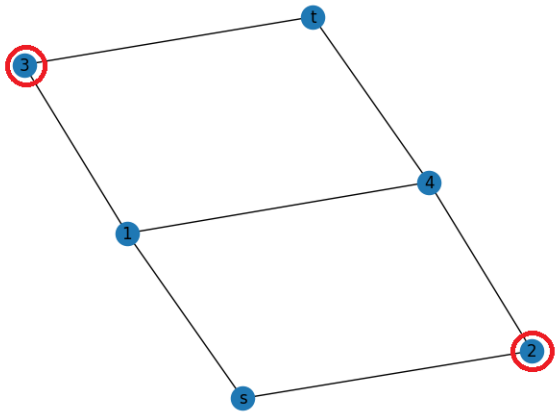


Figure 2.2: Node with degree ≤ 2, excluding s and t

In order to decrease the execution time of our function, we understood that

$$\forall v \in V \quad | \quad \deg(v) \leq 2$$

always contributes. So, we decided to implement a function that allows us to construct the initial action profile dynamically, instead of starting with 's' and 't' that contributes and determine the states of all the others. The action profile is built setting 'y' to all the nodes that has degree ≤ 2 and subsequently we find the Nash equilibrium.

The *build_action_profile* function is implemented as follows:

```
def build_action_profile(graph, players, add):
    action_profile = {player: 'y' for player in players if
   graph.degree(player) <= 2}
    if add:
        action_profile.update({start_end[0]: 'y', start_end[1]:
    'y'})
    new_players = [p for p in players if p not in
   action_profile]
    return action_profile, new_players if new_players else
   players.copy()
```

Proof that optimized version is better than before

We can notice that the optimized version takes less time than the first version.

Here a table which contains the time comparison and a scatter plot.

Solution type	Time (s)
Normal	0.00018787384033203125
Optimized	0.00012755393981933594

Table 2.2: Execution times for different solution types

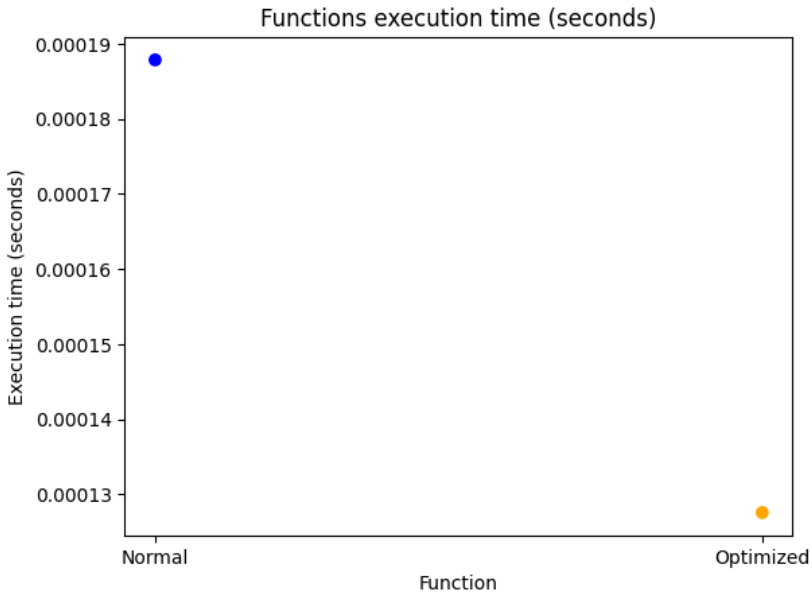


Table 2.3: Graph implemented using *networkx* library

Output:

Player	N.E Found
1	Does not contribute
2	contributes
3	contributes
4	contributes

Table 2.4: Nash Equilibrium for each player

Proof that optimized function works fast on different graphs

```
for i in range(5,16):
    random_graph = nx.fast_gnp_random_graph(randint(10,20),
   uniform(0.3,0.8))
```

This snippet of code allows us to create 10 different random graphs, which the first has 5 nodes and the last one 15. We start to consider the time to find Nahs Equilibrium with normal and optimized function over these graphs.

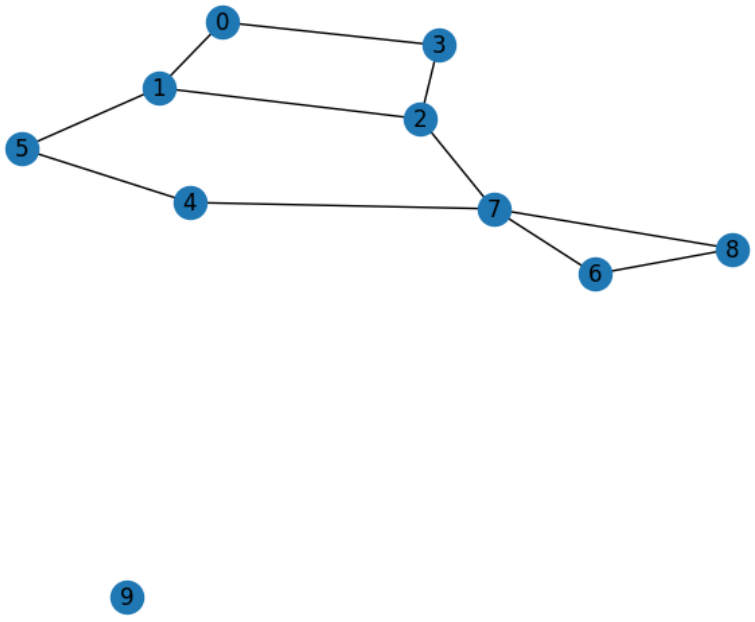


Figure 2.3: In this figure one of the graph included in the bunch of examples

And the computed Nash Equilibrium of 2.3:

Player	N.E Found
0	contributes
1	Does not contribute
2	contributes
3	contributes
4	contributes
5	contributes
6	contributes
7	Does not contribute
8	contributes
9	contributes

Table 2.5: Nash Equilibrium for each player

At the end we got this result about time:

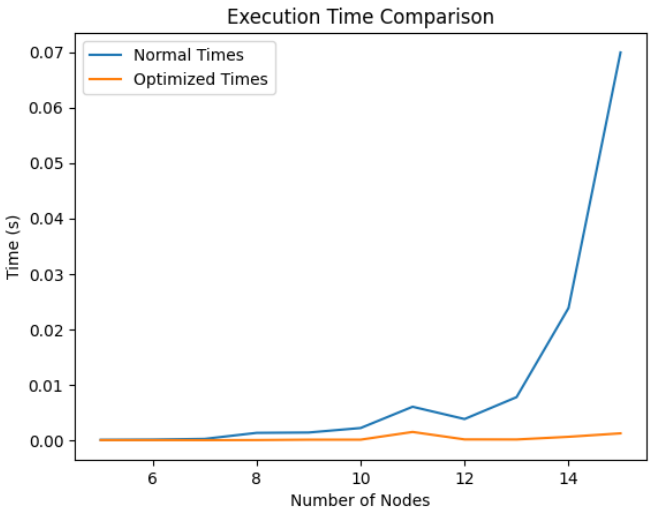


Figure 2.4: One of the graph included in the bunch of examples

2.3 Nash Equilibrium with bounded treewidth

Problem

Assume that G has treewidth bounded by some constant and provide again answer to point 2, by exploiting this additional information.

Problem Solution Taking into account the fact that the graph has bounded treewidth, we can now leverage Courcelle's theorem, which asserts that any problem on graphs expressible in Monadic Second Order Logic (MSO) can be solved in linear time on graphs with bounded treewidth.

By translating point (2.2) into MSO we can demonstrate that there is a linear algorithm that can solve this problem on the graphs on the delimited trees

Sets Definitions

$N : \{v \in N \rightarrow \exists\ x,y,z \in \Gamma(\mathrm{v}) \land (x \neq y \neq z) \land (x \in Y \land y \in Y \land z \in Y)\}$

Y: $\{v \in Y \rightarrow (((\exists\ x \in \Gamma(\mathrm{v}) \land x \in Y) \lor (\exists\ x,y \in \Gamma(\mathrm{v}) \land x \neq y \land (x \in Y \land y \in Y))) \land (\nexists\ x,y,z \in \Gamma(\mathrm{v}) \land (x \neq y \neq z) \land (x \in Y \land y \in Y \land z \in Y)))\}$

- N set of nodes that doesn't contribute to the equilibrium.
- Y set of nodes that contributes to the equilibrium.
- Γ(v) Is the neighbourhood of v

Conditions

$$
\begin{aligned}
&(\exists\ \mathrm{N,Y}) \quad [\ \Big(\ \forall\ \mathrm{v}\ (\mathrm{N(v)}\ \lor\ \mathrm{Y(v)}) \\
&\land\ (\forall\ \mathrm{v}\ (\mathrm{N(v)}\ \Rightarrow\ (\neg\ \mathrm{Y(v)}))) \\
&\land\ (\forall\ \mathrm{v}\ (\mathrm{Y(v)}\ \Rightarrow\ (\neg\ \mathrm{N(v)}))) \\
&\land\ (\forall\ \mathrm{v}\ (\ \mathrm{N(v)} \Rightarrow \exists\ \mathrm{x,y,z} \in \Gamma(v) \land\ (\mathrm{Y(x)} \land \mathrm{Y(y)} \land \mathrm{Y(z)}))) \\
&\land \Big(\forall\ \mathrm{v}\ (\mathrm{Y(v)} \Rightarrow (\exists\ \mathrm{x} \in \Gamma(v) \land (\mathrm{Y(x)}) \\
&\lor (\exists\ \mathrm{x,y} \in \Gamma(v) \land (\mathrm{Y(x)} \land \mathrm{Y(y)}))) \\
&\land (\nexists\ \mathrm{x,y,z} \in \Gamma(v) \land (\mathrm{Y(x)} \land \mathrm{Y(y)} \land \mathrm{Y(z)}))\Big)\ \Big)\ ]
\end{aligned}
$$

2.4 VCG

Problem

Assume that agent i in $\{1, 2, ..., n\}$ has some internal utility – say $i * 10\$$ – for being selected in a path connecting s and t, and that s/he might cheat in declaring a different utility. Assume moreover that the goal is to form a path with the maximum overall possible utility, and compute a payment scheme that provides incentives to truthfully report such utility values.

Problem Solution

In this game scenario, an outcome o represents a group of agents for which there exists a path from s to t that encompasses all of them.

The Vickrey-Clarke-Groves (VCG) mechanism serves as the mechanism for devising a payment scheme aimed at encouraging agents to truthfully disclose their actual utility for being chosen in a path. This mechanism identifies the optimal outcome o^* by maximizing $\sum_i \hat{v}_i(o, \theta_i)$. Subsequently, once o^* is determined, the payment requested from each agent equals $h_i - \sum_{j \neq i} \hat{v}_j(o^*, \theta_j)$, where h_i denotes the value of the optimal outcome in the game excluding agent i.

Such a payment scheme fosters truthfulness, as it precludes any potential gain for an agent by misreporting their concealed type. The following function calculates the payment for each agent based on the declared types. For illustrative purposes, the utility function is implemented as follow (To avoid agent misreporting or declare false type).

```
def utilities(nodes,who_cheats):
    uti={}
    for node in nodes:
      if node == who_cheats:
        uti[node] = 100
      else:
        uti[node] = node*10
    return uti
```

Explanation

In the context of the solution proposed for the problem using the VCG (Vickrey-Clarke-Groves) mechanism, $\hat{v_i}(o, \theta_i)$ represents the virtual value that agent i attributes to outcome o given their declared type θ_i. This virtual value can be interpreted as the marginal contribution of agent i to the overall outcome o.

The sum $\sum_i \hat{v_i}(o, \theta_i)$ represents the sum of the virtual values attributed by all agents to outcome o, each based on their declared type.

The term $h_i - \sum_{j \neq i} \hat{v_j}(o \cdot \theta_j)$ represents the difference between the value of the optimal outcome h_i excluding agent i, and the sum of the virtual values attributed by the other agents (excluding i) to the optimal outcome o. This calculation determines how much agent i contributes to the optimal outcome when excluding their own contribution.

In summary, these terms are used in the VCG mechanism to compute the payments that agents must make, and they are essential for incentivizing truthfulness in the declaration of utility values. Maximizing the sum of virtual values identifies the optimal outcome, while the difference between the value of the optimal outcome excluding the agent and the sum of virtual values of the other agents measures the agent's contribution to the optimal outcome.

And the rest of the implementation to estimate payments:

```
def find_best_result_excluding_player(paths, excluded_player,
    uti):
     best_result = 0
     for path in paths:
         if excluded_player not in path:
             result=0
             for node in path:
               if node not in start_end:
                 result+=uti[node]
             best_result = max(result, best_result)
     return best_result

def estimate_payment(best_result, best_path, player,uti):
     result_player = best_result - uti[player] if player in
    best_path else best_result
     result_excluding_player = find_best_result_excluding_player
    (paths, player,uti)
     return result_excluding_player - result_player
```

best_result is determined by a function that allows us to find the path with the best result considering all nodes.

We provide two scenario to proof that our VCG mechanism works and take care about truthfulness about player's declared utility.

- In the first scenario we assume that neither of our agent try to the declared false and cheats over the mechanism.

  1st Scenario Output:

Player	**Payment**
1	-30
2	-30
3	-10
4	-20

Table 2.6: Payments without cheating

- In the second one, for illustration purposes, each agent in different round will try to cheat, declaring a fake utility of 100.

  2nd Scenario Output:

Cheater	**Player 1**	**Player 2**	**Player 3**	**Player 4**
1	**-30**	-30	-20	-20
2	-30	**-30**	-20	-100
3	-100	-40	**-10**	-20
4	-30	-30	-10	**-20**

Table 2.7: Payments within cheating

– **Observation:**

We can see that, even if a player tries to lie, he will not improve his situation and will remain as he is, on the contrary, he will probably benefit some other player.