POLITECNICO
MILANO 1863

Project for the course
"Advanced Programming for Scientific Computing"
Tutorized by Nicola Gatti

MARKOV PERSUASION PROCESSES: LEARNING TO PERSUADE FROM SCRATCH

Miguel Alcañiz Moya

Lecturers: Prof. L. Formaggia and Prof. C. De Falco

Date: July 17th, 2024

Contents

Abstract

In this project we try to implement the algorithms from the paper Markov Persuasion Processes: Learning to Persuade from Scratch [1]. In Bayesian persuasion, an informed sender strategically discloses information to a receiver so as to persuade them to undertake desirable actions. Recently, Markov persuasion processes (MPPs) have been introduced to capture sequential scenarios where a sender faces a stream of myopic receivers in a Markovian environment. The MPPs studied so far in the literature suffer from issues that prevent them from being fully operational in practice, e.g., they assume that the sender knows receivers' rewards. We fix such issues by addressing MPPs where the sender has no knowledge about the environment. We are testing a learning algorithm for the sender in which attain regret sublinear in the number of episodes T while being persuasive.

Chapter 1

Introduction

1.1 Introduction to MPPs

Bayesian persuasion studies how an informed sender should strategically disclose information to influence the behavior of an interested receiver. The vast majority of works on Bayesian persuasion focuses on one-shot interactions, where information disclosure is performed in a single step. Despite the fact that real-world problems are usually sequential, there are only few exceptions that consider multi-step information disclosure [4].

In particular, Wu et al. (2022) ([4]) initiated the study of Markov persuasion processes (MPPs), which model scenarios where a sender sequentially faces a stream of myopic receivers in an unknown Markovian environment. In each state of the environment, the sender privately observes some information—encoded in an outcome stochastically determined according to a prior distribution—and faces a new receiver, who is then called to take an action. The outcome and receiver's action jointly determine agents' rewards and the next state. In an MPP, sender's goal is to disclose information at each state so as to persuade the receivers to take actions that maximize long-term sender's expected rewards. MPPs find application in several real-world settings, such as e-commerce and recommendation systems. For example, an MPP can model the problem faced by an online streaming platform recommending movies to its users. Indeed, the platform has an informational advantage over users (e.g., it has access to views statistics), and it exploits available information to induce users to watch suggested movies.

Nevertheless, the MPPs studied by Wu et al. (2022) ([4]) suffer from several issues that prevent them from being fully operational in practice. In particular, they make the rather strong assumption that the sender has perfect knowledge of receiver's rewards. This is unreasonable in real-world applications. For instance, in the online streaming platform example described above, such an assumption requires that the platform knows everything about users' (private) preferences over movies.

In our setting we considerably relax the assumptions of Wu et al. (2022) ([4]), by addressing MPPs where the sender does not know anything about the environment. We consider settings in which the sender has no knowledge about transitions, prior distributions over outcomes, sender's stochastic rewards, and receivers' ones. Thus, they have to learn all these quantities simultaneously by repeatedly interacting with the MPP.

Summaring, the algorithm implemented is mixing the classic Bayesian Persuasion problem with a Markov decision problem, trying to bring this area to a more realistic setting for the real world in which we do not know how does the enviroment work, so we have to learn about it by interacting with receivers. The enviroment unkown by the sender refers to the rewards won from the interactions and the probability transitions between the states.

1.2 Bayesian Persuasion

The classical Bayesian persuasion framework introduced by Kamenica and Gentzkow (2011) [1] models a one-shot interaction between a sender and a receiver. The latter has to take an action a from a finite set A, while the former privately observes an outcome ω sampled from a finite set Ω according to a prior distribution $\mu \in \Delta(\Omega)$, which is known to both the sender and the receiver. The rewards of both agents depend on the receiver's action and the realized outcome, as defined by the functions

$$r_S, r_R : \Omega \times A \to [0, 1],$$

where $r_R(\omega, a)$ and $r_S(\omega, a)$ denote the rewards of the sender and the receiver, respectively, when the outcome is $\omega \in \Omega$ and action $a \in A$ is played.

The sender can strategically disclose information about the outcome to the receiver, by publicly committing to a signaling scheme ϕ, which is a randomized mapping from outcomes to signals being sent to the receiver. Formally,

$$\phi : \Omega \to \Delta(S),$$

where S denotes a suitable finite set of signals. For ease of notation, we let $\phi(\cdot|\omega) \in \Delta(S)$ be the probability distribution over signals employed by the sender when the realized outcome is $\omega \in \Omega$, with $\phi(s|\omega)$ being the probability of sending signal $s \in S$.

The sender-receiver interaction goes on as follows:

1. the sender publicly commits to a signaling scheme ϕ;
2. the sender observes the realized outcome $\omega \sim \mu$ and draws a signal $s \sim \phi(\cdot|\omega)$; and
3. the receiver observes the signal s and plays an action. Specifically, after observing s under a signaling scheme ϕ, the receiver infers a posterior distribution over outcomes and plays a best-response action $b_\phi(s) \in A$ according to such distribution. Formally:

$$b_\phi(s) \in \arg\max_{a \in A} \sum_{\omega \in \Omega} \mu(\omega)\phi(s|\omega)r_R(\omega, a),$$

Where the expression being maximized encodes the (unnormalized) expected reward of the receiver. As it is customary in the literature [1], we assume that the receiver breaks ties in favor of the sender, by selecting a best response maximizing sender's expected reward when multiple best responses are available.

The goal of the sender is to commit to a signaling scheme ϕ that maximizes their expected reward, which is computed as follows:

$$\sum_{\omega \in \Omega} \mu(\omega) \sum_{s \in S} \phi(s|\omega) r_S(\omega, b_\phi(s)).$$

If you need a better understanding of the problem have a look at the paper *Public Signaling in Bayesian Ad Auctions* [2].

1.3 Markov Persuasion Processes

A Markov persuasion process (MPP) (Wu et al., 2022) [4] generalizes the one-shot Bayesian persuasion framework by Kamenica and Gentzkow (2011) [1] to settings in which the sender sequentially interacts with multiple receivers in a Markov decision process (MDP). In an MPP, the sender faces a stream of myopic receivers who take actions by only accounting for their immediate rewards, thus disregarding future ones.

A Markov persuasion process (MPP) (Wu et al., 2022) generalizes the one-shot Bayesian persuasion framework by Kamenica and Gentzkow (2011) to settings in which the sender sequentially interacts with multiple receivers in a Markov decision process (MDP). In an MPP, the sender faces a stream of myopic receivers who take actions by only accounting for their immediate rewards, thus disregarding future ones.

Formally, an (episodic) MPP is defined by means of a tuple

$$M = (X, A, \Omega, \mu, P, \{r_{S,t}\}_{t=1}^{T}, \{r_{R,t}\}_{t=1}^{T}),$$

where:

- T is the number of episodes.
- X, A, and Ω are finite sets of states, actions, and outcomes, respectively.
- $\mu : X \to \Delta(\Omega)$ is a prior function defining a probability distribution over outcomes at each state. For ease of notation, we let $\mu(\omega \mid x)$ be the probability with which outcome $\omega \in \Omega$ is sampled in state $x \in X$.
- $P : X \times A \to \Delta(X)$ is a transition function. For ease of notation, we let $P(x' \mid x, a)$ be the probability of moving from $x \in X$ to $x' \in X$ by taking action $a \in A$.

- $\{r_{S,t}\}_{t=1}^{T}$ is a sequence specifying a sender's reward function $r_{S,t} : X \times \Omega \times A \to [0,1]$ at each episode t. Given $x \in X, \omega \in \Omega$, and $a \in A$, each $r_{S,t}(x,\omega,a)$ for $t \in [T]$ is sampled independently from a distribution $\nu_S(x,\omega,a) \in \Delta([0,1])$ with mean $r_S(x,\omega,a)$.

- $\{r_{R,t}\}_{t=1}^{T}$ is a sequence defining a receivers' reward function $r_{R,t} : X \times \Omega \times A \to [0,1]$ at each episode t. Given $x \in X, \omega \in \Omega$, and $a \in A$, each $r_{R,t}(x,\omega,a)$ for $t \in [T]$ is sampled independently from a distribution $\nu_R(x,\omega,a) \in \Delta([0,1])$ with mean $r_R(x,\omega,a)$.

For ease of presentation, we focus w.l.o.g. on episodic MPPs enjoying the loop-free property, as customary in the online learning in MDPs literature. In a loop-free MPP, states are partitioned into $L+1$ layers $X_0, \ldots, X_L$ such that $X_0 := \{x_0\}$ and $X_L := \{x_L\}$, with x_0 being the initial state of the episode and x_L being the final one, in which the episode ends. Moreover, by letting $\mathcal{K} := [0 \ldots L-1]$ for ease of notation, $P(x' \mid x, a) > 0$ only when $x' \in X_{k+1}$ and $x \in X_k$ for some $k \in \mathcal{K}$

At each episode of an episodic MPP, the sender publicly commits to a signaling policy $\phi : X \times \Omega \to \Delta(\mathcal{S})$, which defines a probability distribution over a finite set $\mathcal{S}$ of signals for the receivers for every state $x \in X$ and outcome $\omega \in \Omega$. In our setting we will take $\Omega = A$. For ease of notation, we denote by $\phi(\cdot \mid x, \omega) \in \Delta(\mathcal{S})$ such probability distributions, with $\phi(s \mid x, \omega)$ being the probability of sending a signal $s \in \mathcal{S}$ in state x when the realized outcome is ω.

As customary in Bayesian persuasion settings, a revelation-principle-style argument allows to focus w.l.o.g. on signaling policies that are direct and persuasive. Formally, a signaling policy is direct if the set of signals coincides with the set of actions, namely $\mathcal{S} = A$. Intuitively, in such a case, signals should be interpreted as action recommendations for the receivers.

So our first algorithm to we implement is the sender-receivers interaction in the sequential decision process defined previously.

Algorithm 1 Sender-Receivers Interaction at $t \in [T]$

All the rewards $r_{S,t}(x,\omega,a), r_{R,t}(x,\omega,a)$ are sampled
Sender publicly commits to $\phi_t : X \times \Omega \to \Delta(A)$
The state of the MPP is initialized to x_0
for $k = 0, \ldots, L-1$ **do**
    Sender observes outcome $\omega_k \sim \mu(x_k)$
    Sender draws recommendation $a_k \sim \phi(\cdot|x_k, \omega_k)$
    A *new* Receiver observes a_k and plays it
    The MPP evolves to state $x_{k+1} \sim P(\cdot|x_k, \omega_k, a_k)$
    Sender observes the next state x_{k+1}
Sender observes *feedback* for every $k \in [0 \ldots L-1]$:
    • *full* $\to r_{S,t}(x_k, \omega_k, a), r_{R,t}(x_k, \omega_k, a) \quad \forall a \in A$
    • *partial* $\to r_{S,t}(x_k, \omega_k, a_k), r_{R,t}(x_k, \omega_k, a_k)$

The interaction between the sender and the stream of receivers at episode $t \in [T]$ is described in Algorithm 1. Let us remark that sender and receivers do not know anything about the transition function P, the prior function mu, and the rewards $r_{R,t}(x,\omega,a)$, $r_{R,t}(x,\omega,a)$ (including their distributions). At the end

of each episode, the sender gets to know the triplets (x_k, ω_k, a_k) $\forall k \in K$ that are visited during the episode, and an additional feedback about rewards. Although in the paper this project it's inspired of they consider two types of feedback, we just consider the feedback of the rewards for all the triplets (x_k, ω_k, a_k) visited during the episode.

1.4 Optimisitc Persuasive Policy Search

The algorithm proposed in [1] we propose an algorithm called Optimisitc Persuasive Policy Search (OPPS). At each episode, the algorithm solves a variation of the offline optimization problem called Opt-Opt, defined in Appendix C of [1]. Crucially, by using occupancy measures, Opt-Opt can be formulated as a linear program, and, thus, it can be solved efficiently. Although in this project we couldn't finally make it to program this part. So the signaling scheme obtained out of this problem in our program is just a random function which outputs a valid signaling scheme. Because of this issue, significant results proved in [1] could not be tested. The Optimistic Persuasive Policy Search with full feedback of the following code correspond to the incomplete OPPS algorithm programmed in this project.

Algorithm 2 Optimistic Persuasive Policy Search *(full)*

Require: X, A, T, confidence parameter $\delta \in (0, 1)$

Initialize all estimators to 0 and all bounds to $+\infty$
for $t = 1, \ldots, T$ **do**
    Update all estimators $\overline{P}_t, \overline{\mu}_t, \overline{r}_{S,t}, \overline{r}_{R,t}$ and bounds $\epsilon_t, \zeta_t, \xi_{S,t}, \xi_{R,t}$ given new observations
    $\widehat{q}_t \leftarrow$ Solve `Opt-Opt` (Problem (2), Appendix C)
    $\phi_t \leftarrow \phi^{\widehat{q}_t}$
    Run Algorithm 1 by committing to ϕ_t
    Observe *full* feedback from Algorithm 1

For more understanding of the algorithm and the analysis of its performance look at the paper 'Markov persuasion processes: learning to persuade from scratch' [1].

Chapter 2

Implementation

The algorithm implementation is composed by a c++ code formed by three main parts, the episode generator files, the Opt Opt files and the Markov Persuasion Process part. Each part consists of a *.hpp* file and a *.cpp* file. All these files are kept in the */src* folder.

2.1 Episode Generator

This first part is where the environment setting for the Markov Persuasion Process is defined. Referring to this environment setting the following functions and sets:

- The prior function $\mu : X \to \Delta(\Omega)$. Defining a probability distribution over outcomes at each state.
- The transition function $P : X \times A \to \Delta(X)$. Defining the probability to move to another possible next state given a chosen action.
- The rewards container. Which is used to save the rewards for the sender and receiver for every possible state, action and outcome.

2.1.1 Prior function

The header file defines the `prior` class. It refers to the probability distribution over outcomes at each state. We let μ(l, x) be the vector of probabilities with which outcomes are sampled in a state.
(Inputs l and s determine the state of S)
(Input w determine the outcome)
Its protected components are the following:

- The protected member function *size_t L* correspond to the number of partitions of the states set $\mathcal{S}$.

- The protected member function *size_t A* correspond to the number of possible actions of the set $\mathcal{A}$.
- The protected member function *TensorI states* correspond to the vector of states of every partition of $\mathcal{S}$.
- The protected member function *Tensor3D priorD* correspond to a three dimensional tensor of doubles that keeps the probability of having an outcome o in each state s of partition l ($priorD[l][s][o]$).

```
    protected:
        size_t L;
        size_t A;
        TensorI states;
        Tensor3D priorD;
    };
```

Then it has as public constructors:

- The default constructor.
- The constructor that initializes the prior class with given state values and action size.

```
    prior() = default;

    prior(const TensorI &states_values, const size_t A_value){
      A = A_value;
      L = states_values.size();
      states = states_values;

      priorD.resize(L);
      for(int l = 0; l < L; l++)
        priorD[l].resize(states[l]);
    };
```

And finally as public methods and operators:

- The method to initialize the prior with given state values and action size, used with the default constructor.
- Method to set the probability distribution for a specific state.
- Method to get the probability distribution for a specific state.
- Method to get the probability to get outcome (w) for a specific state (l,s).
- Method to generate a random outcome based on the prior probability distribution for a given state.

- Stream operator to print the prior distribution.

```
init_prior(const TensorI &states_values, const size_t A_value){
   A = A_value;
   L = states_values.size();
   states = states_values;

   priorD.resize(L);
   for(int l = 0; l < L; l++)
     priorD[l].resize(states[l]);
 }

set_prior(const int l, const int s, const TensorD &probs){
  priorD[l][s] = probs;
  if(probs.size() != A) {
    std::cerr << "The vector of probabilities has"
              << "the wrong size. And the size is "
              << probs.size() << std::endl;
    return;
  }
}

const TensorD& get_prior(const int l, const int s) const{
  return priorD[l][s];
}
const double& get_prior(const int l, const int s,
                        const int w) const{
  return priorD[l][s][w];
}

int generate_outcome(const int l, const int s){
  std::random_device re;
  std::knuth_b knuth(re());

  TensorD &prob = priorD[l][s];
  std::discrete_distribution<>
    distribution(prob.begin(), prob.end());
  return distribution(knuth);
}

std::ostream &
operator<<(std::ostream &stream, prior &mu){

  stream << "The prior function is:\n\n";

  for(int l = 0; l < mu.L; l++){
```

```
        int sMax = mu.states[l];
        for(int s = 0; s < sMax; ++s){
          for(int a = 0; a < mu.A; ++a)
            stream << mu.get_prior(l,s,a) << ' ';
          stream << std::endl;
        }
      stream << std::endl;
      }
      return stream;
    }
```

2.1.2 Transition function

The transition function class sets the discrete probability distribution over states of the next partition $l+1$ for which next state will be c given a tuple state-action (l, s, a) of partition l, state s and action a.

Its protected components are the following:

- The protected member function *size_t L* correspond to the number of partitions of the states set $\mathcal{S}$.
- The protected member function *size_t A* correspond to the number of possible actions of the set $\mathcal{A}$.
- The protected member function *TensorI states* correspond to the vector of states of every partition of $\mathcal{S}$.
- The protected member function *Tensor4D tr* correspond to a three dimensional tensor of doubles that keeps the probability to move to each state of the next partition given a state (partition and number of state (l, s)) and an action (a).

```
protected:
        size_t L;
        size_t A;
        TensorI states;
        Tensor4D tr;
```

Then it has as public constructors:

- The default constructor.
- The constructor that initializes the prior class with given state values and action size.

```
transitions() = default;

transitions(const TensorI &state_values, const size_t A_value){
  A = A_value;
  L = state_values.size();
  states = state_values;
  tr.resize(L-1);
}
```

And finally as public methods and operators:

- Method to initialize the transitions with given state values and action size.
- Method to set the probability distribution for a specific state-action tuple over states.
- Method to set the probability for a specific state-action-action tuple.
- Method to get the probability for a specific state-action-action tuple.
- Method to generate a random next state based on the transition probabilities for a given state and action
- Stream operator to print the transition function

```
void init_transitions(const TensorI &state_values,
                      const size_t A_value){
  A = A_value;
  L = state_values.size();
  states = state_values;
  tr.resize(L-1);
  for(int l = 0; l < L-1; ++l){
    tr[l].resize(states[l]);
    for(int s = 0; s < states[l]; ++s)
      tr[l][s] = Tensor2D(A, TensorD(states[l+1]));
  }
}

void set_transitions(const int l,  const int s, const int a,
                     const TensorD &probs){
   tr[l][s][a] = probs;
}

void set_transitions(const int l, const int s, const int a,
                     const int x, const double p){
  tr[l][s][a][x] = p;
}
```

```
TensorD& get_transitions(const int l, const int s, const int a){
  return tr[l][s][a];
}

int next_state(const int l, const int origin, const int action){
  if(l >= L || origin > states[l]){
    std::cerr<< "The input for the transitions ";
             << "next_state function is wrong.\n";
    return 0;
  }
  std::random_device re;
  std::knuth_b knuth(re());

  TensorD &prob = tr[l][origin][action];
  std::discrete_distribution<>
       distribution(prob.begin(), prob.end());
  return distribution(knuth);
}

std::ostream &
operator<<(std::ostream &stream, transitions &trans){
   stream << "The probability transitions are:\n\n";
   size_t L = trans.L;
   size_t nStates0;
   for(int l = 0; l < L-1; l++){
      nStates0 = trans.states[l];
      for(int s0 = 0; s0 < nStates0; s0++){
        for(int action = 0; action < trans.A; action++){
         for(auto const& el: trans.get_transitions(l,s0,action))
           stream << el << ' ';
         stream << std::endl;
        }
      }
      stream << std::endl;
   }
  return stream;
}
```

2.1.3 Rewards container

First of all we define an enum class called TypeReward to differ between receiver's and sender's reward.

```
enum class TypeReward
```

```
    {
        Sender,
        Receiver
    };
```

Then the rewards class. Computed as a template class to distinguish between Sender and Receiver. It sets the reward from every tuple state-outcome-action (l, s, a, o). It is defined as:

```
    template<TypeReward R>
    class rewards;
```

Its protected components are the following:

- The protected member function *size_t L* correspond to the number of partitions of the states set $\mathcal{S}$.
- The protected member function *size_t A* correspond to the number of possible actions of the set $\mathcal{A}$.
- The protected member function *TensorI states* correspond to the vector of states of every partition of $\mathcal{S}$.
- The protected member function *Tensor4D rw* correspond to a four dimensional tensor of doubles that keeps the rewards values for every state-action-outcome.

```
  protected:
        size_t L;
        size_t A;
        TensorI states;
        Tensor4D rw;
    };
```

Then it has as public constructors:

- The default constructor.
- The constructor that initializes the rewards class with given state values and action size.

```
rewards() = default;

template<TypeReward R>
rewards(const TensorI &states_values, const int A_value)
{
  A = A_value;
```

```
    L = states_values.size();
    states = states_values;
    rw = Tensor4D(L);
    for(int i = 0; i < states.size(); i++)
      rw[i] = Tensor3D(states[i], Tensor2D (A, TensorD(A)));
  }
```

And finally as public methods and operators:

- Method that initializes the rewards class with given state values and action size
- Method to set the reward value for a specific state, action, and outcome
- Method to get the rewards for a specific state and outcome
- Method to get the reward value for a specific state, action, and outcome
- Stream operator to print the rewards values

```
  template<TypeReward R>
  void init_rewards(const TensorI &states_values, const int A_value){
    A = A_value;
    L = states_values.size();
    states = states_values;
    rw = Tensor4D(states.size());
    for(int i = 0; i < states.size(); i++)
      rw[i] = Tensor3D(states[i], Tensor2D (A, TensorD(A)));
  }

  template<TypeReward R>
  void set_rewards(const int l, const int state, const int outcome,
                   const TensorD& rewards){
    if(l > L || state > states[l] || outcome > A){
      std::cerr << "The inputs given to the function are incorrect.\n";
      return;
    }
    rw[l][state][outcome] = rewards;
  }

  template<TypeReward R>
  const double& get_reward(const int l, const int state,
                           const int outcome, const int action) const
  {
    if(l > L || state > states[l] || outcome > A || action > A)
      std::cerr << "The inputs given to the function are incorrect.\n";
    return rw[l][state][outcome][action];
```

```
}

template<TypeReward R>
const TensorD& get_rewards(const int l, const int state,
                           const int outcome) const {
  if(l > L || state > states[l] || outcome > A)
    std::cerr << "The␣inputs␣given␣to␣the␣function␣are␣incorrect.\n";
  return rw[l][state][outcome];
}

template<TypeReward R>
std::ostream &
operator<<(std::ostream &stream, rewards<R> &rewards){
  if(R == TypeReward::Sender)
     stream << "The␣sender␣rewards␣are:\n\n";
  if(R == TypeReward::Receiver)
     stream << "The␣receiver␣rewards␣are:\n\n";

  for(int l = 0; l < rewards.L; ++l){
   for(int s = 0; s < rewards.states[l]; ++s){
      for(int o = 0; o < rewards.A; ++o){
         for(int a = 0; a < rewards.A; ++a){
           stream << rewards.get_reward(l, s, o, a) << "␣";
         }
         stream << "\n";
      }
   }
   stream << std::endl;
  }
  return stream;
}
```

2.2 Opt Opt

This short part is focused on getting the optimal signaling scheme with the OptOpt algorithm. It defines the class *sign_scheme* with a friend function which is suposed to execute the OptOpt.

2.2.1 Signaling scheme

This class is defined to represent the signaling scheme the sender shows to the receiver to maximize his expected rewards.

Its private components are:

- The private member function *size_t L* correspond to the number of partitions of the states set $\mathcal{S}$.
- The private member function *size_t A* correspond to the number of possible actions of the set $\mathcal{A}$.
- The private member function *TensorI states* correspond to the vector of states of every partition of $\mathcal{S}$.
- The private member function *Tensor4D squeme* correspond to a four dimensional tensor of doublesthat keeps the signal probabilities for every state-outcome-action.

```
  private:
    size_t L;
    size_t A;
    TensorI states;
    Tensor4D squeme;
  };
```

Its constructor is:

- Default constructor.

```
  sign_scheme(){};
```

Its methods are:

- Method for initializing the vector with size L.
- Method for getting the value of the recommendation of action given tuple state-outcome.
- Method for getting the recommendation probability vector over the actions.

```
  void init_scheme(const TensorI& states_values,
                                  const size_t A_value){
    L = states_values.size();
    states = states_values;
    A = A_value;

    squeme.resize(L);

    for(int l = 0; l < L; ++l){
      squeme[l].resize(states[l]);
      for(int s = 0; s < states[l]; ++s){
```

```
        squeme[l][s].resize(A);
        for(int k = 0; k < A; ++k)
          squeme[l][s][k].resize(A);
      }
    }
  }

  double get_sign(int l, int s, int o, int a_value){
    return squeme[l][s][o][a_value];
  }

  int recommendation(int l, int s, int outcome){
    std::random_device re;
    std::knuth_b knuth(re());

    TensorD &prob = squeme[l][s][outcome];
    std::discrete_distribution<>
          distribution(prob.begin(), prob.end());
    int rec = distribution(knuth);
    return rec;
  }
```

2.3 Markov Persuasion Process

In this last part I tryed to get all together and use all the constructios seen before.

The parts seen in this part are:

- Enviroment struct.
- SOA class.
- Episode class.
- Estimated derivate classes of transitions, prior and rewards.
- Estimators struct.
- Read and print enviroment functions and print estimators.
- Algorithm 1, Sender-Receivers interaction
- Algorithm 2, Optimistic Persuasive Policy Search

2.3.1 Enviroment struct

This struct collects the enviroment variables to take all the information needed for the algorithms.

```
struct Enviroment {
    size_t L;
    TensorI states;
    size_t A;
    transitions trans;
    rewards<TypeReward::Sender> Srewards;
    rewards<TypeReward::Receiver> Rrewards;
    prior mu;
};
```

2.3.2 SOA

Tuple class formed of a state, an outcome and an action in form of integers.

```
class SOA {
public:
    /* Default constructor */
    SOA() = default;

    /* Constructor that initializes the values */
    SOA(size_t xValue, size_t wValue, size_t aValue):
        x(xValue), w(wValue), a(aValue){};

    /* Methods for setting the state value */
    void setX(size_t value){
        x = value;
    };

    /* Methods for getting the state value */
    const size_t& getX() const{
        return x;
    };

    /* Methods for setting the outcome value */
    void setW(size_t value){
        w = value;
    };

    /*Methods for getting the outcome value*/
    const size_t& getW() const{
        return w;
```

```
    };

    /* Methods for setting the action value */
    void setA(size_t value){
        a = value;
    };

    /* Methods for getting the action value */
    const size_t& getA() const{
        return a;
    };
private:
    size_t x;  // State
    size_t w;  // Outcome
    size_t a;  // Action
};
```

2.3.3 Episode

The episode class collects the tuples SOA of an episode in a vector of SOA.

```
class episode {
public:
    /* Constructor that sets the variable sizes */
    episode(TensorI values_states){
        L = values_states.size();
        states = values_states;
        ep.resize(L);

    };

    /* Methods for setting the SOA value */
    void set_soa(int n, SOA soa){
        ep[n] = soa;
    }

    /* Methods for getting the SOA value */
    const SOA& get_soa(int n) const{
        return ep[n];
    }

    /*
      Method that generates an episode with a given prior
      distribution, signaling scheme and probabilities of
      transitions
```

```
    */
    void generate_episode(prior mu, sign_scheme phi, transitions trans);

    /* Stream operator for the episode class */
    friend std::ostream &
    operator<<(std::ostream &stream, episode &ep);

private:
    size_t L; // Number of partitions of the states
    TensorI states; // Number of states in each partition
    /* Vector that collects the tuples (s, o, a) of the episode */
    std::vector<SOA> ep;
};
```

2.3.4 Derivative classes

This three derived classes from the father classes transitions, prior and rewards are defined to collect the information about the approximation of these function classes from the sender's point of view.

Similarly to the father classes they have a default and another contructor with the same functions. But these add some new initializations about the new private members, that are in the three of them a out of variable and a visits variable. Then a method that counts the visits to the corresponding tuples and another that updates the original members.

The estimated prior class:

```
/* Derived class of the prior for the estimation of the prior */
class est_prior : public prior {
public:
  /* Default constructor */
  est_prior() = default;

  /* Constructor to initialize the estimated prior
     with given state values and action size */
  est_prior(const TensorI &states_values, const size_t A_value);

  /* Function that counts a new visited state-outcome tuple */
  void visited(const int l, const int s, const int o){
    visits[l][s][o]++;
    out_of[l][s]++;
  }

  /* Method that updates the estimated prior function
```

```
    after a new episode is generated */
  void update_prior(const episode &ep);

private:
  Tensor2I out_of; // Counts the states visited
  Tensor3I visits; // Counts the states-outcomes tuples visited
};
```

Then the derived template class of the reward template class for the estimation of the reward, the rewards theorically are sampled from a distribution with a mean value, so it makes sense to estimate it though the mean of several samples. But in our code we just take a fixed value for the rewards given in the data.txt file, so doing the mean of the same values is unnecessary. Anyway the estimation of the reward calculating the mean of the seen values is programmed.

The estimated transitions class:

```
class est_transitions : public transitions {
public:
  /* Default constructor */
  est_transitions() = default;

  /* Constructor to initialize the estimated transitions
     with given state values and action size */
  est_transitions(const TensorI &state_values, const size_t A_value);

  /* Function that counts a new visited state-action-state tuple */
  void visited(const int l, const int s, const int a, const int x)
  {
    visits[l][s][a][x]++;
    out_of[l][s][a]++;
  };

  /* Method that updates the estimated transition function after
     a new episode is generated */
  void update_transitions(const episode &ep);

private:
  Tensor3I out_of; // Counts the state-action tulpes visited
  Tensor4I visits; // Counts the state-action-state tuples visited
};
```

The estimated reward class:

```
template<TypeReward R>
class est_rewards : public rewards<R> {
public:
```

```
  /* Default constructor */
  est_rewards() = default;

  /* Constructor to initialize the estimated rewards with given
     state values and action size */
  est_rewards(const TensorI &states_values, const int A_value);

  /* Function that counts a new visited state-outcome-action tuple */
  void visited(const int l, const int s, const int o, const int a){
    visits[l][s][o][a]++;
  }

  /* Method that updates the estimated rewards values after a new
     episode is generated */
  void update_rewards(const episode &ep, const rewards<R> &rw);

private:
  // Counts the visits for the state-outcome-action tuples
  Tensor4I visits;
};
```

2.3.5 Estimators struct

Structure that takes all the estimated function and values by the sender.

```
struct Estimators {
    est_prior estimated_mu;
    est_transitions estimated_trans;
    est_rewards<TypeReward::Sender> estimated_SR;
    est_rewards<TypeReward::Receiver> estimated_RR;
};
```

2.3.6 Read and print enviroment

These two functions just collect the information from the .txt file in the variables in the enviroment structure and print it from it.

```
void read_enviroment(Enviroment &env, const std::string& fileName)
{
/* Defining some references to the variables to avoid writing env. */
  size_t& L = env.L;
  TensorI& states = env.states;
  size_t& A = env.A;
  transitions& trans = env.trans;
  rewards<TypeReward::Sender>& Srewards = env.Srewards;
```

```
rewards<TypeReward::Receiver>& Rrewards = env.Rrewards;
prior& mu = env.mu;

 /* Initializing the .txt stream reader */
 std::ifstream inputFile(fileName);

 if(!inputFile) {
    std::cerr << "The␣file␣couldn't␣be␣opened.\n";
    return;
 }

 /* Reading the number of states */

 std::string textLine;
 std::getline(inputFile, textLine);

 inputFile >> L;
 states.resize(L);

 for(int i = 0; i < L; ++i)
     inputFile >> states[i];

 /* Reading the number of actions */

 for(int i = 0; i < 3; ++i)
    std::getline(inputFile, textLine);

 inputFile >> A;

 for(int i = 0; i < 3; ++i)
    std::getline(inputFile, textLine);

 trans.init_transitions(states, A);

 /* Reading the probability transitions */

 int nStates0, nStatesD;
 for(int l = 0; l < L-1; l++){
    nStates0 = states[l];
    nStatesD = states[l+1];
    for(int s0 = 0; s0 < nStates0; s0++){
      for(int action = 0; action < A; action++){
        double p, sum = 0;
        TensorD probs;
        for(int sD = 0; sD < nStatesD; sD++){
          inputFile >> p;
```

```
            if(p > 1 || p < 0){
              std::cerr << "The␣inputs␣given␣for␣partition␣" << l
              << "␣and␣state␣" << s0 << "␣doesn't␣correspond␣"
              << "to␣a␣probability␣distribution.\n";
              return;
            }
            sum += p;
            probs.push_back(p);
          }
          if(sum != 1){
            std::cerr << "The␣inputs␣given␣for␣partition␣" << l
            << "␣and␣state␣" << s0 << "␣doesn't␣correspond␣to␣a"
            << "␣probability␣distribution.\n";
            return;
          }
        trans.set_transitions(l, s0, action, probs);
        }
      }
  }

/* Reading the rewards of the sender */

for(int i = 0; i < 3; ++i)
  std::getline(inputFile, textLine);

Srewards.init_rewards(states, A);

double r;
for(int l = 0; l < L; ++l){
  for(int s = 0; s < states[l]; ++s){
    for(int o = 0; o < A; ++o){
       TensorD v;
       for(int i = 0; i < A; ++i){
         inputFile >> r;
         if(r > 1 || r < 0){
           std::cerr << "The␣rewards␣are␣not␣in␣the␣required"
                     << "rank␣(inside␣[0,1]).\n";
           return;
         }
         v.push_back(r);
       }
       Srewards.set_rewards(l, s, o, v);
    }
  }
}
```

```
/* Reading the rewards of the receiver */

for(int i = 0; i < 3; ++i)
  std::getline(inputFile, textLine);

Rrewards.init_rewards(states, A);

for(int l = 0; l < L; ++l){
  for(int s = 0; s < states[l]; ++s){
    for(int o = 0; o < A; ++o){
       TensorD v;
       for(int i = 0; i < A; ++i){
         inputFile >> r;
         if(r > 1 || r < 0){
           std::cerr << "The␣rewards␣are␣not␣in␣the"
                     << "required␣rank␣(inside␣[0,1]).\n";
           return;
         }
         v.push_back(r);
       }
       Rrewards.set_rewards(l, s, o, v);
    }
  }
}

/* Reading the prior function, the probability of
   the outcomes given a state */

for(int i = 0; i < 3; ++i)
  std::getline(inputFile, textLine);

mu.init_prior(states, A);

double p;
for(int l = 0; l < L; l++){
  int sMax = states[l];
  for(int s = 0; s < sMax; ++s){
    TensorD ps;
    double sum = 0;
    for(int o = 0; o < A; ++o){
      inputFile >> p;
      sum += p;
      if(p > 1 || p < 0){
        std::cerr << "The␣prior␣function␣is␣incorrect\n";
        return;
      }
```

```
        ps.push_back(p);
      }
      if(sum != 1){
          std::cerr << "The prior function is incorrect\n";
          return;
        }
      mu.set_prior(l,s,ps);
    }
  }

   /* Finalizing the .txt stream reader */

  inputFile.close();
};


/* Function that prints the information about the enviroment */
void print_enviroment(Enviroment& env)
{
  /* Defining some references to the variables to avoid writing env. */
  size_t& L = env.L;
  TensorI& states = env.states;
  size_t& A = env.A;
  transitions& trans = env.trans;
  rewards<TypeReward::Sender>& Srewards = env.Srewards;
  rewards<TypeReward::Receiver>& Rrewards = env.Rrewards;
  prior& mu = env.mu;

  std::cout<< "-------------------------------------------------\n";
  std::cout<< "THIS IS ALL THE INFORMATION ABOUT THE ENVIROMENT\n";
  std::cout<< "-------------------------------------------------\n";

  /* Printing the states */

  std::cout<< "The lenght of the episodes L is : " << L << "\n";
  std::cout<< "And every partition has the following"
           << " numbers of states: ";
  for(int i = 0; i < L; ++i)
      std::cout<< states[i] << ' ';
  std::cout<< std::endl<< std::endl;

  /* Printing the number of actions */

  std::cout<< "The number of actions is : " << A << "\n\n";

  /* Printing the probability transitions */
```

```
  std::cout<< trans << std::endl;

  /* Printing the sender rewards */

  std::cout<< Srewards << std::endl;

  /* Printing the receiver rewards */

  std::cout<< Rrewards << std::endl;

  /* Printing the prior function */

  std::cout<< mu << std::endl;

};

/* Function that prints the Estimators */
void print_estimators(Estimators& est)
{
  /* Defining some references to the variables to avoid writing env. */
  est_prior& estimated_mu = est.estimated_mu;
  est_transitions& estimated_trans = est.estimated_trans;
  est_rewards<TypeReward::Sender>& estimated_SR = est.estimated_SR;
  est_rewards<TypeReward::Receiver>& estimated_RR = est.estimated_RR;

  /* Printing the probability transitions estimation */
  std::cout<< estimated_trans << std::endl;

  /* Printing the sender rewards */
  std::cout<< estimated_SR << std::endl;

  /* Printing the receiver rewards */
  std::cout<< estimated_RR << std::endl;

  /* Printing the prior function */
  std::cout<< estimated_mu << std::endl;
}
```

2.3.7 Algorithm 1, Sender-Receiver's interaction

This function returns an episode generated by the Sender-Receivers interaction that we call Algorithm 1.

```
episode S_R_interaction(Enviroment& env, sign_scheme phi){
```

```
  // Defining some references to the variables to avoid writing env.
  size_t& L = env.L;
  TensorI& states = env.states;
  size_t& A = env.A;
  transitions& trans = env.trans;
  rewards<TypeReward::Sender>& Srewards = env.Srewards;
  rewards<TypeReward::Receiver>& Rrewards = env.Rrewards;
  prior& mu = env.mu;

  // Declaration of the auxiliar variables for the algorithm
  int action, outcome, actual_state = 0;

  episode ep(states);

  /*
    For the partition l from 0 to L-1 starting with x0 it does
    the following:
      -Generates an outcome with the prior mu
      -Generates a signal/reccomendation with the siganling scheme
      -Saves the SOA (state-outcome-action) tuple in the episode
      -Generates the next state with the transition function
  */
  for(int l = 0; l < L; ++l){
    outcome = mu.generate_outcome(l, actual_state);
    action = phi.recommendation(l, actual_state, outcome);
    SOA soa(actual_state, outcome, action);
    ep.set_soa(l, soa);
    if(l != L-1)
      actual_state = trans.next_state(l, actual_state, action);
  }

  return ep;
}
```

This algorithm is also implemented as a method of the episode class called generate episode:

```
void generate_episode(prior mu, sign_scheme phi, transitions trans)
{
    /* Auxiliar variables */
    int actual_state = 0;
    int outcome, action;

    /*
      For the partition l from 0 to L-1 starting with x0 it does
```

```
      the following:
        -Generates an outcome with the prior mu
        -Generates a signal/reccomendation with the siganling scheme
        -Saves the SOA (state-outcome-action) tuple in the episode
        -Generates the next state with the transition function
    */
    for(int l = 0; l < L; ++l){
      outcome = mu.generate_outcome(l, actual_state);
      action = phi.recommendation(l, actual_state, outcome);
      SOA soa(actual_state, outcome, action);
      ep[l] = soa;
      if(l != L-1)
        actual_state = trans.next_state(l, actual_state, action);
    }
}
```

2.3.8 Algorithm 2, Optimistic Persuasive Policy Search

This last code is the uncomplete OPPS algorithm called Algorithm 2. Uncomplete due to the lack of the OptOpt function.

This function returns the estimators of the transition function, the rewards and the prior. Calculated by executing sequentially T times algorithm 1.

```
Estimators OPPS(Enviroment& env, unsigned int T){

  /* Defining some references to the variables */
  size_t& L = env.L;
  TensorI& states = env.states;
  size_t& A = env.A;
  rewards<TypeReward::Sender>& Srewards = env.Srewards;
  rewards<TypeReward::Receiver>& Rrewards = env.Rrewards;
  prior& mu = env.mu;

  /* Initialiting the estimators variables */
  est_prior estimated_mu(states, A);
  est_transitions estimated_trans(states, A);
  est_rewards<TypeReward::Sender> estimated_SR(states, A);
  est_rewards<TypeReward::Receiver> estimated_RR(states, A);

  /* Initialiting the signaling scheme */
  sign_scheme phi;
  phi.init_scheme(states, A);
```

```
  for(int t = 0; t < T; ++t){
    // Executing OptOpt algorithm to get the signaling scheme
    OptOpt(phi);

    episode ep = S_R_interaction(env, phi); // Executing Algorithm 1

    estimated_mu.update_prior(ep); // Update of the estimated prior

    // Update estimation of the transitions function
    estimated_trans.update_transitions(ep);

    // Update senders rewards approximation
    estimated_SR.update_rewards(ep, Srewards);
    // Update receivers rewards approximation
    estimated_RR.update_rewards(ep, Rrewards);
  }

  /* Defining the estimators struct variable to return
     all the estimations */
  Estimators estimators;
  estimators.estimated_mu = estimated_mu;
  estimators.estimated_trans = estimated_trans;
  estimators.estimated_SR = estimated_SR;
  estimators.estimated_RR = estimated_RR;

  return estimators;
}
```

Chapter 3

Test

The only test of the code programmed is in the main.cpp file.

There it can be seen the use of the read enviroment funtion, the print enviroment and the tests of both algorithm 1 and algorithm 2. The data taken to test the structures is extracted from a data.txt file in the /test folder.

To execute the code it is just needed to execute the make command and the test of the parts explained above will be executed.

The code is the following:

```
int main() {

  // Declaring the Markov Persuasion Process variables

  Enviroment env;

  // Defining some references to the variables to avoid writing env.

  size_t& L = env.L;
  TensorI& states = env.states;
  size_t& A = env.A;
  transitions& trans = env.trans;
  rewards<TypeReward::Sender>& Srewards = env.Srewards;
  rewards<TypeReward::Receiver>& Rrewards = env.Rrewards;
  prior& mu = env.mu;

  // Setting the the data file name and the data file source

  std::string source = "./../test/";
```

```
  std::string filename = "data.txt";
  std::string DATA = source + filename;

  // Reading the Markov Persuasion Process variables

  read_enviroment(env, DATA);

  // Printing the Markov Persuasion Process variables

  print_enviroment(env);

  // Algorithm 1 (Sender-Receivers Interaction)
  std::cout<< "-------------------------------------------------\n";
  std::cout<< "TEST OF ALGORITHM 1 (Sender-Receivers Interaction)\n";
  std::cout<< "-------------------------------------------------\n";
  sign_scheme phi;
  phi.init_scheme(states, A);

  episode ep = S_R_interaction(env, phi);

  std::cout<< ep << std::endl << std::endl;

  // Algorithm 2 (Optimistic Persuasive Policy Search (full))

  std::cout<< "-------------------------------------------------\n";
  std::cout<< "TEST OF ALGORITHM 2:"
           << " Optimistic Persuasive Policy Search\n";
  std::cout<< "-------------------------------------------------\n";

// T correspond to the number of iterations of the
// Optimistic Persuasive Policy Search algorithm
  unsigned int T = 1000;

  Estimators est = OPPS(env, T);

  print_estimators(est);

  return 0;
}
```

Bibliography

[1] Francesco Bacchiocchi et al. *Markov Persuasion Processes: Learning to Persuade from Scratch*. 2024. arXiv: 2402.03077 [cs.GT].

[2] Francesco Bacchiocchi et al. *Public Signaling in Bayesian Ad Auctions*. 2022. arXiv: 2201.09728 [cs.GT]. URL: https://arxiv.org/abs/2201.09728.

[3] Emir Kamenica and Matthew Gentzkow. "Bayesian Persuasion". In: *American Economic Review* 101.6 (2011).

[4] Jibang Wu et al. *Sequential Information Design: Markov Persuasion Process and Its Efficient Reinforcement Learning*. 2022.