Heuristic Search trial in Bayesian games
Supervised by Jilles S. Dibangoye - CITI Lab

Conducted by Thomas DAMBRIN - INSA Lyon

December 2021

This document aims at providing a synthetic overview of the advances made in the research project on game theory supervised by Jilles S. Dibangoye. This research project is conducted as part of the Telecommunication Engineering curriculum at INSA Lyon. The student, Thomas Dambrin, should first become familiar with the game theory state-of-the-art on specific problem subclasses. This involves studying theoretical concepts, establishing the corresponding linear programming problems for small examples and the adequate solving method on SDM'S, the Plasma team C++ library providing solvers for sequential decision making problems. All these steps will be performed on the following classes :

- Zero-sum Normal-form games
- Zero-sum Bayesian games [1]

Therefore, the first part of this document will be dedicated to solving approaches for these well-known classes of game theory problems.

Then, we will investigate a new approach to solve Zero-Sum Bayesian games and discuss its results obtained with the SDM'S implementation. We will study different heuristic to fasten solution searches. Research on these classes have shown interest towards finding approximated solutions through constrains that take a cut on computational time [2].

Contents

1 Zero-Sum Normal-Form Games

1.1 Mathematical study & Linear Programming Problem

The purpose of this section is to illustrate how to solve zero-sum normal-form games, both theoretically and with SDM'S experimentally, implementing the solving algorithms in SDMS. Normal-form games are probably the simplest games in game theory as they can be defined by just the following :

- a finite set of players $\mathcal{I}$
- $\{\mathcal{A}_i\}_{i\in\mathcal{I}}$ finite sets of actions for each players
- $\{u_i : \times_{i\in\mathcal{I}}\mathcal{A}_i \rightarrow \mathbb{R}\}$ real-valued utility functions giving the utility for player i depending on every player's actions

Such a game can be represented by a matrix whose rows and columns correspond to players' actions and squares to the game payoffs for each player.

To make it clear, let's say that two basketball players are facing each other, one wants to score the ball and the other has to prevent this. Player 1, i.e. the one who wants to score and has the ball, has two options : Shoot (S) or Feint (F). Player 2, i.e. the defender, reacts to that instantaneously with either going for the Counter (C) or Ignore (I) this threat. Figure 1 is this game's payoff matrix with each square featuring *player 1 payoff, player 2 payoff.*

	C	I
S	-3, 3	1, -1
F	3, -3	-1, 1

Figure 1: Basketball 1vs1

To solve normal-form games, we need to find a Nash equilibrium i.e. a set of unique strategies for each player where none of them would want to deviate from his strategy if he knew the other players' ones. In a normal-form game, a player strategy can be either pure (the player always plays the same action) or mixed (players plays according to a fixed distribution over its actions). Let's first search for pure strategies that form a Nash Equilibrium to the game. To do that, we highlight the best responses of each player to the other player strategies. Without going into more details, just know that to find best responses for a player, we choose the player's action that results into the highest utility when fixing the other players' actions. Indeed, if best responses match (i.e. always in the same square), this would mean that players' best strategies would constitute an equilibrium. As Figure 2 shows us, there aren't any matching best responses so there is no pure strategy equilibrium in this game.

However, there must be a mixed strategy equilibrium [1] so let's dive into this.

[1]**Nash Theorem, 1951** - Every game with a finite number of players and action profiles has at least one Nash equilibrium.

	C	I
S	-3, $\boxed{3}$	$\boxed{1}$, -1
F	$\boxed{3}$, -3	-1, $\boxed{1}$

Figure 2: Basketball 1vs1 best responses

Let's assume that player 1 shoots the ball with a probability of p and so that he feints with a probability of 1 - p. Then, in a mixed-strategy a Nash equilibrium, player 2 has to be indifferent about its two pure strategies, each pure strategy must yield into the same expected utility.

Hence, we can find the equilibrium by solving

$$
\begin{aligned}
U(C) &= U(I) && (1)\\
\Leftrightarrow 3p - 3(1-p) &= (1-p) - p && (2)\\
\Leftrightarrow 6p - 3 &= 1 - 2p && (3)\\
\Leftrightarrow 8p &= 4 && (4)\\
\Leftrightarrow p &= 1/2 && (5)
\end{aligned}
$$

We could do the same thing for player 1's strategy but in this case, it is easy to see that the result will be the same : perfect balance between both his actions. When the game is symmetric, a strategy for player 1 which is part of a Nash Equilibrium is also one for player 2. For this trivial game, there is one mixed strategy Nash equilibrium consisting of the two players playing either one of their pure strategy with probability 1/2.

This method works well for small examples but in a computer-driven method for solving it is not optimal. Indeed, the resolution of such a system runs un polynomial time complexity regarding the number of individual actions of the players. In practice, it may be possible to reach a lower complexity resolution method with solving a linear programming problem.
Let's do so with another simple example : a modified Rock, Paper, Scissors game. The matrix representation is given in Figure 3.

	R2	P2	S2
R	0, 0	-2, 2	1, -1
P	2, -2	0, 0	-4, 4
S	-1, 1	4, -4	0, 0

Figure 3: Modified Rock, Paper, Scissors

In finite, two-players zero-sum games, i.e. pure competition game, the solution for each player consists of finding a maxmin, or minmax, strategy. Indeed, such a strategy is a Nash equilibrium in this context. Let's write the problem for player 1 (row player). A maxmin strategy π_1^* is a pure or mixed strategy which fulfils the following condition :

$$\pi_1^* = \arg\max_{\pi_1} \min_{\pi_2} u_1(\pi_1, \pi_2) \tag{6}$$

However, we don't need to consider player 2's mixed strategies. Pure strategies can yield into lower payoff for player 2 at fixed π_1 given that the utility function for player 2 is linear relatively to mixed strategies who are distributions over the set of pure ones. Therefore, we must only focus on player 2's pure strategies. Let's say that player 1 plays rock with probability p_1, paper with probability p_2 and scissors with probability p_3. Then, we can expect the following payoffs (expected utilities for player 1 EU_1) for each of player 2's pure strategies :

$$R2 : EU_1 = 2 \cdot p_2 - p_3 \tag{7}$$

$$P2 : EU_1 = (-2) \cdot p_1 + 4 \cdot p_3 \tag{8}$$

$$S2 : EU_1 = p_1 - 4 \cdot p_2 \tag{9}$$

In general, the formula for EU_1 given player 2's pure strategy identified by j, the index of the corresponding column in the matrix, is $EU_1 = \sum_i a_{ij} \cdot p_i$ with i the index of the raw identifying a pure strategy for player 1. The first part of the minmax problem is to minimize this quantity so let's write $u = \min_j \sum_i a_{ij} \cdot p_i$. Note that each EU_1 defined by equations (7), (8) and (9) must be greater or equal to u.

Thus, player 1 strategy fulfils $\pi_1^* = \arg\max_{\pi_1, j} \min_j \sum_i a_{ij} \cdot p_i$

which can be transformed to a linear problem with the objective :

$$s_1^* = \arg\max_{\pi_1, u} u$$

and with the constraints :

$$\begin{aligned} 2 \cdot p_2 - p_3 &\geq u \\ (-2) \cdot p_1 + 4 \cdot p_3 &\geq u \\ p_1 - 4 \cdot p_2 &\geq u \\ \sum_i p_i &= 1 \\ \forall i, p_i &\geq 0 \end{aligned}$$

With an online solver (e.g. http://www.phpsimplex.com/) using the simplex method, we can now say that the optimal strategy for player 1 is to play with probabilities distribution (p1=0.571429, p2=0.142857, p3=0.285714) regardless of player 2 strategy. The same result would be obtained for player 2 as the game is symmetric.

1.2 SDM'S Implementation

Normal form games can be seen as a specific case of bayesian game (i.e. with each player only having one type), therefore, the SDM'S implementation will be discussed in the next section.

2 Zero-Sum Bayesian Games

This sections aims at aims at defining a zero-sum bayesian game, showing how to solve it with a linear programming approach and exposing the SDM's associated implementation.

2.1 Mathematical study and associated Linear programming

A Zero-sum bayesian game can be seen as a way to represent player's uncertainties about the very game being played [3]. To do so, we represent a bayesian game as a collection of normal-form games with the same number of agents and the same strategy space, i.e. the composing normal-form games only differ in their payoffs. In order to be as clear as possible, we introduce the concept of type for a given player. Intuitively, a type encapsulates all the information an agent derived from its observations. If an agent knows which type it is in, the space of possible games actually being played is reduced to games that correspond to the agent's type. We also add a probability distribution over the joint types, i.e. common prior over games being played. A visual representation can be the following :

	$\theta_{2,1}$		$\theta_{2,2}$	
$\theta_{1,1}$	1,-1	0,0	-1,1	-2,2
	-2,2	0,0	0,0	3,-3
	p = 0.1		p = 0.4	
$\theta_{1,2}$	0,0	1,-1	1,-1	-1,1
	-1,1	0,0	-1,1	1,-1
	p = 0.3		p = 0.2	

Figure 4: Two players Zero-sum Bayesian Game

Player 1 has two types : $\theta_{1,1}$ and $\theta_{1,2}$. Player 2 also has two : $\theta_{2,1}$ and $\theta_{2,2}$. The goal when solving a two players zero-sum bayesian game for agent i is to define, for each of its types, a strategy that constitutes a Nash-equilibrium. Thus, we must define the linear programming problem for each of the agent's types.
Let's first define some notations.

- Let $\pi_x(a_k|\theta_{x,i})$ be the probability that agent x plays action a_k when it is in the type $\theta_{x,i}$ and follows the strategy π_x

- Let $u_x(a_k, a_l, \theta_{i,j})$ be the agent x payoff when it plays a_k, when the agent y plays a_l and when the agent types are respectively $\theta_{1,i}$ and $\theta_{2,j}$.

- Let P($\theta_{i,j}$) be the joint probability P($\theta_{1,i}$, $\theta_{2,j}$).

The general form of this linear programming problem is a generalization of the normal form one. It corresponds to the following :

$$\begin{aligned}
\forall j \in |\text{types of agent } y|, \pi_x^* = \underset{\pi_x,\alpha_j}{\arg\max} \sum_j \alpha_j \\
\text{and with the constraints :} \\
\text{with } i \in |\text{types of agent } x|, \\
\text{with } j \in |\text{types of agent } y|, \\
\text{with } k \in |\text{actions of agent } x|, \\
\text{with } l \in |\text{actions of agent } y|, \\
\forall j, \forall l, \sum_{i,k} P(\theta_{i,j}) \cdot u_x(a_k, a_l, \theta_{i,j}) \cdot \pi_x(a_k|\theta_{x,i}) \geq \alpha_j \\
\forall i, \sum_k \pi_x(a_k|\theta_{x,i}) = 1 \\
\forall i, \forall k, \pi_x(a_k|\theta_{x,i}) \geq 0
\end{aligned}$$

2.2 SDM'S Implementation

In this section, we are going to discuss the SDM's Implementation to solve two-players zero-sum bayesian form games, and thus, indirectly, two-players zero-sum normal form games.

2.2.1 Model

As discussed earlier, an N-players bayesian game is represented by several things:
- a set of N agents
- a set of types for each agents
- a probability distribution over the joint types
- a set of normal-form games with the same number of agents and strategy space

Thus, an interface called *BayesianGameInterface* must be inherited when writing a class with specific choices for modelling the game information such as the classes *TwoPlayersBayesianGame* and *TwoPlayersNormalFormGame*. This interface enforce the definition of 4 main functions : *getNumAgents(), getActionSpace(), getTypeSpace(), getJointTypesProba(..)* and *getPayoff(..)*.

To actually model two-players bayesian games and two-players normal-form games, two subclasses have been created. Note that the *TwoPlayersNormalFormGame* class is, more or less, the same class as *TwoPlayersBayesianGame* with only one type per agent. Consequently, we will only dive in the *TwoPlayersBayesianGame* class.

In both of this subclasses, several informations about the game are built and stored :

- actions and types as **std::shared_ptr<MultiDiscreteSpace>**
- all the game's payoffs as a **RecursiveMap<std::shared_ptr<State>, std::shared_ptr<Action>, int, float>** where int is the agent id
- joint types probabilities as **RecursiveMap<std::shared_ptr<State>, float>**

A parser has been defined in the parser.cpp file that constructs such a game with the following input format :
- number of types for each agent separated by spaces
- number of actions for each agent separated by spaces
- payoffs of each normal form games with values for each agent being put in subsequent matrices
- joint types probabilities separated by spaces

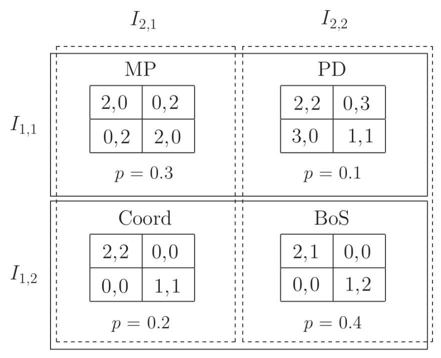
Figure 5b is an example of a Bayesian game [2] and the corresponding input file.

The instances built by this parser are then given to a solver : the *BayesianGameSolver* class which will write the Linear Programming problem when initialized using CPLEX library to represent it. Using CPLEX solve method with automatic selection of the best algorithm, it will then store the solution in a class attribute as a *StochasticDecisionRule*.
An example of how to use the solver is given in the file *examples/ex-bayesian.cpp*.

2.2.2 Algorithm evaluation

In order to evaluate the performances of our implementation, we perform a time-complexity analysis of it with a focus on the impact of types and actions distributions for the same input size. From now on, the term "input size" will refer to the number of constraints required for the LP plus the number of *(type, action)* possible pairs for the agent whose solution we seek. It is therefore computed with $|\Theta^i| \cdot (1 + |\mathcal{A}^i|) + |\Theta^{\neg i}| \cdot |\mathcal{A}^{\neg i}|$. LP building time and LP solving

[2]Note that the previous LP for this game is not a Nash-equilibrium as the game is not ZS. The LP solution can be interpreted as the maximum security level reachable for an agent.



(a) Bayesian game.

```
# number of types for each agent
2 2
# game dimensions i.e. #actions for each agent
2 2
#payoff values :
#   for joint_type = (0,0)
#       for agent 1
2 0
0 2
#       for agent 2
0 2
2 0
#   for joint_type = (0,1)
#       for agent 1
2 0
3 1
#       for agent 2
2 3
0 1
#   for joint_type = (1,0)
#       for agent 1
2 0
0 1
#       for agent 2
2 0
0 1
#   for joint_type = (1,1)
#       for agent 1
2 0
0 1
#       for agent 2
1 0
0 2
# joint types probabilities
#   for joint_type = (0,0) and (0,1)
0.3 0.1
#   for joint_type = (1,0) and (1,1)
0.2 0.4
```

(b) Input file.

Figure 5: Bayesian Game Input

time are evaluated independently for a better analysis of each of the algorithm's components. Note that, from a theoretical point of view, the number of constraints of the LP is $|\Theta^{\neg i}| \cdot |\mathcal{A}^{\neg i}| + |\Theta^i|$ when searching for i's strategy. Tests were performed over 50 instances of Bayesian games for each data point with uniform-like joint type probability distributions. We detail below the results for different input variations.

Let's first dive in the algorithm's behavior over changes in types and actions distributions. Types (resp. actions) distributions refer to the number of types (resp. actions) for each agent. Such distributions will be expressed in terms of percentages i.e. a types distributions of 50/50 means that each agent has the same number of types. Independent studies of types and actions distributions resulted in graphics 6 and 7.

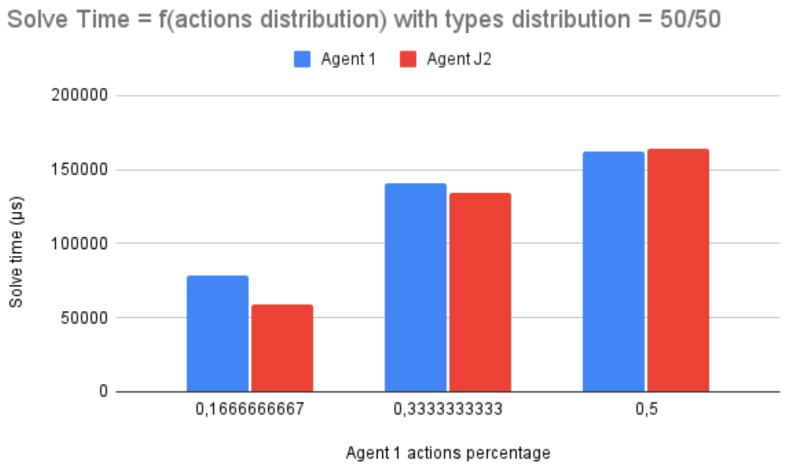


Figure 6: Actions distribution influence

We observe that the more equally distributed the actions or types are the faster the algorithm will perform. For the LP solving step, we can explain by the duality between searching for 1's strategy (primal) and 2's strategy (dual) [4] and CPLEX makes use of this duality to solve the LP. In our scenario, either the primal or the dual becomes easy to solve when one agent has very low number of actions/types which is why we observe better performances when the distribution is not balanced. The difference between agents solving time increasing with distribution imbalance may be due to the time CPLEX takes to build the dual instance.

Now that we have established that distributions have a direct impact on the algorithm's performances, let's look at its behavior regarding input size for

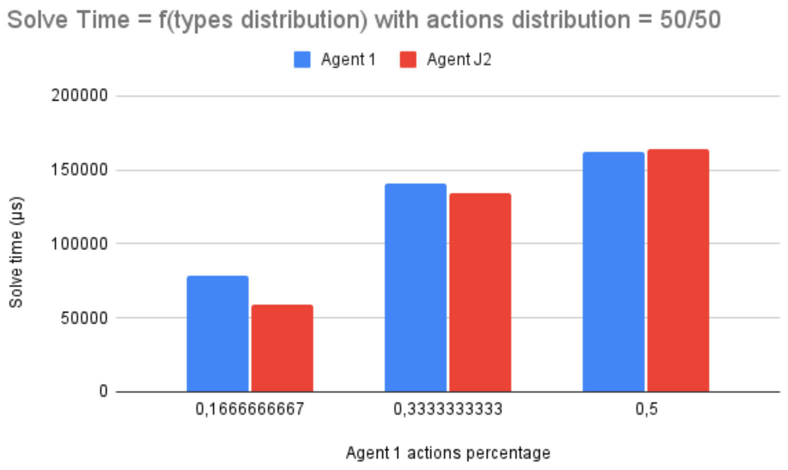


Figure 7: Types distribution influence

which all test instances distributions are set to 50/50.

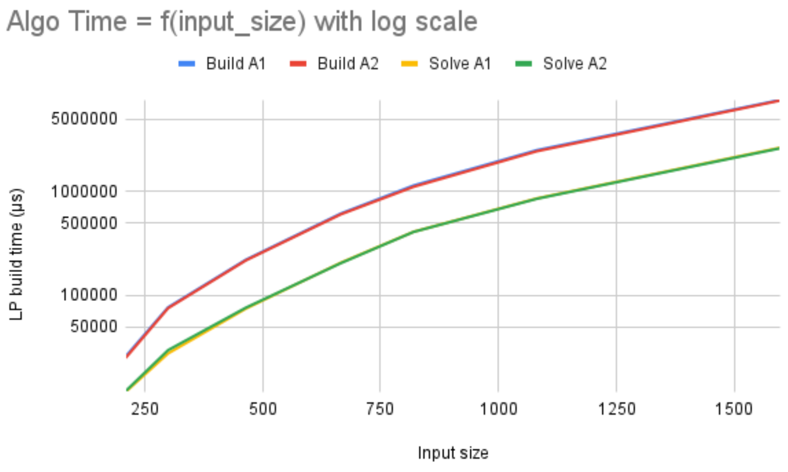


Figure 8: Input size influence

From what we see in Figure 8, we can conjecture two things. First, the LP building step is below exponential complexity. More tests should show that it runs in linear time regarding input size i.e. in polynomial time regarding number of actions and types for each agent. This is expected since we said that building time's complexity is $|\theta^1| \cdot |\theta^2| \cdot |\mathcal{A}^1| \cdot |\mathcal{A}^2|$. Then, the LP solving step is shown to be below exponential complexity as well. It is an expected result as LP solving is known for being a polynomial time problem and CPLEX should manage to choose the best way to solve a LP thanks to its LP examination step. We can also check that there is no significant running time difference between each agent for the same reasons as exposed above (duality and 50/50 distributions).

2.3 Heuristic Search approach

The previous results i.e. polynomial time complexity suggests that this complete LP approach would be intractable for complex Bayesian games (i.e. with a high number of types and/or actions and/or with equally distribution actions/types according to the previous observations). Thus, we wish to investigate a second approach in which the solution could be ϵ-optimal but with a significant time reduction. Our strategy is not to build the entire LP at first but to do it step by step until a quality criteria (e.g. convergence criteria) is reached. To do so, we propose to model the Bayesian game as sequential game whose tree representation has the semantic pictured in Figure 9.

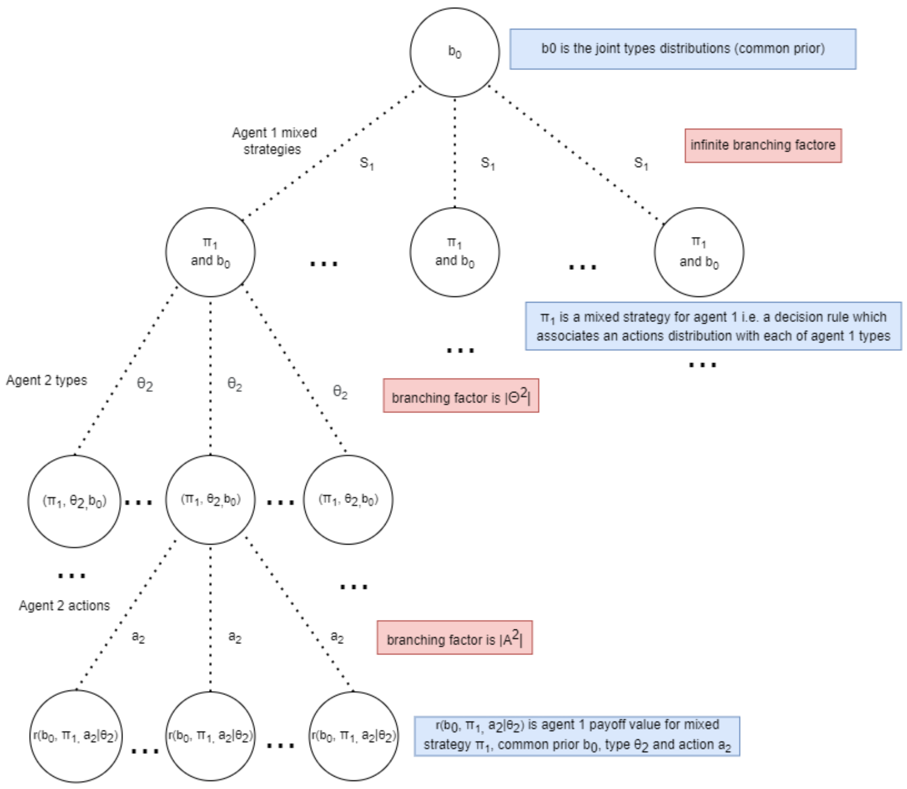

b0
b0 is the joint types distributions (common prior)
infinite branching factore
Agent 1 mixed strategies
s1
π1 and b0
π1 is a mixed strategy for agent 1 i.e. a decision rule which associates an actions distribution with each of agent 1 types
Agent 2 types
θ2
branching factor is |Θ2|
(π1, θ2, b0)
Agent 2 actions
a2
branching factor is |A2|
r(b0, π1, a2|θ2)
r(b0, π1, a2|θ2) is agent 1 payoff value for mixed strategy π1, common prior b0, type θ2 and action a2

Figure 9: Bayesian game tree

As the first branching factor is infinite, we aim at exploring these nodes by selecting the next node as the solution of a LP. The constraints of this LP are possible responses $\theta^2 \to a^2$ from player 2. They are defined by previous exploration of the tree second level for which the leaf selection will be performed through an heuristic search approach (e.g. sampling over joint types distribution).

Formally, the Bayesian game is given by a tuple $(\Theta^1, \Theta^2, \mathcal{A}^1, \mathcal{A}^2, r, b_0)$ where

- Θ^i is a finite set of types for player i
- $\mathcal{A}^i$ is a finite set of actions for player i
- $r : \Theta^1 \times \Theta^2 \times \mathcal{A}^1 \times \mathcal{A}^2$ is the utility function giving the payoff for 1 playing a^1 in type θ^1 and 2 playing a^2 in θ^2.
- $b_0 \in \Delta(\Theta^1 \times \Theta^2)$ is the initial distribution over player's types.

and the sequential game representation is given by

- the initial node $b_0 \in \Delta(\Theta^1 \times \Theta^2)$,

- The edges from b_0 represent stochastic strategies for player 1 $\pi_1 \in \Theta^1 \mapsto \Delta(\mathcal{A}^1)$
- level 1 nodes as are tuples (π_1, b_0)
- edges from nodes (π_1, b_0) represent all possible types $\theta^2 \in \Theta^2$ for player 2,
- level 2 nodes are triplets (π_1, b_0, θ^2)
- Edges from nodes (π_1, b_0, θ^2) are deterministic actions $a^2 \in \mathcal{A}^2$ for player 2 knowing the specific type θ^2
- level 3 nodes represent the payoff function $r(b_0, \pi_1, a^2|\theta^2) \in \mathbb{R}$ i.e. the game payoff for strategy π_1, common prior b_0, θ^2 and a^2.

The evaluation of a joint strategy (π_1, π_2) will then be given by $r(b_0, \pi_1, \pi_2) = \sum_{\theta^2} \mathbb{P}(\theta^2 \mid b_0) r(b_0, \pi_1, \pi_2(\theta^2)|\theta^2)$ with $\pi_2 \in \Theta^2 \mapsto \mathcal{A}^2$ the nodes under the level 2 node representing π_1.

2.3.1 Properties of the game tree

Our motivations to introduce the algorithm 1 are based on mathematical properties of such a tree. The following theorems are the main properties we want to focus on to justify the exploration of such a solving algorithm.

Theorem 1 *The function* $v^* : b_0, \pi^1, \pi^2 \mapsto r(\pi^1, \pi^2, b_0)$ *is linear w.r.t* π^2

Proof 1 *Let* R *be the game matrix s.t.* $R = \{b_0(\theta^1, \theta^2) r(\theta^1, \theta^2, a^1, a^2)\}_{(\theta^1, a^1), (\theta^2, a^2)}$. *By definition, we have*

$$\begin{aligned}
r(\pi^1, \pi^2, b_0) &= \sum_{a^1, \theta^1, a^2, \theta^2} \pi^1(a^1|\theta^1) \cdot \pi^2(a^2|\theta^2) \cdot b_0(\theta^1, \theta^2) r(\theta^1, \theta^2, a^1, a^2) \\
\Leftrightarrow r(\pi^1, \pi^2, b_0) &= \sum_{a^1, \theta^1, a^2, \theta^2} \pi^1(a^1|\theta^1) \cdot R(\theta^1, \theta^2, a^1, a^2) \cdot \pi^2(a^2|\theta^2) \\
\Leftrightarrow r(\pi^1, \pi^2, b_0) &= \sum_{a^1, \theta^1} \pi^1(a^1|\theta^1) \left[\sum_{a^2, \theta^2} R(\theta^1, \theta^2, a^1, a^2) \cdot \pi^2(a^2|\theta^2) \right] \\
\Leftrightarrow r(\pi^1, \pi^2, b_0) &= \sum_{a^1, \theta^1} \pi^1(a^1|\theta^1) \left[R \cdot \pi^2 \right] \\
\Leftrightarrow r(\pi^1, \pi^2, b_0) &= \pi^1 \left[R \cdot \pi^2 \right] = \pi^1 \cdot R \cdot \pi^2.
\end{aligned}$$

Thus, $r(\pi^1, \pi^2, b_0) = \pi^1 \cdot R \cdot \pi^2$

Theorem 2 *The function* $v^* : b_0, \pi^1 \mapsto \min_{\pi^2} r(\pi^1, \pi^2)$ *can be maximized by solving a linear program.*

Proof 2 *We have already established, in section 2.1, that the corresponding LP is the following.*

$$\forall j \in |\Theta^2|, \pi_1^* = \arg\max_{\pi_1, \alpha_j} \sum_j \alpha_j$$

$$\textit{and with the constraints :}$$
$$\textit{with } \theta^1 \in \Theta^1,$$
$$\textit{with } j \in |\Theta^2|,$$
$$\textit{with } a^1 \in \mathcal{A}^1,$$
$$\textit{with } a^2 \in \mathcal{A}^2,$$
$$\forall j, \forall a^2, \sum_{\theta^1, a^1} b_0(\theta^1, \theta_j^2) \cdot \pi^1(a^1|\theta^1) \cdot v(a^1, a^2, \theta^1, \theta_j^2) \geq \alpha_j$$
$$\forall \theta^1, \sum_{a^1} \pi^1(a^1|\theta^1) = 1$$
$$\forall \theta^1, \forall a^1, \pi^1(a^1|\theta^1) \geq 0$$

Theorem 3 *The function $v^* : b_0 \mapsto \max_{\pi^1} \min_{\pi^2} r(\pi^1, \pi^2)$ is Lipschitz*

Proof 3 *Firstly, [5] (Lemma 3.5 (p. 33)) showed that a bi-linear function bounded by $[u, v] \subset \mathbb{R}^2$ defined over a simplex is λ-Lipschitz with $\lambda = (u - v)/2$. Thus, the function $b_0 \mapsto r(b_0, \vec{\pi})$ is Lipschitz for any joint strategy $\vec{\pi}$.*

Now considering optimal solutions,

$$\begin{aligned} v^*(b_0) - v^*(\tilde{b}_0) &= \max_{\pi^1} \min_{\pi^2} r(b_0, \pi^1, \pi^2) - \max_{\pi^1} \min_{\pi^2} r(\tilde{b}_0, \pi^1, \pi^2) \\ &\leq \max_{\pi^1} \min_{\pi^2} \left[r(\tilde{b}_0, \pi^1, \pi^2) + \lambda \|b_0 - \tilde{b}_0\| \right] - \max_{\pi^1} \min_{\pi^2} r(\tilde{b}_0, \pi^1, \pi^2) \\ &= \lambda \|b_0 - \tilde{b}_0\|. \end{aligned}$$

Symetrically, $v^*(b_0) - v^*(\tilde{b}_0) \leq \lambda \|b_0 - \tilde{b}_0\|$. Thus, v^* is $(\max_{(i,j)} R_{i,j} - \min_{(i,j)} R_{i,j})/2$-Lipschitz (where R is the matrix representing the linear form r).

2.3.2 Heuristic search algorithm

All these theoretical concepts seems to draw the path for an algorithm that could perform well for Bayesian games. The algorithm we chose to implement and investigate is called Heuristic Search for Bayesian Game (HS4BG). Its overall principle is to instantiate two different LPs, one to optimize player 1's strategy and one to optimize player 2's strategy. As we focus on zero-sum games, the absolute values of expected payoffs of both this strategies should tend to be equal as the constraints of the LP grow and diversify. Thus, the principle of the algorithm is to first initialize both LP and random strategies for each players and then to update it with the following process : select a type to focus on through heuristic search for each player, select the action which is the best response to this type for each agent, add the corresponding constraints to the LP and update players' strategies as LP solutions. As we could repeat this procedure forever, we define the convergence criteria to the difference between absolute values of expected utilities being less than an arbitrary value epsilon. This parameter should be proportional to the game payoff profile as it should represent an reasonable deviation from the very optimal solution. The formal description is depicted by the following algorithm.

Algorithm 1: HS4BGv2

```
1:  Initialize $d^{\neg i} = \{\theta^{\neg i} \to a^{\neg i}\}$ and $\pi^i = random()$
2:  while $\overline{v}(b_0) - \underline{v}(b_0) > \epsilon$ do
3:      $\theta^{\neg i,*} = HS(\pi^i)$
4:      Replace $d^{negi}(\theta^{\neg i,*})$ by $\theta^{\neg i,*} \to \arg\min_{a^{\neg i}} r(b_0, \pi^i, a^{\neg i})$
5:      UpdateLPConstraints($d^{\neg i,\pi^i}$)
6:      $\pi^1 \leftarrow \overline{LP}.solution$, $\pi^2 \leftarrow \underline{LP}.solution$
7:      $\overline{v}(b_0) = \overline{LP}.val$
8:      $\underline{v}(b_0) = \underline{LP}.val$
9:  end while
10: return
```

This algorithm makes use of an heuristic search function that can affect greatly its results. In this first exploration steps, we propose three version of this heuristic search.

2.3.2.1 Naive Heuristic Search returns a type according to its probability.

2.3.2.2 Medium Heuristic Search returns a type according to $\mathbb{P}(\theta^2) \cdot \frac{\Delta(\theta^2)}{R_{max} - R_{min}}$ where $\Delta(\theta^2)$ is the last variation of value for type θ^2 and equals 0 by default. This is encouraged by the fact that π^1 has to make the function $\pi^2 \mapsto r(\pi^1, \pi^2)$ constant so that the better π^1 becomes, the closer the variation should be to 0.

2.3.2.3 Advanced Use oracle method, i.e restrict the support of π^i to be a distribution over already encountered pure strategies on the other tree that were "best" responses to $\pi^{\neg i}$.

2.3.3 SDMS's Implementation

The heuristic search solver's implementation on SDM'S takes advantage of all side classes described in section 2.2.1. However, instead of using the *TwoPlayersBayesianGameSolver* class, it is based on the *HS4BG* class which contains all utility functions, such as *initLP()*, *bestResponse(..)*, *updateLP(..)*, *updateStragegies(..)*, and the C++ implementation of algorithm 1 with the *solve()* method. An example file is given with *examples/ex-hs4bg.cpp* of how to use this solver, which is the exact same way as the *TwoPlayersBayesianGameSolver*.

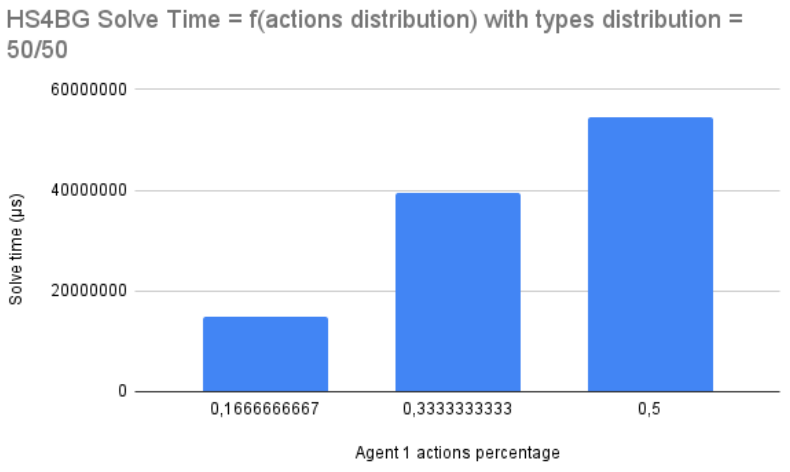


Figure 10: HS4BG with naive HS : actions distribution influence

One main difference with the *TwoPlayersBayesianGameSolver* is that this solver does not allow to solve the game for a specific agent. Instead, it does so for both agents and store the resulting strategies as instances of *StochasticDecisionRule*.

2.3.4 Algorithm's evaluation

In order to draw a comparison with the complete LP algorithm (see section 2.2), we perform the same tests as those in section 2.2. We detail below the results obtained for the HS4BG algorithm with the naive version of the Heuristic Search.

Once again, let's discuss types and actions distributions influence first. Graphics 10 and 11 show that, once again, the algorithm performs better when distributions are imbalanced. Thus, we can admit that this approach also takes advantage of the primal-dual property. i.e. when there are imbalances, a step of the algorithm (LP solving) is faster and the number of constraints required to obtain an ϵ-optimal decreases.

Then, we are interested in comparing the naive heuristic search approach with the complete LP algorithm regarding the sizes of the game to be solved. Figure 12 shows that the HS4BG algorithm with the naive HS is slower than the complete LP approach for games with input size under 1600. Its time complexity, however, seems to also be polynomial as the graphs show the same tendency.

What's good is that the HS4BG algorithm seems to always converge towards an ϵ-optimal solution. The graphs of Figure 13 show the LP solution value evolution at each step of the HS4BG algorithm for games with different sizes and

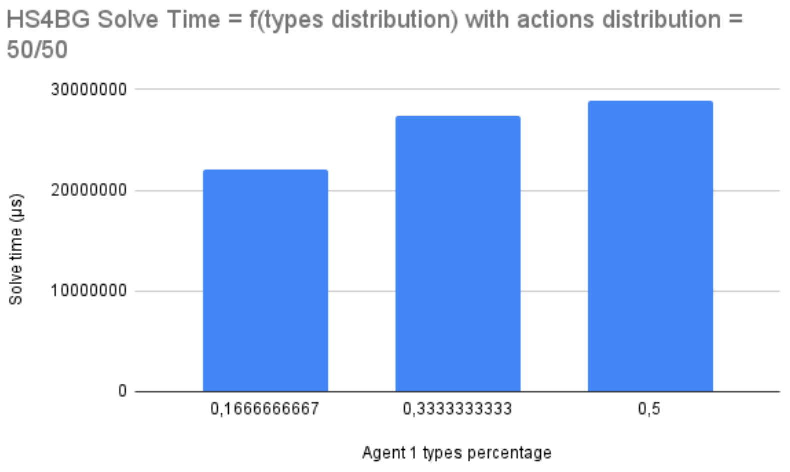


Figure 11: HS4BG with naive HS : types distribution influence

two different values of ϵ.

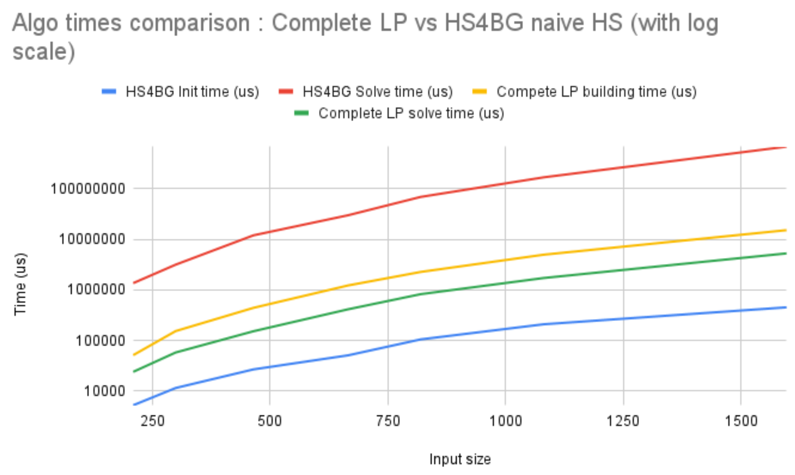
As of now, the results obtained with this version of the HS4BG algorithm show that the naive HS implementation may always be less attractive than the complete LP approach. Let's see how the medium HS performs.

Figure 14 show that the medium HS performed even worse than the naive HS on test data. The complexity tendency is polynomial as well but, for any game size, more steps are needed to converge to an ϵ-optimal solution with the medium HS than with the naive HS.
Figures 15 and 16 also demonstrate the same behaviour than previously observed with players imbalances on number of types and/or actions.

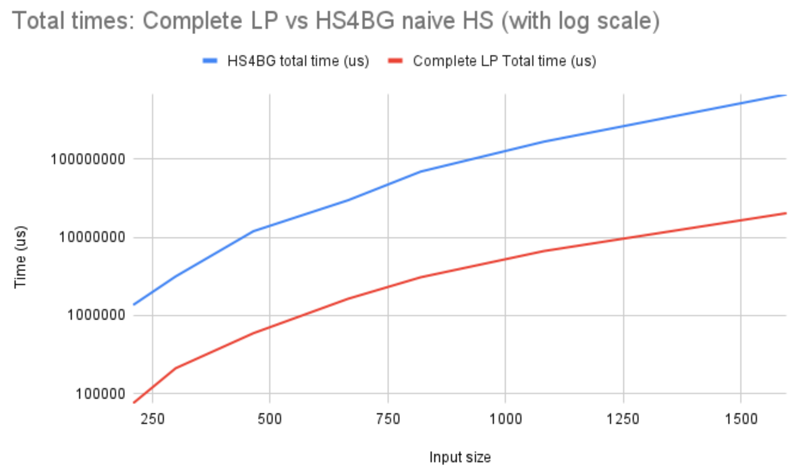
However, even though it takes more steps, the algorithm seems to always converge with the medium HS as shown by Figure 17.

Through these results, we tend to think that weighting the probability of a type to be chosen by its last learning contribution is not accurate i.e. we need to focus on exploring new types.

The advanced HS version that we propose could, indeed, result in faster computation time through faster convergence. However, its implementation and testing won't be featured in this document.

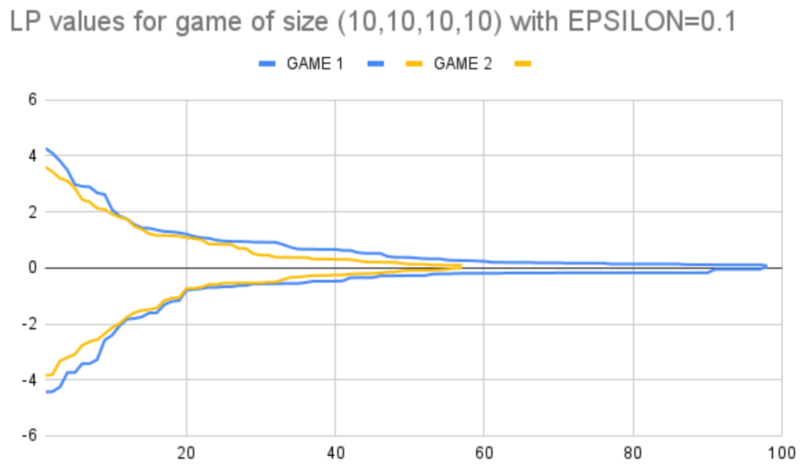


(a) Algorithm steps comparison

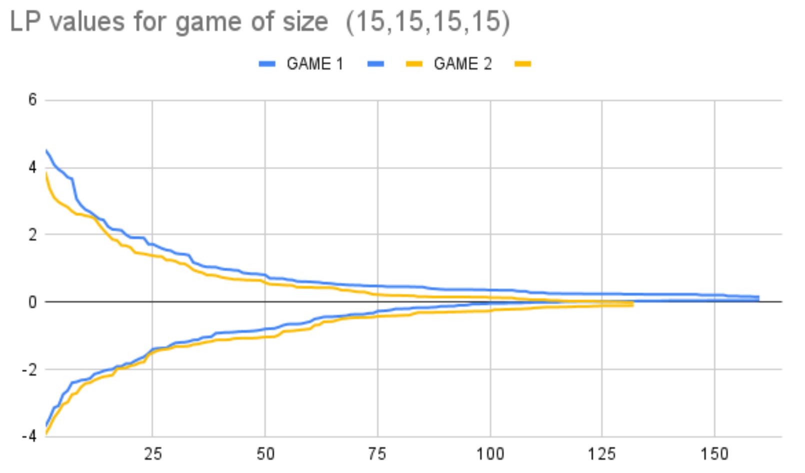


(b) Total time comparison

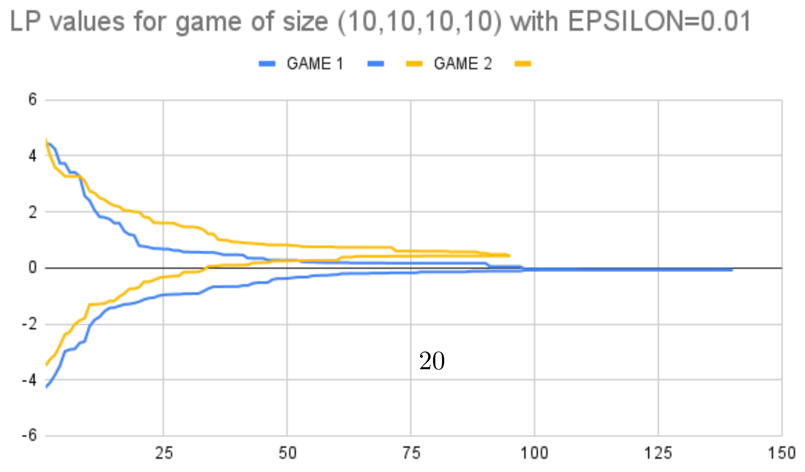
Figure 12: HS4BG vs Complete LP time comparison



(a) (10,10,10,10) Games with ϵ=0.1



(b) (15,15,15,15) Games with ϵ=0.1



(c) (10,10,10,10) Games with ϵ=0.01

Figure 13: LP values for HS4BG with naive HS

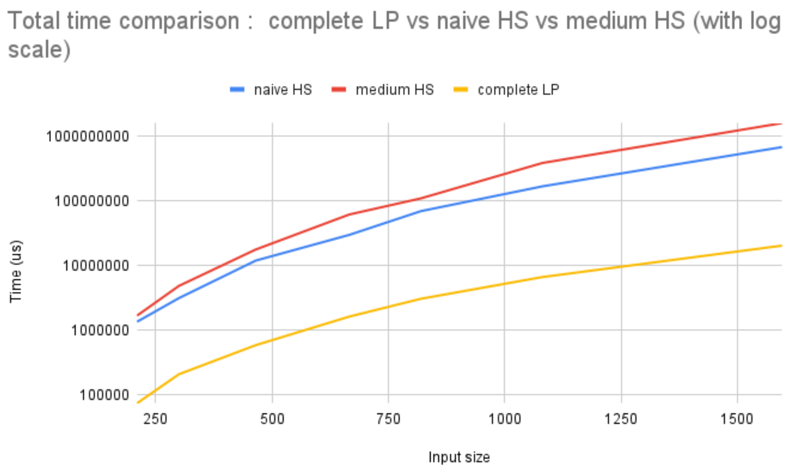


Figure 14: Total times comparison : complete LP vs HS4BG naive HS vs HS4BG medium HS

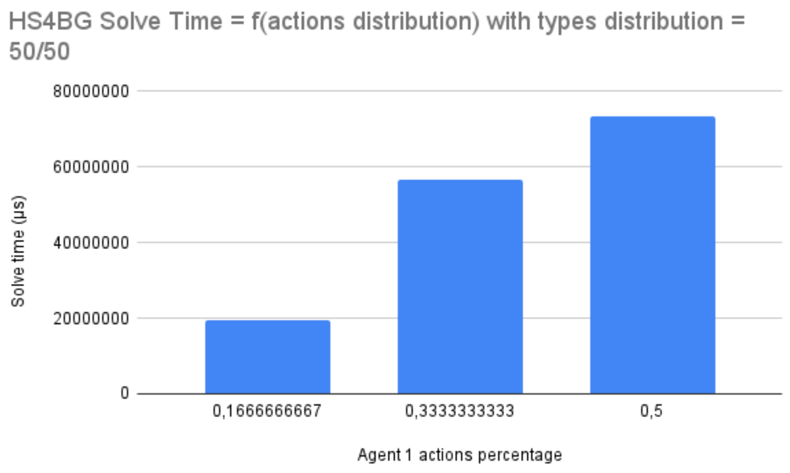


Figure 15: HS4BG with medium HS : actions distribution influence

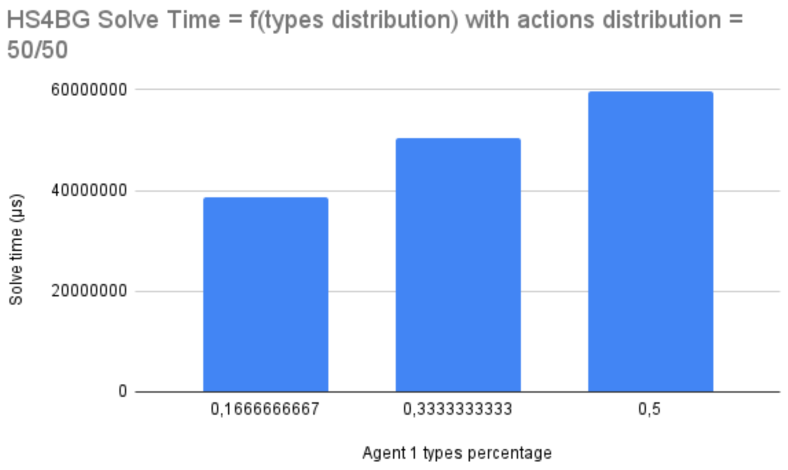


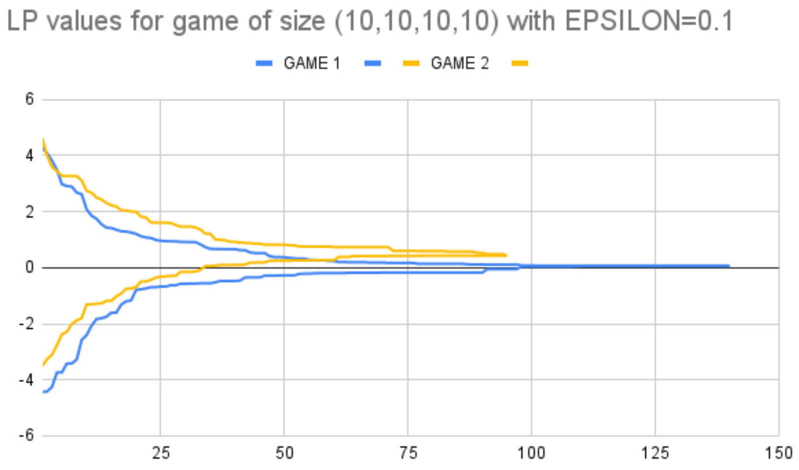
Figure 16: HS4BG with medium HS : types distribution influence

2.4 Discussion

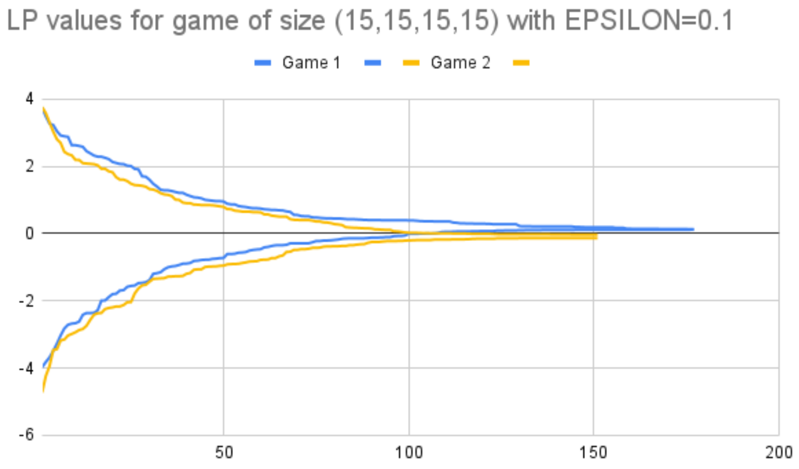
All the previous analysis have been performed on games with a type probability distribution close to uniform. In order to better understand the HS performances and fits, we should later investigate other distributions e.g. Normal distribution.

3 Conclusion

Throughout this document, we have established basic solving theory on ZS normal-form and Bayesian games, detailed their SDM's implementation and analysed the results. In order to do better, we proposed an alternative way of solving Bayesian games by introducing a sequential representation of the game and building the associated linear problems step by step through an heuristic search approach. The results for two different search heuristics showed that there is room for improvement and that it can be performed by choosing better heuristics. However, we observed that this new approach can indeed solve Bayesian games and that the convergence speed depends on many features in the game which could make this approach more suited for a specific type of Bayesian games.



(a) (10,10,10,10) Games with ϵ=0.1



(b) (15,15,15,15) Games with ϵ=0.1

Figure 17: LP values for HS4BG with medium HS

References

[1] Shmuel Zamir. Bayesian Games: Games with Incomplete Information, pages 426–441. Springer New York, New York, NY, 2009.

[2] Olivier Armantier, jean-pierre Florens, and Jean-Francois Richard. Approximation of nash equilibria in bayesian games. Journal of Applied Econometrics, 23:965–981, 11 2008.

[3] Kevin Leyton-Brown and Yoav Shoham. Essentials of Game Theory: A Concise Multidisciplinary Introduction, volume 2. 01 2008.

[4] Minimax Theorem. https://en.wikipedia.org/wiki/Minimax#Minimax_theorem.

[5] Karel Horák. Scalable Algorithms for Solving Stochastic Games with Limited Partial Observability. PhD thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2019.