Interactive Bounded Policy Iteration for solving I-POMDPs in Julia

BY

RICCARDO FICARRA
B.A., Computer Engineering, 2017

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

Defense Committee:

Piotr Gmytrasiewicz, Chair and Advisor
Xinhua Zhang
Giovanni Squillero, Politecnico di Torino

ACKNOWLEDGMENT

I would like to thank professors Piotr Gmytrasiewicz and Giovanni Squillero, my advisors, for giving me counseling, feedback and guidance throughout the development of this thesis, and being always available to address my ideas and doubts. My thanks also go to Iacopo Olivo and the rest of the AI research group.

One huge thanks goes to my family, that gave me this incredible opportunity and supported me fiercely from the moment I decided to enroll in the program. This moment would have never been possible if you weren't there. Special thanks go to my grandparents and my sisters, for always cheering me up in the hardest moments of this year.

One special mention goes to my flatmates, which have become a second family to me and have been invaluable companions in this incredible journey, along with all of the other friends from TOP-UIC and UIC.

RF

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BPI	Bounded Policy Iteration
FSC	Finite State Controller
I-BPI	Interactive Bounded Policy Iteration
I-POMDP	Interactive Partially Observable Markov Decision Process
I-POMDPs	Interactive Partially Observable Markov Decision Processes
POMDP	Partially Observable Markov Decision Process
POMDPs	Partially Observable Markov Decision Processes

SUMMARY

Achieving rational behavior in partially observable multi-agent stochastic environments is becoming a popular field of interest among Artificial Intelligence experts, especially with the growing phenomenon of autonomous vehicles. Intelligent agents must overcome several limitations such as imperfect sensors, other entities interacting with the environment and uncertainty. Solutions to these kind of problems are often obtained through planning, which has extremely high computational costs. In this work, the Julia IPOMDPs framework is used to define one method to solve I-POMDPs called I-BPI, together with its single-agent version, BPI. Both of these algorithms are based on policy iteration, and seek to reduce the complexity of the solution to polynomial. Moreover, these approaches produce policies in form of a finite state machine, which is extremely compact and efficient during execution. This is critical in applications where split-second decisions are needed on devices with limited memory and/or computational power. Exploiting the capabilities of these algorithms, different hierarchies of agent frames are analyzed in the context of the interactive tiger problem, to better understand how different agents interact with each other.

CHAPTER 1

INTRODUCTION

1.1 Previous Work and Motivation

Several teams presented their method of solving I-POMDPs in the past, all using different programming languages and frameworks. The goal of this work is to import one of these techniques, I-BPI [1], in the Julia.IPOMDPs framework [2], so it can be easily compared to other solving methods. As of this date, I-BPI is the first offline method added to the framework, and seeks to be a starting point for other offline and policy iteration based solvers that will be added to the framework in the future. I-POMDPs is a framework to describe multi-agent problems for partially observable stochastic settings defined in 2004 [3] and it is experiencing an increase in attention in the later years. The main challenges of this model are both memory usage and the time complexity, and it has been proven that obtaining exact solutions for I-POMDPs is PSPACE-hard [3]. Some of the techniques developed to solve I-POMDPs are Value iteration [3], and Policy Iteration [4]. Despite various optimizations such as Interactive Particle Filtering [5] and Incremental Pruning [6] that somehow alleviate the computational cost of obtaining optimal solutions, these algorithms still have exponential complexity and scale poorly for large state spaces, higher numbers of agents or deeper nesting levels in the agent hierarchy. Because of this, several approaches have been developed in an effort to make the problem tractable, I-BPI being among these. The algorithm still remains both memory intensive and time con-

suming, making full use of Julia's superior optimization and performance capabilities.

CHAPTER 2

BACKGROUND

2.1 Partially Observable Markov Decision Processes (POMDPs)

Partially Observable Markov Decision Processes (POMDPs) are specified in [7]. A Partially Observable Markov Decision Process (POMDP) can be univocally identified by a tuple $\left\langle S, A, T, R, Z, O \right\rangle$

- S is the set of finite *physical* states that the environment can be in.
- A is the finite set of actions that the agent can take.
- $T : S \times A \to \Pi(S)$ is the state transition function. Given a starting state s and the action a executed by the agent, it returns the probability distribution over the possible result states $s' \in S$.
- $R : S \times A \to \mathbb{R}$ is the reward function. It returns the value of executing action a in state s.
- Z is the finite set of observation that the agent can receive.
- $O : S \times A \to \Pi(O)$ is the observation function. Given an action a and the result state s' after executing said action, it returns the probability distribution over the possible observations $z \in Z$ that the agent can receive.

Partially observable stochastic environments are among the hardest domains to find optimal solutions to, due to the uncertainty on the current state and on the effects of the agents' actions.

Because of partial observability, the agent cannot know for sure which state it is currently in, so it keeps a probability distribution over the possible states. This distribution is called a *belief* or *belief state*, and is used to keep a summary of the previous history. The belief is a *sufficient statistic*, meaning that it contains all needed information about the past and the initial belief of the agent in the world [8] [9].

The standard execution loop for a POMDP is described as follows:

- Action

  The agent executes a certain action, chosen with some kind of criteria.

- Transition

  The state of the world may change or not after the agent action. This step is defined by the transition function described in the POMDP definition.

- Observation

  Depending on the result state and its action, the agent receives an observation based on its observation function. This observation is used to update the agent's belief state through a Bayesian belief update described in Equation 2.1

$$b(s') = O(z|s', a) \sum_{s} b(s) T(s'|a, s) \tag{2.1}$$

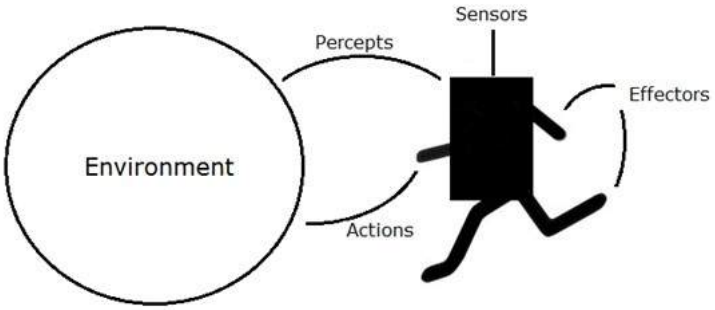


Figure 1: Execution loop

2.1.1 Solving POMDPs

The goal of an agent is to act rationally by maximizing the sum of expected rewards. For finite-horizon problems this is defined as

$$E\Big[\sum_{t=0}^{T-1} r_t\Big]$$

where T is the time horizon. If T is infinite, the infinite sum of non-zero rewards becomes infinite. A discount factor $\gamma \in [0, 1]$ is introduced to keep the sum bounded.

$$E\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big]$$

The resulting agent behavior is called a *policy* $\pi : S \to A$. When a policy represents optimal behavior, it is called an *optimal* policy. The value of an infinite-horizon policy can be computed

recursively as the immediate reward plus the discounted sum of the expected future rewards, as described in Equation 2.2

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s'|s, \pi(s)) V_\pi(s') \tag{2.2}$$

$\pi(s)$ is the action returned by the optimal policy when the agent is in state s.

The overall value of policy π can be obtained as the solution of a $|S|$-dimensional linear system, with one equation for each state. The result is a set of vectors with as many elements as the number of states. Value vectors are linear in the belief state and have higher values at the extremes of the belief simplex, where there is less uncertainty. Because of this the upper surface of the value vector set, which represents the value of the policy, is piecewise linear and convex [7].

There are two types of solvers: offline and online

- Offline

  Offline solvers solve the problem completely before execution. They produce converged optimal policies that can then be used by agents during simulation. They do need to solve the problem completely and without any additional information from execution, but, once a policy is produced, it can be used to take decisions extremely quickly. This is crucial in real world application where thousands of choices must be made each second: one such example is a self-driving car, that needs to decide to brake as quickly as possible whenever it sees an obstacle to not crash into it.

- Online

  Online solvers interleave solving with execution at each step. Since they get more information during execution, they can avoid considering all branches of the policy tree that have not been chosen, dealing with a smaller search space that cuts in both memory and overall time usage by lessening the effects of the Curse of History. However, since they do not have converged policies ready to go at execution time, decision making during execution is noticeably slower and, as such, they are not fit for time-sensitive application.

2.1.1.1 Value Iteration

The most widely known method to solve POMDPs is Value Iteration [7]. It is based on dynamic Bellman backups, and generates all possible policy trees, increasing the time horizon by one step at each iteration. Each step, policy trees branch on actions and observations, growing exponentially. This is referred to as the Curse of History. The expected reward values are back-propagated until the root node, which represents the action to execute at the current timestep. Each policy tree p has an expected reward value for each possible starting physical state $V_p(s)$, which means that the expected value starting from an initial belief $b(s)$ can be computed as $V_p(b) = \sum_s V_p(s)b(s)$. If we consider the set P_t of all policy trees of height t, a rational agent will choose the policy tree that maximizes expected reward as

$$V_t(b) = \max_{p \in P_t} b \cdot V_p(b)$$

Each step, the agent's belief state is updated via a Bayesian belief update. The resulting policy trees can be rearranged in a closed finite-state controller that acts optimally for infinite time horizons [7].

2.1.1.2 Policy Iteration

While the first version of policy iteration has been described in [10], Hansen [11] suggested to use finite state controllers to describe policies in order to simplify and optimize the algorithm. Instead of computing policy trees like value iteration, and then producing a closed finite state controller, PI searches directly in policy space. The produced controllers are already considering an infinite time horizon, and adding new nodes to the controller increases the length (and thus, the complexity) of the plan that the agent will follow.

Policy Iteration alternates between the evaluation and improvement steps until an optimal policy is found. The improvement step includes generation of all possible nodes that can be connected to the already existing nodes in the controller, which are $|A||N|^{|Z|}$.

Techniques such as Incremental Pruning [6], which is considered the best-known technique for exact policy iteration, try to lower the number of nodes by pruning dominated nodes in different stages during the improvement step. The number of nodes generated at each step is reduced, but still $O(|A||N|^{|Z|})$, so, despite pruning, the algorithm slows down considerably as controllers grow in size, especially if the number of observations is not really small.

2.1.2 Finite State Controllers

In this section we describe the finite state controllers that are used to describe policies and their advantages.

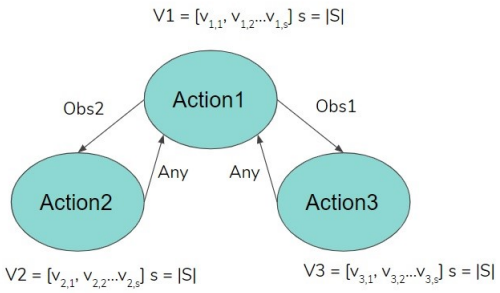


Figure 2: Example of a finite state controller

A node of a finite state controller has three main features:

- Action

  For each node, there is an action that will be executed when that node is reached. This relation is represented as $\alpha : \mathsf{N} \rightarrow \mathsf{A}$ for deterministic finite state machines.

- Edges

  Each edge is bound to one of the possible observations: depending on which observation is received after an action, the corresponding edges will be taken to transition to the next

node (which could be the same as the starting ones, as loops are allowed). This relation is represented as $\beta : N, A, Z \rightarrow N$ for deterministic finite state machines.

- Value Vector

  The value vector, also referred as alpha vector in some cases, is a $|S|$ element vector for POMDPs and a $|S||N_j|$ element vector for I-POMDPs, where $|N_j|$ is the total number of nodes in the controllers in the level immediately in the agent hierarchy. Each element of the value vector associated with node n in state s, represented as $V(n, s)$, represents the converged value of the infinite time-horizon plan that starts at node n and state s and follows the Finite State Controller (FSC) afterwards. The value of the overall FSC is the upper surface of the set of value vectors, and is equivalent to considering the maximum value for each point in the belief space. Elements in a value vector can be visualized as the values of the extreme points in a belief simplex graph like Figure 3. The value of a point on a value vector corresponding to some belief state is computed by performing a dot product operation between the value vector and the belief distribution over states.

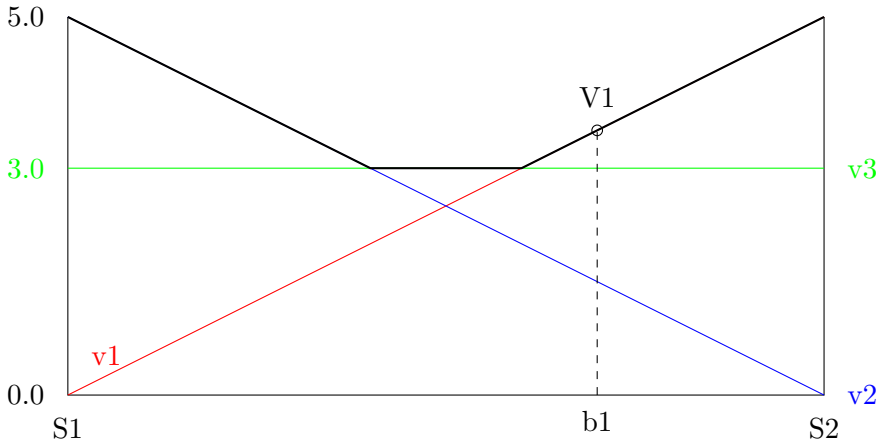


Figure 3: Example of value vectors

Figure 3 illustrates three example value vectors: v1 = $[0, 5]$, v2 = $[5, 0]$, v3 = $[3, 3]$ The value of the overall controller containing nodes with value vectors v1, v2 and v3 is highlighted in black.

V1 is the value corresponding to belief b1 = $[0.3, 0.7]$. Its value is obtained as $\max_{v \in v1,v2,v3}(b1 \cdot v) = 3.5$. The value vector that provides the highest value for a certain belief state will be referred as the "best value vector", and the node to which that value vector refers to will be called the "best node".

Nodes in finite state machines represent a partition of the belief space (sometimes referred as *belief range*) for which the optimal behavior is the same [1]. The belief range associated to a node corresponds to the interval in the belief simplex for which the node has the highest value of all nodes in the controller.This greatly reduces the computations to maintain and update

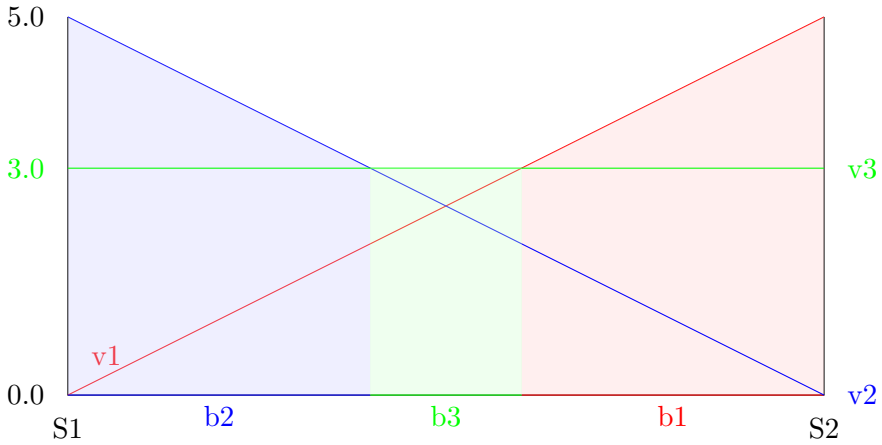


Figure 4: Belief ranges associated with nodes 1,2 and 3

belief states, as we can treat ranges of the belief space as a single node. Edges depend on the observations received after executing the action that the node specifies, and bring the finite state controller to a new state. This is equivalent to a belief update since the agent is moving from a belief range to another, but it is implicitly computed by determining what is the best node for the updated belief range. When using finite state machines, all belief update functions such as Equation 2.1 and Equation 2.5 are replaced by the probability of transitioning to a new node given starting node, actions and observations of the agent $P(n'|n, a, z)$. For deterministic controllers, this probability is 0.0 for all $n' \in N$ except for one node that has probability 1.0. Since this probability is saved into the controller, during execution there is no need to compute a belief update, making agent updates take O(1) time.

BPI and Interactive Bounded Policy Iteration (I-BPI) both use *stochastic finite state machines*

These can represent a policy of equivalent value to a deterministic finite state machine, but using less nodes [12], and, as such, offer a way to lower the computations needed for policy iteration. α and β are modified by returning probability distributions instead of single values.

$$\alpha : N \rightarrow \Pi\{A\}$$

$$\beta : N, A, Z \rightarrow \Delta\{N\}$$

This means that different actions can executed while being in the same node, and that different paths can be taken given the same (action, observation) pair. This makes behavior analysis of the resulting finite state machines complex, requiring techniques such as those described in Chapter 7

2.1.3 Incremental Pruning

Incremental pruning [6] is considered the best technique for exact Policy Iteration. It interleaves pruning with the generation of the new nodes multiple times, removing dominating nodes multiple times to keep the number as limited as possible. Partial nodes with only one observation edge are generated for each action, observation and node in the controller. Their value is computed using Equation 2.3 and grouped by action and observation into $N_{a,z}$ sets. A first round of pruning is executed among nodes in the same set, removing partial nodes that already represent suboptimal choices for any belief state. The remaining nodes are cross summed over actions and observations, performing pruning each time a new $Z_{a,n}$ set is cross summed. Each iteration the partial nodes gain edges for one observation, producing complete nodes at the end of the cross sum operation which are added to the controller. A final pruning step

is executed on the whole controller to remove nodes that are dominated by the newly added nodes.

$$V_{a,z}(n_a, s) = \frac{1}{|Z|} R(s, a_i) + \gamma \sum_{s'} T(s'|s, a) O(z|s', a) V(n_{best}, s') \quad \forall s \in S \tag{2.3}$$

A node n is *pointwise* dominated by node m if $\forall s \in S, V(n, s) \leq V(m, s)$. From a controller standpoint, the value of n is going to be equal or less than the value of m in any belief state, and, as such, n is never going to be visited, so it can be safely removed, reducing the size of the controller (and, consequently, the number of nodes generated in the next improvement step) while not lowering the value of the controller. If the dominated node has any incoming edges, they are redirected to the dominating node. Because of the definition of pointwise domination, the right hand side of the Equation 2.2 never decreases when $V(n, s)$ is substituted by $V(m, s)$ in any state.

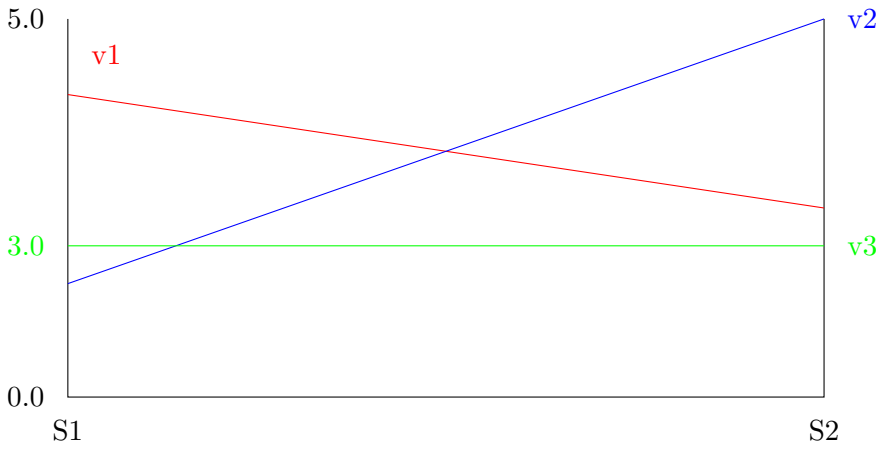


Figure 5: Pointwise domination

V3 is always below v1 in the whole belief space: it is pointwise dominated. V2 does not pointwise dominate v3 because of the small section to the left of the belief space where v3 is above it.

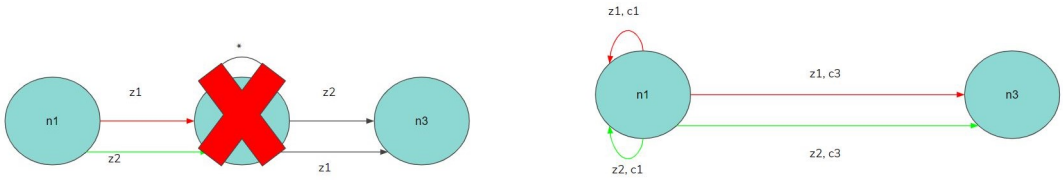
2.1.4 Joint domination and stochastic transitions

I now introduce a different type of domination, called *joint domination* [13], that happens whenever a node is dominated by the combination of a set of nodes. In particular, node n is dominated by $n_1, n_2..n_k$ if $V(n, s) \leq max(V(n_1, s), V(n_2, s)...V(n_k, s))\forall s \in S$.

Another equivalent definition of joint domination is given by the following linear program:

$$\begin{aligned} &\max \delta \\ &\text{where } V(n,s) + \delta \leq \sum_{n_i} c_i V(n_i,s) \forall s \in s \\ &\sum_i c_i = 1 \\ &c_i \geq 0 \ \forall i \text{ in } 1..|N| \end{aligned} \tag{2.4}$$

Whenever the objective function of Equation 2.4 returns a positive value, it means that there is a convex combination $\sum_i c_i \cdot V(n_i, s)$ that pointwise dominates n, which can be removed without lowering overall controller value. The redirection, however, is more complicated, as the *dominating* nodes are more than one. The solution is to use *stochastic* edge transitions, splitting edges between all dominating nodes and using the respective c_is as probabilities of going to each dominating node i.



(a) Joint domination: Pruning

(b) Joint domination: Redirection

Figure 7: Joint domination: finite state machine

N2 is the dominated node. It is removed and the edges are redirected to the dominating nodes, n1 and n3, with corresponding edge transition probabilities c1 and c3.

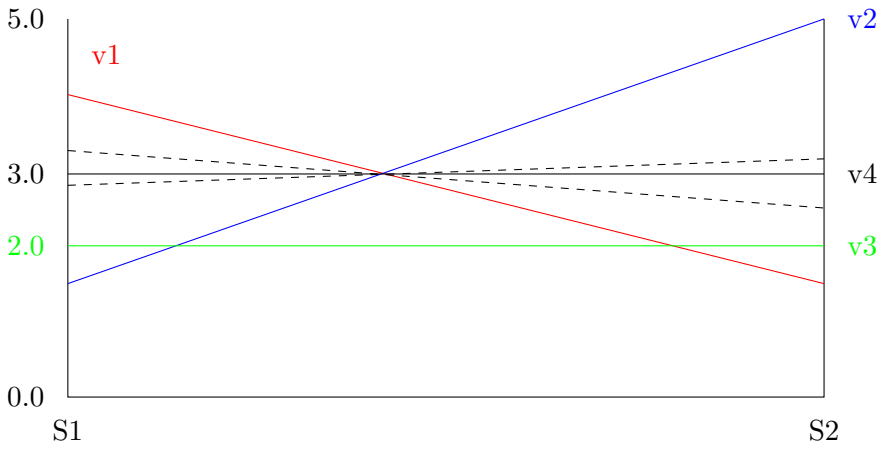


Figure 6: Joint domination

In the figure, v1 and v2 jointly dominate v3. v4 and the dashed vectors are some of the possible convex combinations of v1 and v2, but the LP generates v4, the convex combination parallel to v3 because it seeks to improve all points of the value vector by the same value δ.

2.2 Interactive Partially Observable Markov Decision Processes (I-POMDPs)

Interactive Partially Observable Markov Decision Processes (I-POMDPs) have been specified in [3]. An Interactive Partially Observable Markov Decision Process (I-POMDP) is defined by the tuple $\left\langle IS_i, A_i, T_i, R_i, Z_i, O_i \right\rangle$

- IS_i is the set of interactive states, and is defined as $IS_i = SxM_j$, where M_j is the set of possible models of the other agent j.

  Models can be subintentional or intentional. If an intentional model is an I-POMDP model, it is defined as $m_j \in M_j = \langle b_j, \theta_j \rangle$ where b_j is the belief state of agent j, and $\theta_j = \langle A_j, T_j, R_j, Z_j, O_j \rangle$ is the frame of agent j.

- $A_i = AxA_j$ is the joint set of actions of all agents
- T_i is the transition model. It takes into consideration actions from all agents to return the distribution over result states. This definition assumes *Model Non-Manipulability (MNM)*, which means that an agent cannot change the model (and, consequently, the belief state) of another agent directly through an action. All interaction between agents has to be done via observations.
- R_i is the reward function. It takes into consideration the actions of the other agent to compute the overall rewards. By tweaking R_i we can cause different kinds of behavior in the agent that will be discussed in 2.2.2
- Z_i is the set of observations, defined as in POMDPs.
- O_i is the observation function. It takes into consideration the actions of the other agent to return the distribution over observations received. This definition assumes *Model Non-Observability*, meaning an agent cannot observe the model of another agent directly. This means that agents must keep track of both the belief state and the possible frame of the other agents only using information obtained through observations.

2.2.1 Belief updates

Belief in I-POMDPs are defined as a probability distribution over interactive states $b_i = \Pi(IS_i)$, and are usually referred to as interactive beliefs. Agent i keeps a probability distribution over the physical states S and the models of the other agents M_j. These models are infinite in number, and the beliefs of the other agent b_j can also be interactive, leading to potentially

infinite nesting of beliefs. The formula to compute a belief update for a I-POMDP is the following:

$$\begin{aligned} b'_i(is'_i) &= Pr(is'_i|a_i, z_i, b_i) \\ &= \sum_{is_i} b_i(is_i) \sum_{a_j} P(a_j|m_j) T_i(s'|s, a_i, a_j) x \times \\ & O_i(z_i|s', a_i, a_j) \sum_{z_j} O_j(z_j|s', a_j, a_i) P(b'_j|b_j, a_j, z_j) \end{aligned} \tag{2.5}$$

All of the transition and observation functions in Equation 2.5 are present in their multi-agent versions described in 2.2. $P(a_j|\theta_j)$ is the probability of agent j executing action a_j given its model $m_j = \langle b_j, \theta_j \rangle$, where b_j is j's belief and θ_j is j's frame.

The last factor $P(b'_j|b_j, a_j, z_j)$ is the probability that, after executing action a_j and receiving observation z_j, agent j's belief will become b'_j. Agent j's belief is included in is'_i.

2.2.2 Frame types

Depending on how the reward function is defined, different behaviors can arise. They can be summarized by three main types

- Neutral When the reward of agent i does not depend on the reward of other agents, it will consider it when taking decisions as an additional influence on the environment.
- Cooperative When the reward of agent j impacts positively on i's reward, i will take decisions that help J, as both of the agents being successfull will yield a greater reward. This models very well cooperative games and coordination problems in general.

- Competitive When the reward of agent j impacts negatively on i's reward,i will take decisions that put i at a disadvantage, as causing j to have poor performance will increase i's reward. This kind of frame models well adversary games.

2.2.3 Finitely Nested I-POMDPs

To avoid infinite nesting in the agents belief which make the problem unsolvable, finitely nested I-POMDPs are defined. They consist in building a hierarchy of interactive agents, and then having a non-interactive agent (that can be intentional or subintentional) in its lowest level. This breaks the infinite nesting of beliefs and makes the problem tractable. The height of the hierarchy is referred as the *strategy level*. Agents with a higher strategy level are able to model other agents at a deeper level and obtaining higher value.

2.2.4 Solving I-POMDPs

With the introduction of other agents with multiple possible frames and the need for interactive belief, the size of the interactive state space grows exponentially with the number of states and models. This is known as the *Curse of Dimensionality* [1]

2.2.4.1 Value Iteration

Value Iteration has been proven to converge for I-POMDPs as well [3], and works by constructing a hierarchy of agents in which the agent at level l levels all of the other agents at level $l-1$. This recursion ends at level $l=0$, as agents of level 0 are modeled as POMDPs. Because of this structure, solving an I-POMDP of level l and with M possible models is equivalent to solving $O(M^l)$ POMDPs. This means that solving I-POMDPs is PSPACE-hard for finite time horizons, and undecidable for infinite time horizons [3].

2.2.4.2 Policy Iteration

Policy iteration has been expanded to I-POMDPs in [1], but the extremely high number of nodes generated at each step makes the solution intractable. Because of this, algorithms like I-BPI has been developed in order to avoid generating all possible nodes.

CHAPTER 3

BOUNDED POLICY ITERATION

Bounded Policy Iteration (BPI) is a technique described in [12] and aims to solve POMDPs efficiently. It represents policies as finite state controllers, as suggested by Hansen [11], but, instead of relying on generating all possible nodes that can be connected to the nodes already present in the controller like Incremental Pruning [6], uses linear programming to redirect edges in order to obtain better FSCs. Similarly to gradient ascent [14] [15] [16], this form of policy improvement is prone to get stuck in local optima. Because of this, whenever improvement is not possible, some nodes are added to the controller in order to improve some reachable belief states and push out of the local optima. Whenever the escape fails to add any node, the controller is optimal. BPI can be divided in three steps: evaluation, improvement, and escape from local optima.

3.1 Policy Evaluation

This step computes value vectors for all nodes in the current FSC by solving the linear system defined by equation

$$
\begin{aligned}
V(n,s) = & \\
& \sum_{a} P(a|n)\Big\{R(s,a) + \\
& \gamma \sum_{s'} T(s'|s,a) \sum_{z} O(z|s',a) \sum_{n'_{a,z}} P(n'|n,a,z) V(n',s')\Big\} \\
& n \in N, s, s' \in S, z \in Z
\end{aligned}
\tag{3.1}
$$

The variables of this $|N||S|$ system are $V(n,s)$ and $V(n,s')$. The linear system returns value vectors for all nodes. Each element of a value vector is the converged value for an infinite time horizon of the plan that starts at that node and follows the controller afterwards.

$P(a|n)$ and $P(n'|n,a,z)$ are obtained from the probability distributions returned by $\alpha(n)$ and $\beta(n,a,z)$. $n'_{a,z}$ are the nodes pointed by edges of node n corresponding to action a and observation z. V(n', s') is the value vector of node n' corresponding to state s'. This equation is defined for each state in the state space and for each node in the controller.

3.2 Policy Improvement

This step tries to improve a node by finding a set of stochastic outgoing edges that maximize improvement at each state. This is obtained by solving the linear program defined by Equation 3.2 for each node in the controller, until a positive δ is found. This process improves the FSC without adding new nodes.

maximize δ

subject to

$$V(n,s) + \delta \leq \sum_{a \in A} \left\{ c_a R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \sum_{z \in Z} O(z|s',a) \sum_{n' \in N} c_{a,n',z} V(n',s') \right\} \; \forall s \in S$$

$$\sum_{a \in A} ca = 1.0$$

$$\sum_{n' \in N} c_{a,n',z} = c_a \quad \forall a \in A, z \in Z \tag{3.2}$$

The variables are c_a, $c_{a,n',z}$ and δ, for a total of $|A| + |A||N||Z| + 1$ variables.

This linear program has $|S|$ constraints (one for each state).

After solving the LP, if $\delta > 0$, c_a are set as action probabilities $P(a|n)$, and $c_{a,n',z}$ are renormalized by dividing them by their corresponding c_a and set as edge probabilities $P(n'|a,n,z)$ of the current node n. The newly found action and transition probabilities maximize improvement δ at each state. This corresponds to a rigid upwards translation of the modified value vector, as all elements are increased by the same amount δ. This means that the optimizer does not make any compromise when trying to obtain higher values. This effect can be reduced by maximizing improvement over occupancy frequency [12] [1], in order to improve more states that are more likely to occur. Some approaches [13] allow slight decreases in value at some states in order to obtain higher improvements at relevant states.

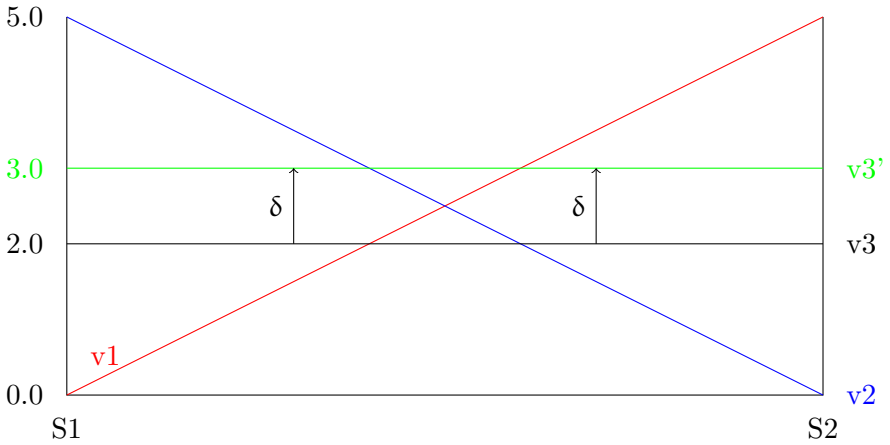


Figure 8: Example of value improvement

3.3 Escape from Local Optima

Whenever the improvement step fails for all nodes, the algorithm is stuck in some kind of optima. To escape it, it is necessary to add new nodes that improve the value of the controller at some points of the belief space. Tangent beliefs offer a simple and quick way to escape such optima.

3.3.1 Local optima and Tangent Beliefs

A controller has reached a local optimum when its value function is tangent to the backed up value function that a traditional DP backup would have generated. This means that the tangent belief states are bottlenecks for further improvement, and that, if it was possible to improve the value at these belief points, it would be possible to escape local optima [12].

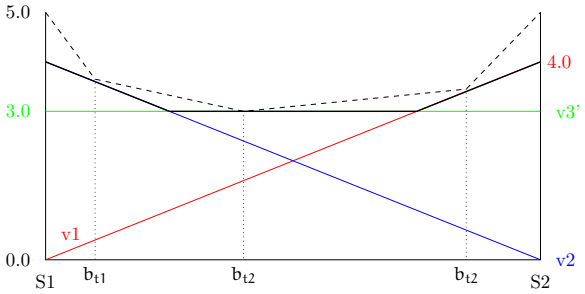


Figure 9: Example of local optima

3.3.2 Escape Technique

Tangent belief states b_t are easily obtainable as the dual solution of Equation 3.2, since most linear programming solvers return both primal and dual solutions. If we could improve the value of the controller at tangent belief points, we would break out of local optima [12]. One-step lookahead is performed from each of these tangent belief points by performing a Bayesian belief update using Equation 2.1 for each possible action and observation, returning a set of reachable belief points.

We add to the controller the nodes corresponding to the value vectors of the backed up value function that improve value in the reachable belief points. The node that maximizes value at a chosen reachable belief b_r is generated as follows: For each action a a new node n_a is created. For each observation pair, we compute the belief b'_r obtained by a Bayesian update starting from b_r after executing action a and receiving observation z. If we add an edge associated with observation z that leads to the best node n_{best} in the FSC for belief b'_r, its contribution

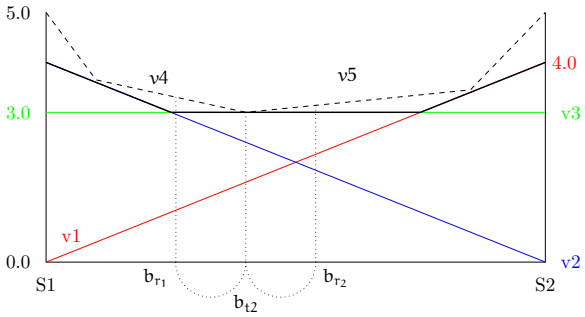


Figure 10: Example of escape from optima

to the new node's value vector will be maximum for the given (a,z) pair and is computed with Equation 2.3

Since all contributions to the value vector of n_a were maximum for their assigned (a,z) pair, each n_a will have the maximum value vector for action a. The node that maximizes value for belief r_b is then chosen as $\mathtt{argmax}_{n_a}\left(r_b \cdot V(n_a)\right)$ and added to the FSC.

This procedure executes $|A||Z|$ belief updates for each of the $|N|$ tangent beliefs returned by the LP to obtain reachable beliefs, and, for each reachable belief, executes $|A||Z|$ belief updates to obtain result beliefs and a $|S|$ dot product $|N|$ times to find the best node in the FSC Belief updates are linear in $|S|$, so the whole process is quadratic in $|S|$, $|N|$, $|A|$ and $|Z|$.

Figure 10 shows an example of the escape process: First, the tangent belief points are obtained as the dual of the LP in the improvement step. Through a belief update, reachable belief points are obtained. If the value of the node in the backed-up controller is greater than the value of the node present in the current controller, that node is generated and added to the controller. This process is repeated for all tangent belief points and all reachable belief points.

3.3.3 Algorithm

Algorithm 1 Bounded Policy Iteration

```
Require: Frame
  Controller = InitializeController(Frame)
  while escape succeeds do
    while improvement succeeds do
      EvaluateController(Controller)
      TangentBeliefs = ImproveController(Controller)
    end while
    EscapeFromOptima(Controller, TangentBeliefs)
  end while
  return Controller
```

InitializeController creates a finite state machine with only one node. This node contains a random action $\mathfrak{a} \in \mathsf{A}$ and all observation edges lead back to itself. This policy represents executing the same action forever, which is usually not a good policy, but it will be the starting point for the algorithm.

EvaluateControllers computes coefficients for for the linear system described in 3.1, solves the system and sets the results as value vectors of the corresponding nodes.

ImproveController defines the LP described in Equation 3.2 using the JuMP Julia interface for CPLEX and solves it for each node. If the linear program returns a positive improvement for a node, the resulting variables are renormalized to sum up to 1 and set as corresponding action and edge transition probabilities in the improved node. If improvement fails for all nodes, the

subroutine returns the dual solutions of all the solved LPs as TangentBeliefs.

EscapeFromOptima adds new nodes to the controller to try and escape local optima as detailed in 3.3.2 using the TangentBeliefs produced by the improvement step.

CHAPTER 4

INTERACTIVE BOUNDED POLICY ITERATION

I-BPI is the extension of BPI for multi-agent problems.

It consists on developing a hierarchy of finite state controllers to represent behavior of all agent frames at all levels. Beliefs become interactive: not only an agent keeps track of the physical state, but it also considers all possible models of the other agent. Value vectors of nodes in controllers of level $l > 0$ have $|S||N_{l-1}|$ elements. This means that each element of a value vector is the value corresponding to a certain physical state, and to the lower level agent being in some node in one of the lower level controllers. Nodes in level 0 controllers are equal to single-agent controllers developed by BPI. The algorithm iteratively recurs bottom-up in the agent hierarchy and applies the three basic steps of BPI, slightly modified to include nodes of the lower level agent, as described in section 4.1. While this is slower than fully developing all controllers at each level before moving to a higher level, it has better any-time property, meaning it can be interrupted and still yield a good (but not converged) controller for the highest level agent.

4.0.1 Algorithm

4.1 Modified functions for multi-agent problems

Policy evaluation and improvement from BPI are modified to include the nodes of the lower level controller to consider the behavior of the other agent. Single agent functions are replaced by the multi-agent ones described in 2.2

Algorithm 2 Interactive Bounded Policy Iteration

```
Require: FrameHierarchy[MaxLevel]
  ControllerHierarchy[MaxLevel]
  for level in FrameHierarchy do
    Controller = InitializeController(FrameHierarchy[level])
  end for
  while at least one level successfully improves or escapes do
    EvaluateAndImprove(ControllerHierarchy[MaxLevel])
  end while
```

Algorithm 3 EvaluateAndImprove

```
Require: ControllerHierarchy[MaxLevel], level
  if level > 0 then
    EvaluateAndImprove(ControllerHierarchy, level -1)
  end if
  Controller = ControllerHierarchy[level]
  if level == 0 then
    EvaluateController(Controller)
    TangentBelief = ImproveController(Controller)
    if improvement failed then
      EscapeFromOptima(Controller, TangentBelief)
    end if
  else
    LowerLevelController = ControllerHierarchy[level-1]
    EvaluateControllerInteractive(Controller, LowerLevelController)
    TangentBeliefs = ImproveControllerInteractive(Controller, LowerLevelController)
    if improvement failed then
      EscapeFromOptimaInteractive(Controller, LowerLevelController, TangentBeliefs)
    end if
  end if
```

4.1.1 Policy Evaluation

The linear system for evaluation is modified as follows:

$$
\begin{aligned}
V(n_i, n_j, s) = &\sum_{a_i} P(a_i|n_i) \sum_{a_j} P(a_i|n_j) \Big\{ R(s, a_i, a_j) + \\
&\gamma \sum_{s'} T(s'|s, a_i, a_j) \sum_{z_i} O_i(z_i|s', a_i, a_j) \sum_{z_j} O_j(z_j|s', a_j, a_i) \\
&\times \sum_{n_j'} P(n_j'|n_j, a_j, z_j) \sum_{n_i'} P(n_i'|n_i, a_i, z_i) V(n_i', n_j', s') \Big\} \\
&\forall \quad n_i, n_i' \in N_i, \quad n_j, n_j' \in N_j \quad s, s' \in S, \quad z_i \in Z_i \quad z_j \in Z_j
\end{aligned}
\tag{4.1}
$$

The system is modified to keep into consideration the possible nodes (and, consequently, belief ranges) in which the other agent might be in, and, from them, which action it is going to choose. This system has $|N_j||S|$ variables and scales quadratically with the number of nodes in the lower level. This means that keeping the number of nodes and frames in the lower level as small as possible is crucial. $P(a_i|n_i)$, $P(a_j|n_j)$, $P(n_i'|n_i, a_i, z_i)$ and $P(n_j'|n_j, a_j, z_j)$ are obtained from the α and β function of the nodes. R, O_i, O_j and T are specified in the frame, and $V(n_i', n_j', s')$ is contained in the value vector of n_j'.

4.1.2 Policy Improvement

The linear program is defined as Equation 4.2. The formula is modified to keep into consideration the nodes of the other agent, as well as all possible node transitions of the other agent. This means that the resulting node will be optimized for both physical states and possible

models of the other agent. If there are multiple possible frames in the lower level, all of the controllers are considered as one controller with different isolated subcontrollers.

maximize δ

subject to

$$
\begin{aligned}
&V(n_i, n_j, s) + \delta \leq \sum_{a_j \in A_j} P(a_j|n_j) \sum_{a_i \in A_i} \Big\{ c_{a_i} R(s, ai, aj) \\
&+ \gamma \sum_{s' \in S} T(s'|s, ai, aj) \sum_{z_i \in Z_i} O_i(z_i|s', a_i, a_j) \sum_{z_j \in Z_j} O_j(z_j|s', a_j, a_i) \\
&\times \sum_{n'_j \in N_j} P(n'_j|n_j, a_j, z_j) \sum_{n' \in N_i} c_{n'_i, a_i, z_i} V(n'_i, n'_j, s') \Big\}
\end{aligned}
\tag{4.2}
$$

$$\forall s \in S$$

$$\sum_{a_i \in A+i} c_{a_i} = 1.0$$

$$\sum_{n' \in N} c_{n'_i, ai, zi} = c_{a_i} \quad \forall a_i \in A_i, z_i \in Z_i$$

The modified linear program has $|S||Nj|$ constraints and $|A| + |A||N||Z| + 1$ variables.

4.1.3 Escape from Local Optima

Both the belief update formula and the partial node value formula are updated considering the other agent's possible nodes, actions and transitions. The tangent belief points obtained by the dual of Equation 4.2 will be $|S||N_j|$ in size, as they also represent the belief over the other agent's nodes.

$$b_r(s', n_j') = \sum_s \sum_{n_j} b_t(s, n_j) \sum_{a_j} P(a_j|n_j) T(s'|s, a_i, a_j) \\ O_i(z_i|s', a_i, a_j) \sum_{z_j} O_j(z_j|s', a_j, a_i) \sum n_j' P(n_j'|n_j, a_j, z_j) \tag{4.3}$$

$$V_{a_i,z_i}(n_{a_i}, s, n_j) = \frac{1}{|Z|} \sum_{n_j} \sum_{a_j} \Big\{ P(a_j|n_j) R(s, a_i, a_j) \\ + \gamma \sum_{s'} T(s'|s, a_i, a_j) O(z_i|s', a_i, a_j) \qquad \forall s \in S, n_j \in N_j \\ \sum_{z_j} O_j(z_j|s', a_j, a_i) \sum_{n_j'} P(n_j'|n_j, a_j, z_j) V(n_{best}, n_j, s') \Big\} \tag{4.4}$$

CHAPTER 5

JULIA IMPLEMENTATION

5.1 IBPISolver

Julia.IBPISolver acts as a hub for both BPI and IBPI. It sets up the environment for both solvers and provides some extra functionality like simulation and statistics.

5.1.1 IBPISolver object

The IBPISolver object contains parameters that are used throughout the solver such as

- force

  Specifies the position of the action in the action vector that is assigned in the initial node of all controllers. If it set to 0, the action is chosen randomly among all possible actions.

- maxrep

  Specifies the maximum number of iterations that BPI or IBPI are allowed to run. A single iteration of BPI corresponds to evaluation, improvement and escape if needed. A single iteration of I-BPI corresponds to evaluation, improvement and escape if needed of all controllers at all levels from the bottom up. If set to -1 there is no limit on the iterations.

- minval

  Specifies the tolerance used for floating point numbers by Julia. This means that a number $\mathfrak{n}$ is considered as such if its floating point representation N is $\mathfrak{n} - \epsilon \leq \mathsf{N} \leq \mathfrak{n} + \epsilon$

- timeout

  Specifies the maximum amount of time that the algorithm is allowed to run for. After the timeout expires the algorithm completes the current iteration, re-evaluates all controllers and terminates.

- min_improvement

  This parameter is used in the escape step. A node is added to the controller if the difference in value at the reachable belief that we are trying to improve between the node in the controller and the new node is greater than this parameter.

- normalize

  If this parameter is set to true, all action probabilities and outgoing edge probabilities for the same observation are normalized to one, to account for the missing actions/edges that have been skipped because the probabilities were below minval

- maxnodes

  This parameter is an array used to set a maximum number of nodes for each strategy level.

The module contains one global IBPISolver instance called *config*, that is accessed by most functions when parameters are needed.

load_policy and *save_policy* are two utility functions used to read and write IBPIPolicy objects from and to files.

5.1.2 Simulator

This section of the IBPISolver module provides a simple simulator with statistic computation.

The whole simulator is based on the IBPIAgent object, that contains a controller, keeps track of the current node of the controller and of various statistics such as value, visited nodes and an interface *agent_statistics* structure that can be defined to contain problem-specific statistics. In the tiger problem case, this structure keeps track of percentage of correct doors opened and average listens before opening.

IBPISimulate creates two agents, one with the highest level controller called agent i, and one with one of the controllers of the second-highest level, called agent j. The controller can be chosen with a parameter to easily change frames. The standard action $\rightarrow$ transition $\rightarrow$ observation loop is repeated for a number of times specified by the user, while the behavior of the agents are determined by the controllers using the *best_action* and *update_agent* functions. The environment can be simulated by using a POMDP or IPOMDP frame through the *compute_s_prime* and *compute_observation* functions, or specific user-created *Scenarios* can be executed, to study the agents' reaction to certain sequences of events.

The whole history of states, actions and observations is saved so it can be examined later.

5.1.3 Statistics

This section of the IBPISolver keeps track and computes statistics related to the solving process of the POMPDP, such as time, number of nodes and memory usage.

5.2 BPI

This module sets up a controller using a POMDP frame and calls the Bounded Policy Iteration algorithm.

5.2.1 Node

This data structure is the basic component for controllers. It is implemented as an unmutable structure for quicker operations. Each node contains:

- Id A controller-wide unique id equal to the position of the node in the controller node array. It is used to have constant time access to the controller node array given a node.
- actionProb An array of tuples, defining the probability of each action. If an action probability is 0, it is not added to this data strucure to save time and memory.
- edges This data structure is made of nested dictionaries that store, for each action and observation, the set of edges that can be taken with their relative probabilities.
- value This field contains the value vector used throughout the algorithms. It is defined as an array with variable dimensions, so it can be used by both BPI (which uses a 1-dimension array) and I-BPI (which uses a 2-dimension array).

5.2.2 Controller

This data structure contains a single controller corresponding to a POMDP frame. The controller is initialized as a single node with a random action (or a user-specified one if the *force* parameter is not set to 0) among the possible ones, and all observation edges leading back to itself. It also contains other fields such as the POMDP frame, statistics related to the solver

and a boolean flag to mark it as converged, so it can be skipped, saving computation time during execution (this mostly needed during I-BPI, as BPI stops once the controller converges).

5.2.3 evaluate!

The *evaluate!* function produces value vectors for all nodes in a controller as described in 3.1. The coefficients for the linear systems are computed sequentially as both tests and documentation report that it is faster than array operations and, furthermore, it allows some extra optimization by skipping nested loops when one of the factors is zero. This is especially impactful when the problem has very sparse transition and observation functions [word this better].

The system is solved using the linear solver provided by Julia, and the solution vector is split into the value vectors assigned to the associated nodes.

5.2.4 partial_backup!

The *partial_backup!* function tries to improve each node as described in 3.2. The linear programming solver used for all experiments is CPLEX, and it is called through the Julia JuMP interface. This interface allows to quickly change solving backend, and other solvers such as GLPK can be used if there is need for a free-to-use solver. The coefficients for the constraints are computed sequentially for the same reasons specified in the evaluate function. The obtained optimization variables are used as probabilities. Edges and actions with probability below the solver's tolerance are not added to nodes to keep the controllers small and reduce computations. A normalization step is necessary to make sure all probabilities sum up to 1.

5.2.5 escape_from_optima!

The *escape_from_optima!* function is used to escape from local optimas as described in 3.3.2. The coefficients for the belief updates are computed sequentially for the same reasons specified in the evaluate function. All of the reachable beliefs are computed first and duplicates are removed to avoid repeated computations.Potential new nodes are generated and added to the controller if the improvement is above a parameter specified by the user.

5.2.6 BPIPolicy

The BPIPolicy structure is a wrapper for the single Controller structure. It is used for compatibility with the simulator functions.

5.2.7 bpi

The *bpi* function is used to start the BPI algorithm on a BPIPolicy. It uses the *evaluate*, *partial_backup* and *escape_optima* functions to improve the controller until convergence, a set number of iterations or a maximum amount of time set by the user, and collects statistics on time, number of nodes and used memory.

5.3 IBPI

This section sets up the environment for the IBPI algorithm.

5.3.1 InteractiveController structure

IBPI uses the same Node structure as BPI, but has a different controller data structure, InteractiveController. This contains the id, frame, nodes, statistics and converged fields as the single agent controller, but the frame is an I-POMDP instead of a POMDP, and there is an

additional level field to keep track of what is the strategy level of the controller. Note that, while in the literature the lowest level frame (the POMDP) is referred to as level 0, in the Julia implementation is considered as level 1 to facilitate work with Julia's 1-based indexing arrays.

5.3.2 IBPIPolicy

This structure defines the agent hierarchies that are developed by the algorithms. In any level below the highest it is possible to define multiple frames at the same level, which are considered the possible frames for agents at that strategy level. Other fields are the policy name (mainly used for saving and loading the policy to and from files) and the highest strategy level of the policy.

Defining an agent hierarchy is extremely simple: the highest level frame is passed individually as a parameter, followed by as many arrays of frames as the desired lower levels. All controllers will be initialized with a single random node as in BPI and the IBPIPolicy object will be returned, ready to be used by I-BPI.

5.3.3 Evaluation, Improvement and Escape

evaluate!, *partial_backup!* and *escape_optima!* are updated as described in 2.2, with the same implementation details as in 5.2

CHAPTER 6

EXPERIMENTS

6.1 Tiger Game

In the following chapter the tiger game will be described in both its single[7] and multi-agent[3] form. It will be widely used for examples and experiments.

6.1.1 Single-Agent Tiger Game

There are two identical doors in a room. Behind one lies a tiger, and behind the other a monetary prize. The agent must open the door hiding the prize, or face terrible doom by opening the one hiding the tiger. The agent has three choices each turn: to open either the left or right door, or to listen. Listening gives the agent a chance to hear the tiger growling either behind the left or right door, but it's not perfectly accurate and will sometimes hear the growl behind the door that hides the prize. Whenever the agent opens any door the tiger and the prize get re-assigned randomly to a new door.

- States $S = \{\mathsf{TL}, \mathsf{TR}\}$
- Actions $A = \{\mathsf{OL}, \mathsf{OR}, \mathsf{L}\}$
- Transition Function

		Action		
State	Result State	OL	OR	L
TL	TL	0.5	0.5	1.0
	TR	0.5	0.5	0.0
TR	TL	0.5	0.5	0.0
	TR	0.5	0.5	1.0

- Reward Function

	Action		
State	OL	OR	L
TL	-100	10	-1
TR	10	-100	-1

- Observations $\mathsf{Z} = \{\mathsf{GL}, \mathsf{GR}\}$
- Observation Function

Result State	Observation	Action		
		OL	OR	L
TL	GL	0.5	0.5	0.85
	GR	0.5	0.5	0.15
TR	GL	0.5	0.5	0.15
	GR	0.5	0.5	0.85

6.1.2 Multi-Agent Tiger Game

There are two players playing the game at the same time. An agent cannot see the other one, but it can hear a creak when the other agent opens a door, or silence if it spent its turn listening. Similarly to the growls, these observations are informative only if the agent spent its turn listening, as observations received after opening a door are random and, as such, completely uninformative. The tiger and the prize are moved only if at least one of the two agents has opened a door, and after both agents have taken an action in that turn.

- States $S = \{\mathsf{TL}, \mathsf{TR}\}$
- Actions $A = \{\mathsf{OL}, \mathsf{OR}, \mathsf{L}\}$
- Transition Function

(ai, aj)	(OL, *)	(OR, *)	(*, OL)	(*, OR)	(L, L)	(L, L)
State	*	*	*	*	TL	TR
TL	0.5	0.5	0.5	0.5	1.0	0.0
TR	0.5	0.5	0.5	0.5	0.0	1.0

- Reward Function Agent i computes its reward by combining in different ways the rewards both agents would have got independently as described in 6.1.1. Different agent types compute their reward differently as described in 6.2.

- Observations $\mathsf{Z} = \{\mathsf{GL}, \mathsf{GR}\} \times \{\mathsf{CL}, \mathsf{CR}, \mathsf{S}\}$

- Observation Function

Observation	Action				
	OL, *	OR, *	L, OL	L, OR	L, L
CL	1/6	1/6	0.9	0.05	0.05
CR	1/6	1/6	0.05	0.9	0.05
S	1/6	1/6	0.05	0.05	0.9

6.1.2 shows the observation probability of observation related to agent j $\mathsf{O}(\mathsf{a_i}, \mathsf{a_j})$ given $\mathsf{a_i}$ and $\mathsf{a_j}$. The overall observation function is computed as $\mathsf{O}(\mathsf{s'}, \mathsf{a_i}, \mathsf{a_j}) = \mathsf{O}(\mathsf{s'}, \mathsf{ai}) \cdot \mathsf{O}(\mathsf{a_i}, \mathsf{a_j})$

6.2 Environment and setup

In the next chapter we set up some different agent hierarchies and solve them using the Julia implementation of I-BPI. Policies are saved each time an improvement step fails and benchmarked later, to affect solving times as little as possible. All results are averages computed over 1000 simulations of 20 steps each, starting from a randomized initial belief. The default floating point tolerance of CPLEX is $\epsilon = 10^{-6}$. I-BPI is set to use the same tolerance as CPLEX.

Five types of frames will be used:

- Single Agent
  - *Standard*

    This is the single agent POMDP defined in 6.1.1.
  - *Random*

    This is a subintentional model with a single node with equal probability of executing any of the three actions. It is marked as converged from creation so it does not get improved by any of the steps.
- Multi-Agent
  - *Neutral*

    This I-POMDP only considers its own reward while playing and ignores the performance of the other agent, but considers the other agent's actions as they can change

the environment.

$R(s, a_i, a_j) = R(s, ai)$

– *Cooperative*

The reward of the agent is partially additive to the other agents' reward, so the agents are encouraged to cooperate.

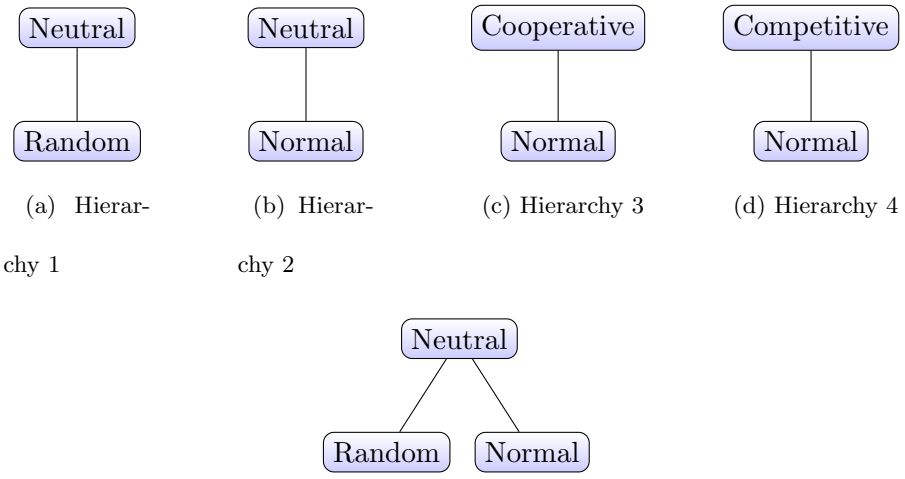
$R(s, a_i, a_j) = R(s, ai) + 0.5R(s, aj)$

– *Competitive*

The reward of the other agent is partially subtracted to the other agents' reward, so the agents are encouraged to compete against each other.

$R(s, a_i, a_j) = R(s, ai) - 0.5R(s, aj)$

6.2.1 Hierarchies



(a) Hierarchy 1

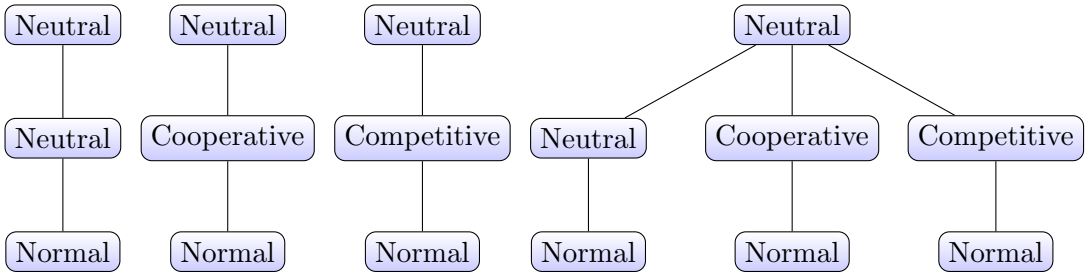
(b) Hierarchy 2

(c) Hierarchy 3

(d) Hierarchy 4

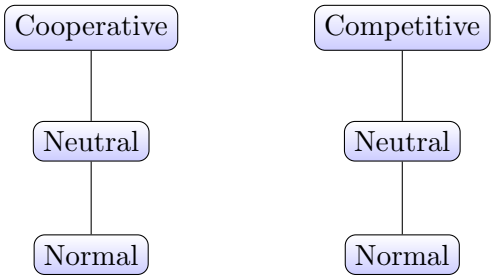
(e) Hierarchy 5

Figure 11: Level 1 hierarchies



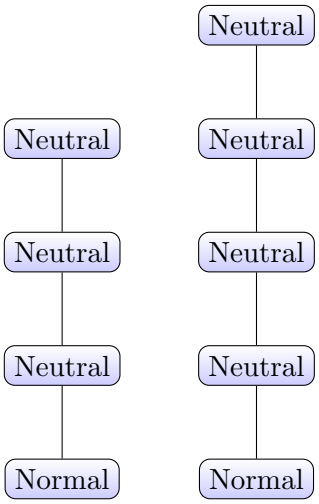
(a) Hierarchy 6 (b) Hierarchy 7 (c) Hierarchy 8 (d) Hierarchy 9

Figure 12: Level 3 hierarchies with neutral agent i



(a) Hierarchy 10 (b) Hierarchy 11

Figure 13: Level 3 hierarchies with cooperative and cooperative agent i



(a) Hierarchy 12

(b) Hierarchy 13

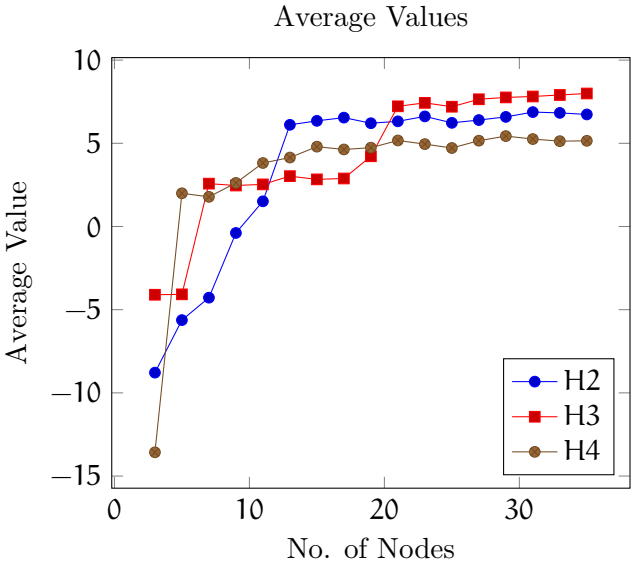
Figure 14: Higher level hierarchies

6.3 Results

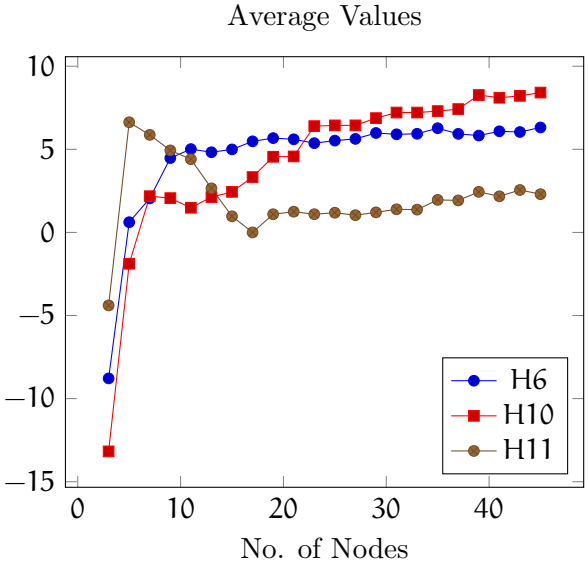
In this sections we analyze the results obtained by developing different agent hierarchies with I-BPI.

Level	Hierarchy	Average Value	Time(s)	Nodes	Memory (MB)
1	1	-1.239	7.268	12	2.148
	2	6.393	26.65	25	19.578
	3	7.198	31.93	25	19.561
	4	4.717	25.582	25	19.547
	5	6.850	70.88	35	45.957
2	6	6.270	293.55	35	204.948
	7	7.122	377.68	35	224.061
	8	5.620	268.09	35	215.700
	9	7.098	2866.187	40	821.069
3	12	6.267	884.574	34	283.661
4	13	6.067	1712.46	45	601.273

The next graphs show statistics computed for all policies at various points in time during the algorithm. Snapshots of the policy are taken everytime an improvement step fails, just before the *escape_from_optima!* routine is called. The last snapshot is the controller returned after the algorithm stops. For all hierarchies with a single controller in the second highest level, the simulated agent j is the same as the modeled agent j. This yields higher average value as the model and the true simulated agent coincide.



(a) Level 2



(b) Level 3

Figure 15: Average Value

The graphs represent the average simulation performance of the finite state machines at different snapshots taken throughout the development of the finite state controller. At first the performance is poor, but, as more nodes are added, the finite state controller is able to represent more complex plans and improve performance until the algorithm converges and stops. Higher level controllers need more nodes to converge as the plans to react to a more sophisticated lower level agent are more complex. Cooperative agents are worse in the beginning, as the small controllers cannot cause correct cooperative behavior with the other agent, but achieve higher average scores once they become complex enough to cooperate well with the other agent. Competitive agents, on the other hand, have higher starting value because the other agent starts from a simple policy as well, but, after agent j develops completely, the average value is lower than the cooperative or neutral counterpart. Neutral agents have average values in between those of competitive and cooperative.

The performance of agents of different strategy levels is comparable: this happens because in these experiments the agent j used in simulation is the same as the corresponding agent j modeled by agent i. This means that agent i modeled j perfectly and reacts near-optimally to it.

6.3.1 Time and memory usage

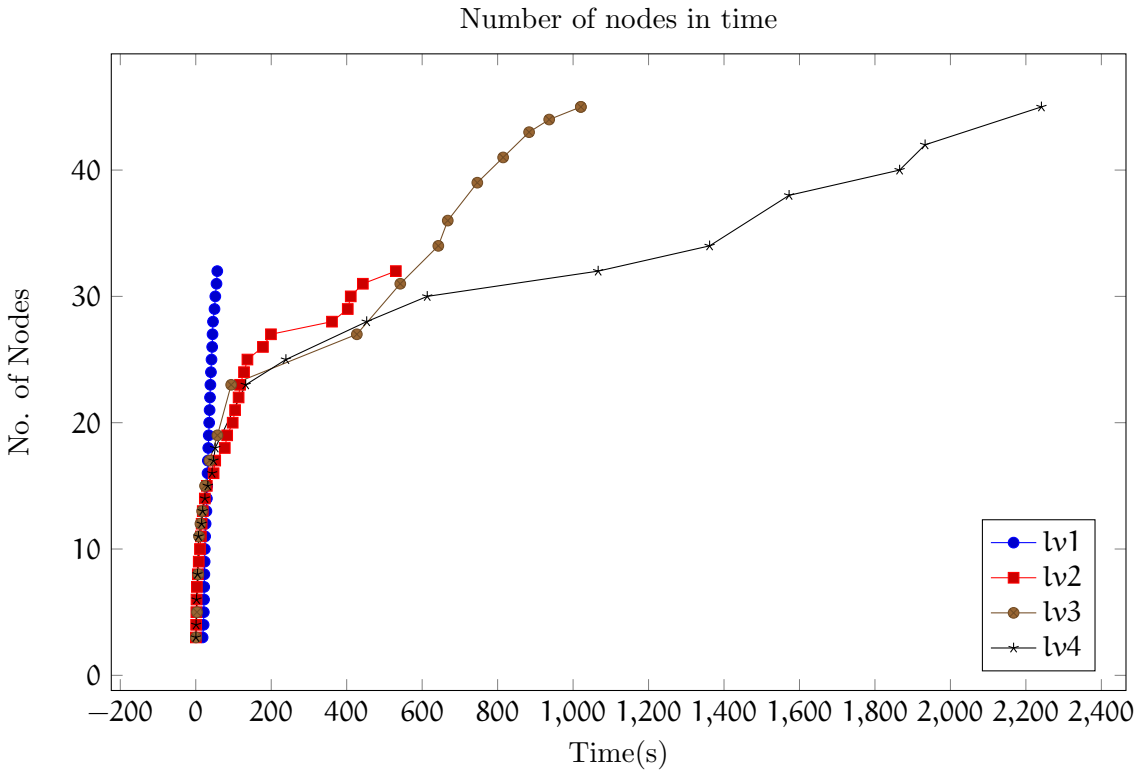


Figure 16: Node growth in time

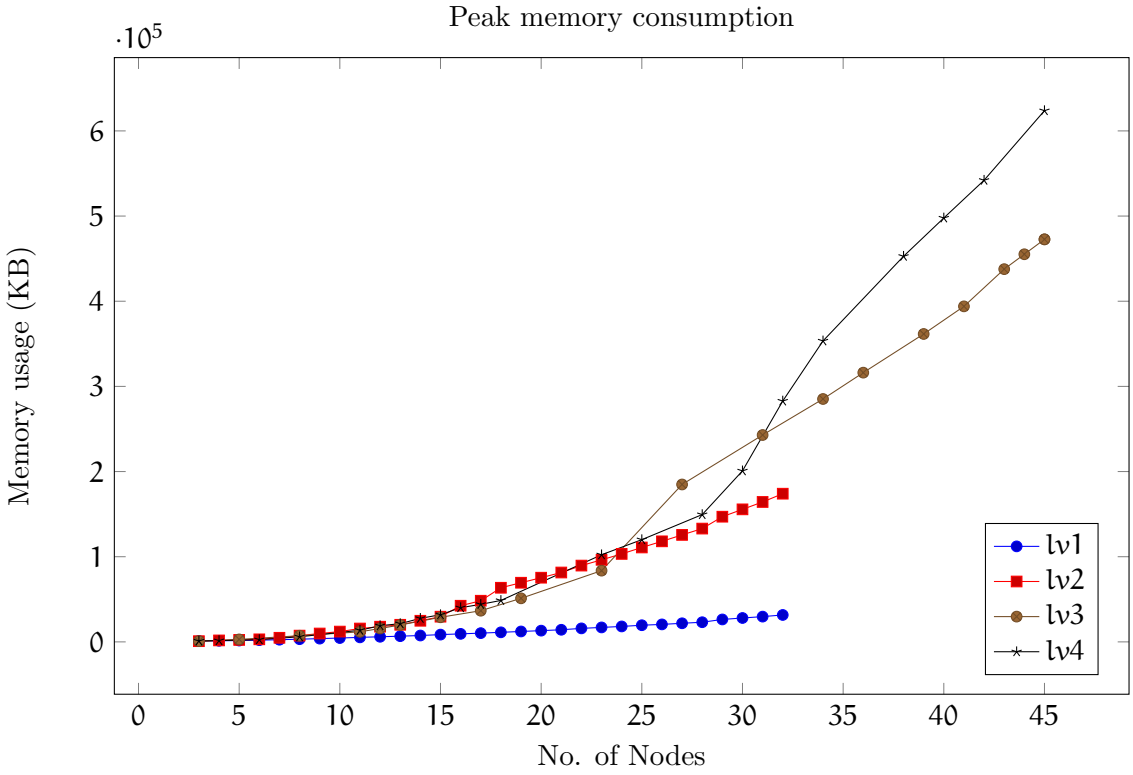


Figure 17: Memory usage in time

As the number of nodes in the controller grow, the memory needed to perform the improvement step increases, as there are more variables in the linear programming problem described in Equation 4.2. Since lower level controllers grow concurrently with the top-level one, the interactive belief space increases in size too, requiring more constraints to be added to the LP and slowing down all other operations that consider lower level agents. This is why it is crucial to keep the size of the controllers as small as possible while still providing good values.

6.3.2 Agent identification

This section analyzes the benefits of having multiple possible agent types in the lower levels of the hierarchies. For hierarchies with more than one frame in the second highest level, the type of the simulated agent j is specified next to the hierarchy.

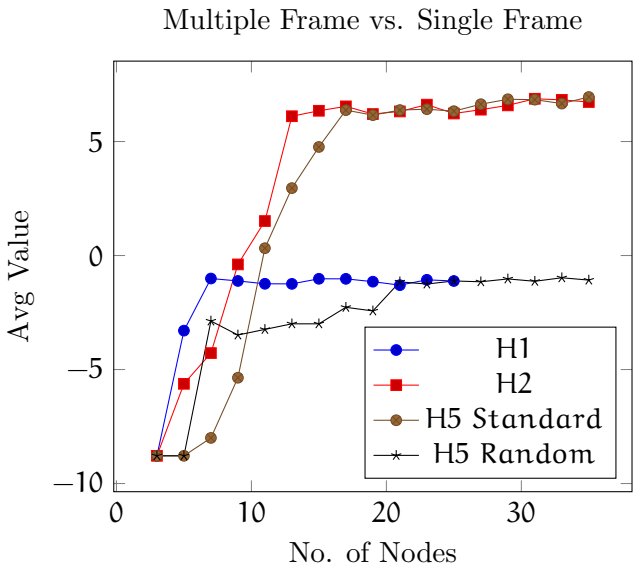
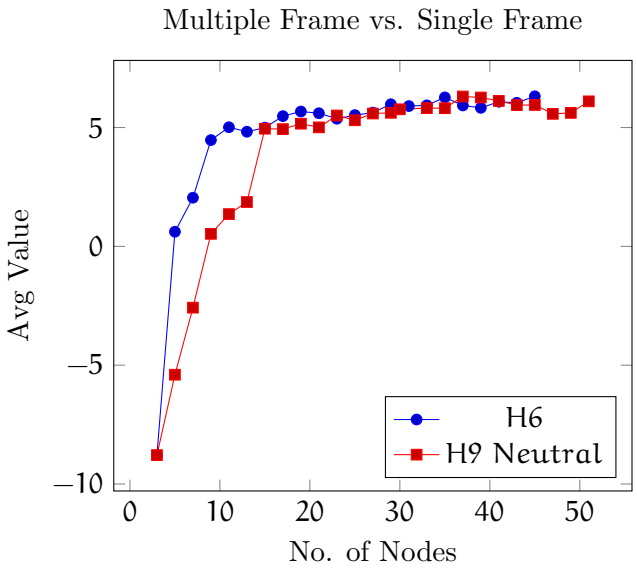
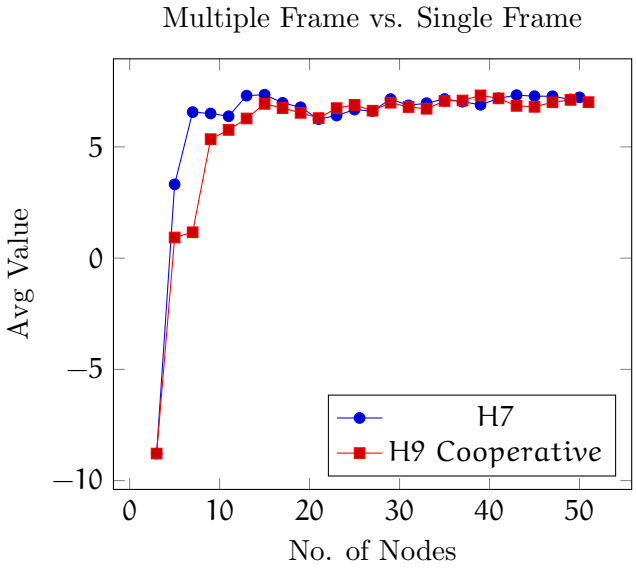


Figure 18: Level 2 Single Frame vs. Multiframe comparison

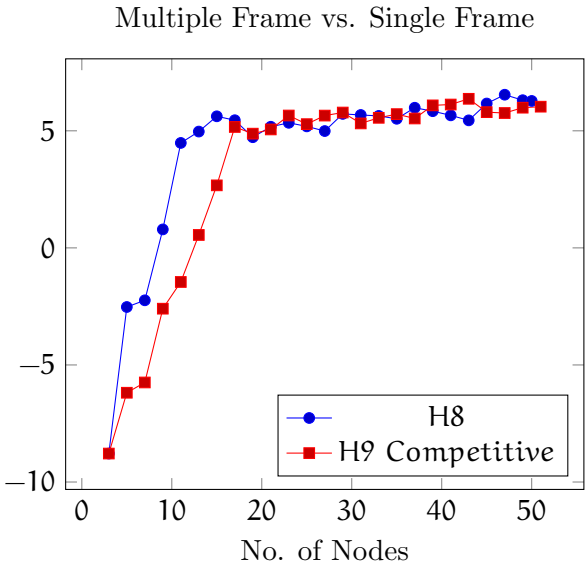
As showed in Figure 21c, these controllers need more nodes to achieve good performance. After the necessary number of nodes is reached, their value is comparable with the single frame version of the same controller. This is caused by the fact that a multiple frame controller needs



(a) Neutral



(b) Cooperative



(c) Competitive

Figure 19: Multiple frame vs. Single frame during solving

nodes to describe the behavior to react to multiple possible frame, in addition to intermediate nodes that are used to recognize the frame of agent j. The type of the simulated agent j for multiframe hierarchies is specified in the legend.

These controllers require more memory and time to develop, but are more flexible and allow near-optimal behavior even when there is uncertainty about agent j's type.

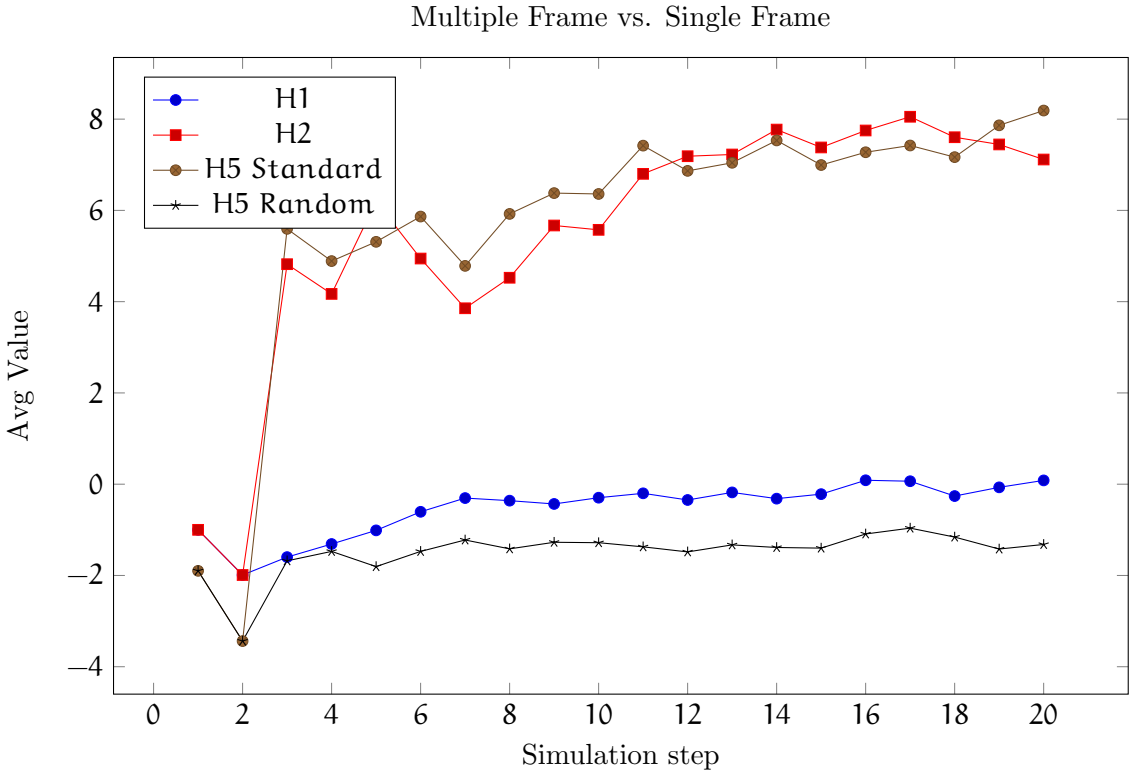
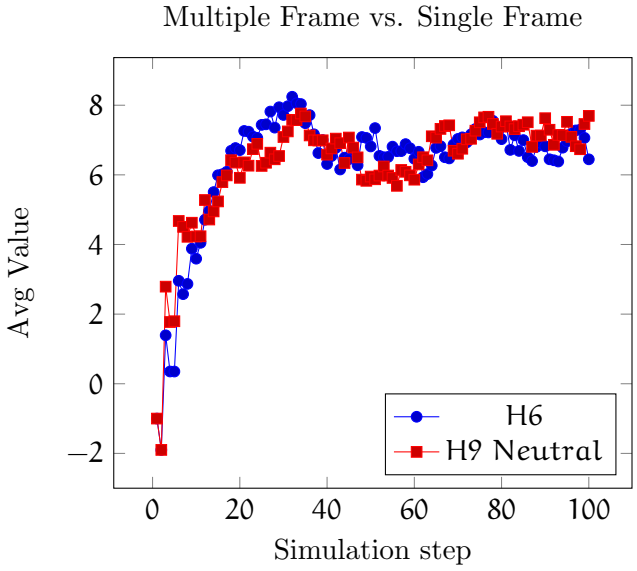
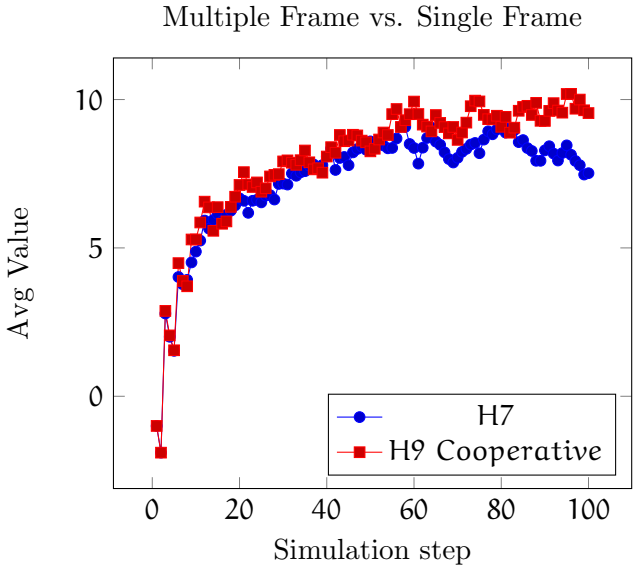


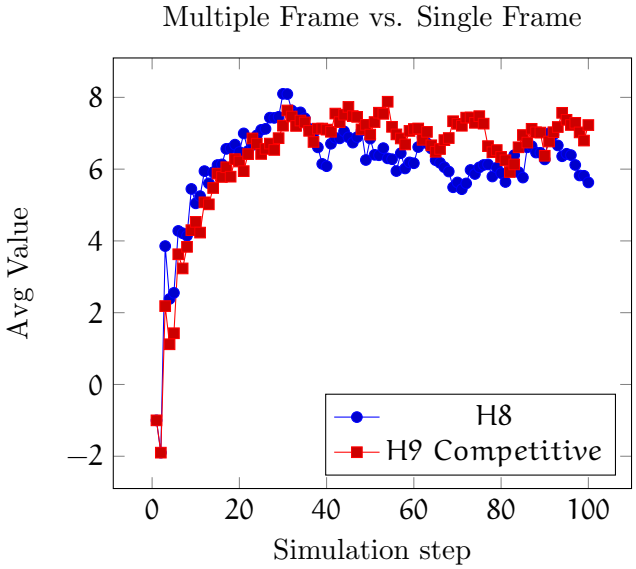
Figure 20: Level 2 Single Frame vs. Multiframe during Simulation



(a) Neutral



(b) Cooperative



(c) Competitive

Figure 21: Level 3 Multiple frame vs. Single frame during simulation

As showed in Figure 21c, these controllers need more nodes to obtain good results. After the necessary number of nodes is reached, their value is comparable with the single frame version of the same controller. This is caused by the fact that a multiple frame controller needs nodes to describe the behavior to react to multiple possible frame, in addition to intermediate nodes that are used to recognize the frame of agent j. The type of the simulated agent j for multiframe hierarchies is specified in the legend. These controllers require more memory and time to develop, but are more flexible and allow near-optimal behavior even when there is uncertainty about agent j's type.

6.3.3 Interacting with higher level agents

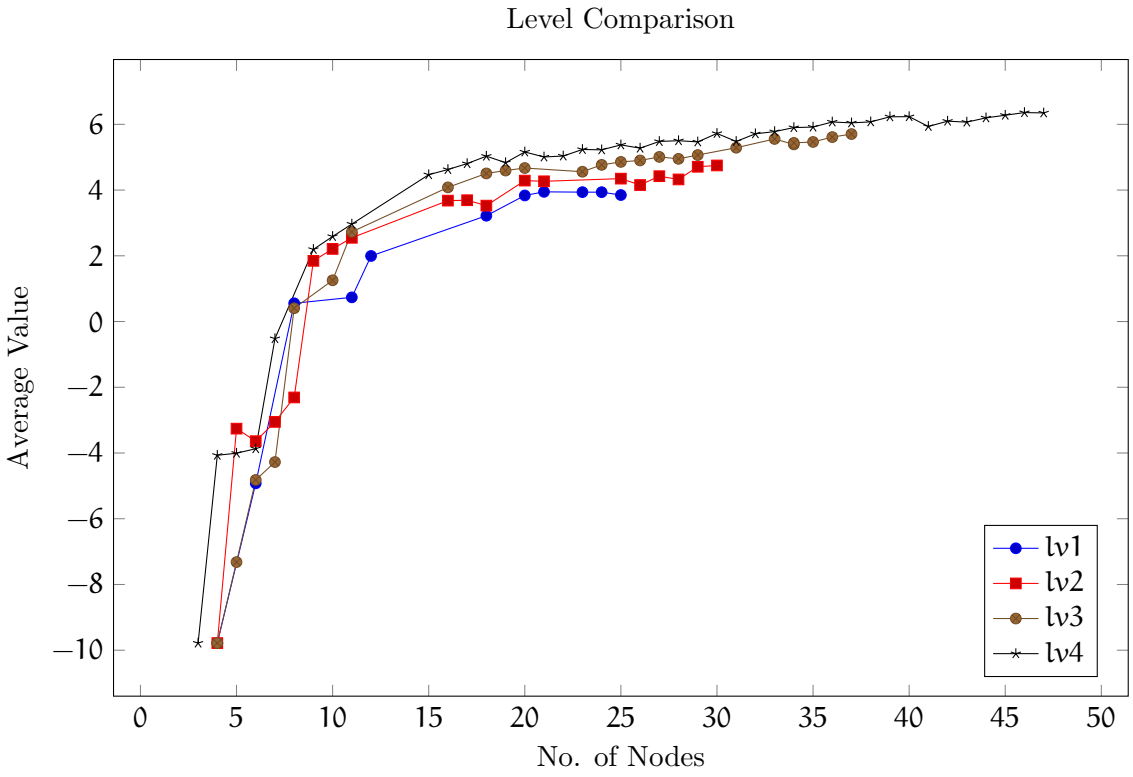


Figure 22: Average value with a higher level agent j

In this section the average value for agents of different strategy levels is reported. The reported values are obtained by agent i's of strategy levels from 1 up to 4 playing with a common agent j of the highest strategy level obtained. The performances improve as the other

agent is described with increasingly complex models. This means that there is an advantage to compute more strategic agents when the other agent's strategy level is unknown.

CHAPTER 7

BEHAVIOR ANALYSIS

7.1 Overview

I-BPI allows to obtain good solutions for complex hierarchies, but the resulting controllers are difficult to interpret. In this chapter I propose methods to analyze the behavior that is obtained by following the finite state machines, in order to understand what the agent does and why it is optimal or near-optimal.

Multiple level-1 frames are used to see the differences among cooperative, neutral and competitive agents.

7.2 Scenarios

This technique consists in putting the agent through specific sequences of transitions and observations called *Scenarios*, and observing the actions that the different agents execute in response to it. In order to rule out the stochastic nature of the controllers, the scenarios is re-executed multiple times. Each scenario is run 1000 times and the behavior of the agent is recorded. The behaviors with the most occurrences are recorded and analyzed. Scenarios require previous study of the behavior of the finite state machine, and are usually problem (and frame type) specific. I describe next some scenarios for the Multi-agent tiger game, describing what behavior each scenario tries to examine and what is the most common action string executed by the agents.

- Scenario 1: Standard execution

TABLE II: STANDARD EXECUTION SCENARIO

Step	1	2	3	4	5	6	%
State	TR	TR	TR	TL	TL	TL	
Obs_i	GRS	GRS	Any	GLS	GLS	Any	
Obs_j	GR	GR	Any	GL	GL	Any	
H2	L, L	L, L	OL, OL	L, L	L, L	OR, OR	0.998
	L, L	L, L	L, OL	L, L	L, L	L, OR	0.002
H3	L, L	L, L	OL, OL	L, L	L, L	OR, OR	0.999
	L, L	L, L	OL, OL	L, L	L, L	L, OR	0.001
H4	L, L	L, L	OL, OL	L, L	L, L	OR, OR	1.0

This scenario examines what happens whenever there are no wrong observations. This is mostly used to examine what is the "standard" behavior of a certain agent. The results show that higher strategy level agents follow a similar policy as a normal POMDP agent.

Any agent waits to receive two more observations on one side, together with silence from the other agent, before opening the door.

- Scenario 2: Immediate False Creak

TABLE III: IMMEDIATE FALSE CREAK SCENARIO

Step	1	2	3	4	5	6	%
State	TR	TR	TR	TL	TL	TL	
Obs_i	GRCL	GRS	GRS	GLS	GLS	Any	
Obs_j	GR	GR	GR	GL	GL	Any	
H2	L, L	L, L	OL, OL	L, L	L, L	OR, OR	0.986
	L, L	L, L	L, OL	OL, L	L, L	L, OR	0.014
H3	L, L	L, L	OL, OL	L, L	L, L	OR, OR	1.0
H4	L, L	L, L	OL, OL	L, L	L, L	OR, OR	1.0

This scenario is used to examine how an agent reacts when a creak is received immediately after the start of the execution or straight after opening a door. Agent i's node represents a belief range in which it is extremely unlikely that agent j opens any door, and, as such,

the creak will be recognized as a false creak and ignored. The resulting behavior does not differ from standard execution.

- Scenario 3a: Opposite side creak after one growl

TABLE IV: OPPOSITE SIDE CREAK AFTER ONE GROWL SCENARIO

Step	1	2	3	4	5	6	7	%
State	TR	TR	TR	TR	TL	TL	TL	
Obs_i	GRS	GRCL	GRS	GRS	GLS	GLS	Any	
Obs_j	GR	GL	GR	GR	GL	GL	Any	
H2	L, L	L, L	L, L	OL, L	L, OL	L, L	OR, L	0.982
	L, L	L, L	L, L	L, L	OL, OL	L, L	L, L	0.018
H3	L, L	L, L	L, L	OL, L	L, OL	L, L	L, L	0.957
	L, L	L, L	OL, L	L, L	L, OL	L, L	OR, L	0.043
H4	L, L	L, L	L, L	OL, L	L, OL	L, L	OR, L	1.0

This scenario is used to check how agent i reacts to a creak obtained after a growl on the opposite side. Having received a first growl, the other agent might be closer to opening

and agent i is not as sure if it is a false creak. Agents listen one extra time to make sure that the other agent did not actually open the door.

- Scenario 3b: Same side creak after one growl

TABLE V: SAME SIDE CREAK AFTER ONE GROWL SCENARIO

Step	1	2	3	4	5	6	7	%
State	TR	TR	TR	TR	TL	TL	TL	
Obs_i	GRS	GRCR	GRS	GRS	GLS	GLS	Any	
Obs_j	GR	GL	GR	GR	GL	GL	Any	
H2	L, L	L, L	L, L	OL, L	L, OL	L, L	OR, L	1.0
H3	L, L	L, L	L, L	OL, L	L, OL	L, L	OR, L	0.998
	L, L	L, L	L, L	OL, L	L, OL	L, L	L, L	0.002
H4	L, L	L, L	L, L	OL, L	L, OL	L, L	OR, L	0.98
	L, L	L, L	L, L	L, L	OL, OL	L, L	L, L	0.02

This scenario is used to check how agent i reacts to a creak obtained after a growl on the same side. Having received a growl on the right side, agent i thinks that there is a good chance that the tiger is behind the right door. A creak on the right means that j opened

the right door, which is unlikely given i's belief. Despite this, all kinds of agent i wait one extra turn to listen.

- Scenario 4a: True Creak

TABLE VI: TRUE CREAK SCENARIO

Step	1	2	3	4	5	6	%
State	TR	TR	TR	TL	TL	TL	
Obs_i	GRS	GLS	GLCL	GLS	GLS	Any	
Obs_j	GR	GR	Any	GL	GL	Any	
H2	L, L	L, L	L, OL	L, L	OR, L	L, OR	1.0
H3	L, L	L, L	L, OL	L, L	OR, L	L, OR	0.964
	L, L	L, L	L, OL	L, L	L, L	OR, OR	0.036
H4	L, L	L, L	L, OL	L, L	OR, L	L, OR	1.0

To obtain this scenario, it is necessary to introduce some wrong observations on agent i's side so J can receive two correct observations in a row and open the door.After hearing the creak, agent i recognizes that there is a chance that one of the observations it received

were wrong and that agent j might have opened the door, so it ignores any growls received so far and starts following standard behavior from a neutral belief state.

- Scenario 5: Late Agent j

TABLE VII: LATE AGENT J SCENARIO

Step	1	2	3	4	5	6	7	%
State	TR	TR	TL	TL	TR	TL	TL	
Obs_i	GLS	GRS	GLCL	GLS	GLS	GLS	Any	
Obs_j	GR	GR	Any	GL	GL	Any	Any	
H2	L, L	L, L	L, OL	L, L	OR, L	L, OR	L, L	1.0
H3	L, L	L, L	L, OL	L, L	L, L	OR, OR	L, L	0.977
	L, L	L, L	L, OL	L, L	OR, L	L, OR	L, L	0.023
H4	L, L	L, L	L, OL	L, L	OR, L	L, OR	L, L	1.0

This scenario is mostly used to analyze how different frame types react to agent j having received one less growl on the same side. The first part of the scenario is needed to bring the agents in such a situation without allowing I to open. If agent i opens as soon as it got the two more growls on the left side on turn 5, there is a 50% chance that the tiger moves

and that j, which is oblivious to the actions of i, opens the wrong door. Neutral agents do not consider the reward of agent j in their own reward function, so they open as soon as possible. Cooperative agents, instead, wait one extra turn to coordinate with agent j, as j opening the wrong door would yield a negative reward on turn 6. Competitive agents act in the same way as neutral agents, with the added benefit of gaining extra value if the tiger is reset and j opens the wrong door.

CHAPTER 8

FUTURE WORK

Stochastic finite state controllers are poorly readable and add a lot of extra computations to the algorithm. New techniques that use deterministic nodes have been developed [17] and the improvement and escape steps of I-BPI can be swapped out with those used by the more efficient single-agent techniques. This would produce more easily readable deterministic controllers and avoid a lot of issues with inaccuracy due to floating point representation and normalization. It would also remove the need for multiple repetitions during scenario analysis, making it faster and more reliable.

Due to the interleaved improvement steps of controllers at different levels, I-BPI can be parallelized by distributing controllers among different machines. This would allow to execute the basic iteration step on all controllers in a concurrent way, optimizing the process and having more memory available for each controller.

A concurrent simulator for the Julia framework would also be extremely useful, as executing enough simulations to have stable data requires a long time, slowing down the research process.

One of the main challenges encountered during the behavior analysis sections was that, whenever an agent opens a door, the received observation is completely uninformative regarding both the tiger position and the action of the other agents. Because of this, agents

often lose track of each other. One possible solution would be to define an I-POMDP where the observation received after opening a door is uninformative on the tiger location but still gives some information on the other agent's action. With this modification, more interesting behaviors would arise due to more precise information on the other agents' actions.

CITED LITERATURE

1. Sonu, E. S.: Scalable algorithms for sequential decision making under uncertainty in multiagent systems. PhD Dissertation, 24(1):29–80, 2015.

2. Olivo, I.: Solving interactive pomdps in julia. 2016.

3. Gmytrasiewicz, P. J. and Doshi, P.: framework for sequential planning in multi-agent settings. J. Artif. Int. Res., 24(1):49–79, 2005.

4. Sonu, E. and Doshi, P.: Generalized and bounded policy iteration for finitely-nested interactive pomdps: scaling up. in AAMAS, 2012.

5. Doshi, P. and Gmytrasiewicz, P. J.: Monte carlo sampling methods for approximating interactive pomdps. J. Artif. Int. Res., 34:297–337, 2009.

6. Anthony R. Cassandra, M. L. L. and Zhang., N. L.: Incremental pruning: A simple, fast, exact method for pomdps. Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence, pages 56–41, 1997.

7. Leslie Pack Kaelbling, Michael L. Littman, A. R. C.: Planning and acting in partially observable stochastic domains. Artificial Intelligence 101, pages 99–104, 1998.

8. Astrom, K.: Optimal control of markov decision processes with incomplete state estimation. J.Math.Anal.Appl., 10:174–205, 1995.

9. Smallwood, R. and Sondik, E.: The optimal control of partially observable markov processes over a finite horizon. Operations Research, 21:10711088, 2002.

10. Sondik, E. J.: The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. Oper. Res., 26(2):282–304, 1978.

11. Hansen, E. A.: Solving pomdps by searching in policy space. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, pages 211–219, 1998.

CITED LITERATURE (continued)

12. Poupart, P.: Exploiting structure to efficiently solve large scale partially observable markov decision processes. 2005.

13. Poupart, P.: Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes. Doctoral dissertation, Toronto, Ont., Canada, Canada, 2005. AAINR02727.

14. Aberdeen, D.: Scaling internal-state policy-gradient methods for pomdps. 2002.

15. Meuleau, N., Peshkin, L., Kim, K., and Kaelbling, L. P.: Learning finite-state controllers for partially observable environments. CoRR, abs/1301.6721, 2013.

16. Ng, A. Y. and Jordan, M. I.: PEGASUS: A policy search method for large mdps and pomdps. CoRR, abs/1301.3878, 2013.

17. Grzes, M. and Poupart, P.: Incremental policy iteration with guaranteed escape from local optima in pomdp planning. In Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS '15, pages 1249–1257, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems.

VITA

NAME Riccardo Ficarra

EDUCATION M.S., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2019.

M.S., Specialization Degree in Data Science, Politecnico di Torino, Torino, Italy, 2019.

B.A., Computer Engineering, Politecnico di Torino, Torino, Italy, 2017.

LANGUAGE SKILLS

Italian	Native Speaker
English	Full working proficiency
2017/18	One year of study abroad in Chicago, Illinois
2016/17	Lessons and exams attended exclusively in English

COMPETITIONS Participated in the 2018 Cyberforce Competition hosted by Argonne National Laboratory

Participated in the 2018 ITA Tech Challenge hosted by Trustwave, and ranked in the top 150.

SCHOLARSHIPS Italian scholarship for Extra-UE students from Politecnico di Torino