UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA E INFORMATICA**

Project

Professors:

Prof.Gianlugi Greco

Dott.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.

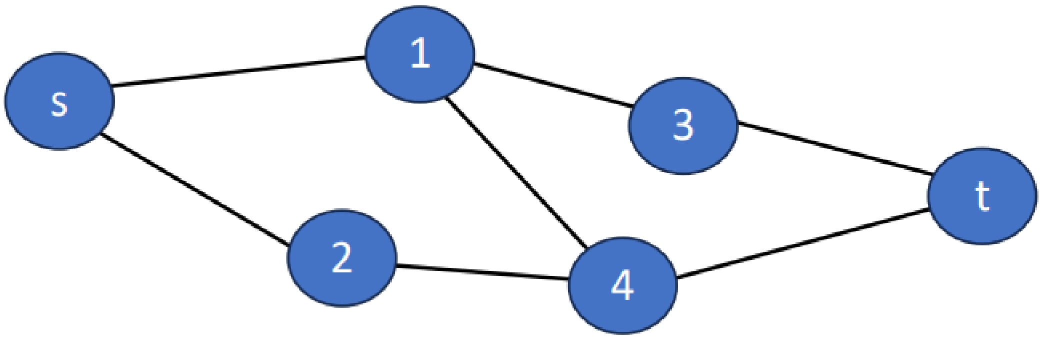

Figure 1.1: Assignment Graph

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

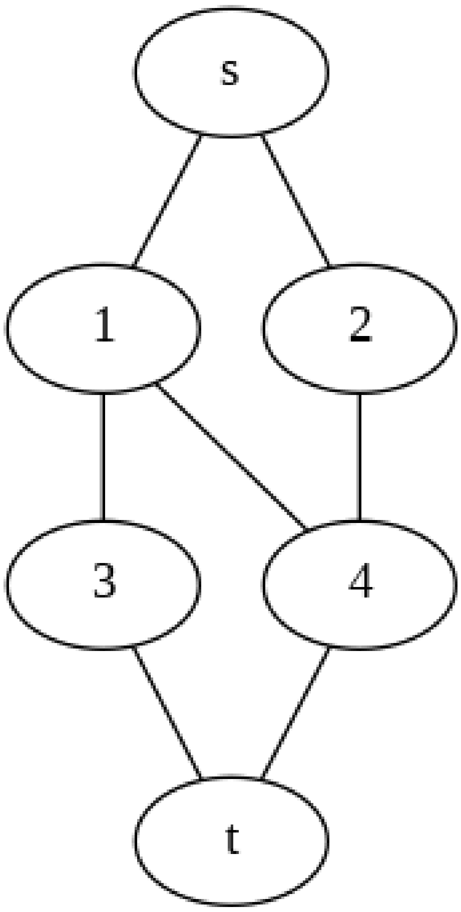

Figure 2.1: Graph implemented using *pydotplus* library

```python
import pydotplus
from IPython.display import Image, display
start_end = ('s','t')
graph = pydotplus.Dot(graph_type='graph')
nodes = [start_end[0], start_end[1]] + [str(i) for i in range
    (1, 5)]
for node in nodes:
    graph.add_node(pydotplus.Node(node))
edges = [(start_end[0], 1), (start_end[0], 2), (1, 3), (1, 4),
    (2, 4), (3, start_end[1]), (4, start_end[1])]
for edge in edges:
    graph.add_edge(pydotplus.Edge(str(edge[0]), str(edge[1])))
display(Image(graph.create_png()))
```

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: pydotplus.Dot,
    coalition: frozenset):
     return 100 if exists_path_between_s_and_t(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

def shapley(characteristic_function):
    return {player: shapley_value(player,
   characteristic_function) for player in max(
   characteristic_function)}
```

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 $\{1, 2 . . . , n\}$ 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 find_neighbors(graph, str(player)):
            if neighbor.isdigit():
                if action_profile[int(neighbor)] == "y":
                  sum_ns+=1
            else:
              if action_profile[neighbor] == "y":
                sum_ns += 1
        if action_profile[player] == 'y' and sum_ns > 2:
          return False
        if action_profile[player] == 'n' 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 find_nash_equilibrium(graph, players,
    action_profile, i + 1) is not None:
            return action_profile
    return None
```

Output:

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

Table 2.2: {'s': 'y', 't': 'y', 1: 'n', 2: 'y', 3: 'y', 4: 'y'}

2.3 Graph treewidth bounded

Problem

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

Problem Solution

With the newfound understanding that the graph possesses bounded treewidth, we can leverage Courcelle's theorem, which posits that any problem formulated within Monadic Second Order Logic (MSO) on graphs of bounded treewidth can be resolved in linear time.
By capitalizing on this theorem, we can articulate the second point (2.2 Nash Equilibrium) in Monadic Second Order Logic to establish the existence of an algorithm capable of resolving the problem within linear time complexity.

MSO Formulation for Nash Equilibrium Verification

Step 1: Definition of Node Sets

Let V_0 and V_1 be two sets of nodes defined as follows:

$$V_0 = \{v \in N \mid (deg(v) \leq 2) \wedge (\sigma(v) = 0)\}$$

$$V_1 = \{v \in N \mid (deg(v) > 2) \wedge (\sigma(v) = 1)\}$$

Step 2: Formulation of Conditions in MSO

We use second-order quantification to express the conditions in MSO. The MSO formula is as follows:

Condition 1: No agent changes action if the number of contributing neighbors is greater than 2

$$\forall v \in A \Big[(v \in V_0 \land \forall u \in N(v)\, (u \notin V_1)) \\ \lor\ (v \in V_1 \land \forall u \in N(v)\, (u \in V_1)) \Big]$$

Condition 2: All agents follow this condition

$$\forall v \in A\, \forall u \in N(v) \Big[(u \in V_0 \land \forall w \in N(u)\, (w \notin V_1)) \\ \lor\ (u \in V_1 \land \forall w \in N(u)\, (w \in V_1)) \Big]$$

These formulations express the necessary conditions for an action profile to be a Nash equilibrium in the graph using second-order quantification in monadic second-order logic (MSO), with a clear definition of node sets and the use of second-order quantification for the conditions.

Here's a breakdown of how the formulas adhere to MSO principles:

1. **Node Sets**
   - (a) The formulas define two sets of nodes, V0 and V1, using first-order logic over the set of nodes N.
   - (b) This aligns with MSO, which allows quantification over sets of elements in the domain.
   - (c) **Second-Order Qualification**
     - i. The conditions for Nash equilibrium are expressed using second-order quantification over sets of nodes.
     - ii. This is a key feature of MSO, enabling statements about relationships between sets of elements.
     - iii. **Restriction to Monadic Qualification**
       - A. The quantification is restricted to sets of nodes, not relations or functions.
       - B. This adheres to the monadic nature of MSO, which only permits quantification over sets.
       - C. **Expressing Graph Properties**
       - D. The formulas capture properties of the graph, such as node degrees and relationships between neighboring nodes.
       - E. MSO is well-suited for expressing graph properties due to its ability to quantify over sets of nodes and edges.
       - F. **Capturing Nash Equilibrium Conditions**
       - G. The formulas successfully express the necessary conditions for an action profile to be a Nash equilibrium in the graph.
       - H. This demonstrates the expressive power of MSO in formulating strategic concepts in game theory.

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 utility(node,cost):
    if cost==10:
      return node * cost
    return cost
```

And the rest of the implementation to estimate payments:

```
def find_best_result(paths,cost):
    best_result = 0
    best_path = []
    for path in paths:
        result = sum(utility(int(node),cost) for node in path
   if node not in start_end)
        if result > best_result:
            best_result = result
            best_path = path
    return best_result, best_path

def find_best_result_excluding_player(paths, excluded_player,
   cost):
    best_result = 0
    for path in paths:
        if str(excluded_player) not in path:
            result = sum(utility(int(node),cost) for node in
   path if node not in start_end)
            best_result = max(result, best_result)
    return best_result

def estimate_payment(best_result, best_path, player,cost):
    result_player = best_result - utility(player,cost) if str(
   player) in best_path else best_result
    result_excluding_player = find_best_result_excluding_player
   (paths, player,cost)
    return result_excluding_player - result_player
```

Output:

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

Table 2.3: Payments