Sampling methods for Mini-Max Action Identification

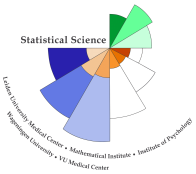
J.A. Dubbeldam

Thesis advisor: Dr. W.M. Koolen

Second advisor: Dr. T.A.L. van Erven

MASTER THESIS

Defended on December 20, 2016



STATISTICAL SCIENCE
FOR THE LIFE AND BEHAVIOURAL SCIENCES

Jarko Dubbeldam, 2016
jarkodubbeldam@gmail.com

Summary

Mini-max is a concept often used for solving games. The idea behind it is a constant alternation of minimizing and maximizing the value of moves to account for an adversarial opponent in the game. Lots of established methods have been developed to allow computers to play games like Chess [2] and Go [12]. Many of these methods involve evaluating sequences of moves to determine the best move to play. Because games like Chess and Go are quite big, it is infeasible to evaluate all possible sequences, so we resort to algorithms that pick sequences selectively, collected under the name Monte Carlo Tree Search [5]. These methods, in their quest to find the best move, already try to play as optimal as possible while figuring out the best move. We think that by letting go of the desire to only sample good sequences, and instead only caring for a good conclusion on the best move, we can improve on current algorithms.

We do this by adapting Best-Arm Identification's objective to fit a Mini-max structure: Mini-max Action Identification. We believe that this has not been done before. In Section 2 we will establish the framework and details of Mini-max Action Identification.

We define the problem of finding an optimal algorithm in two ways: In Section 3 we define the problem as the algorithm that provides the best guaranteed performance on the hardest set of parameters. In Section 4 the problem will be based on parameters following a fixed distribution. Further algorithms will be provided in Section 5. In this Section the algorithms will be compared in their performance as well.

Lastly we present some findings on the worst-case set of parameters in Section 6, providing proofs on a couple of these findings.

Acknowledgments

This thesis was written under daily supervision of Wouter Koolen at 'Centrum Wiskunde & Informatica' (CWI). I want to thank Wouter and the CWI for giving me this opportunity. I also want to thank Wouter for helping put things in more 'mathematical' phrasing, especially the proofs. The patience, great ideas and valuable feedback are all things I really appreciated during my time there. Thanks also to the staff at CWI and the group 'Algorithms and Complexity' for the nice lunch meetings and providing remote access to a computer at the CWI to run my experiments on.

Thanks to Tim van Erven, my supervisor at Leiden University, who not only gave a fresh view of the problems, but sparked my interest in Machine Learning in the first place and pointed me to Wouter and this project.

Finally, I want to thank Luc Edixhoven, who not only managed to withstand my endless ramblings about this project for the majority of seven months, but also helped me out with writing and debugging what is my first project in LaTeX.

Contents

1

Introduction

In this Section we will introduce the idea of Mini-max Action Identification by the hand of a few examples. To this end we will introduce the concepts of Best-Arm Identification and Mini-max, and show how the intersection of the two provide methods that can be used to solve the examples.

1.1 General introduction

To set the stage, let us consider the following real-life example of the problem we will consider in detail:

Example 1. *Imagine, the San Diego Comic Con is right around the corner and you really want to go. However, you are really late booking a hotel-room for your stay. Most of the hotels are already filled up. You find two hotels with each one room leftover. The hotel-owners of course give away their best rooms first, so the rooms left available are likely to be the worst rooms. You do not want a bad room, so your goal is to find the best room available. Sadly, the quality of the rooms is not directly available, so you will have to guess the quality through reviews on a review website like TripAdvisor. You can see reviews of each specific room, but you do not know which rooms are already taken.*

In the end you get one choice, which hotel do you pick? You want to pick the hotel where the worst room is the best. The best rooms are already taken, so there is no point paying attention to which hotel has the very best room overall, because there is no chance that you would get that room anyways. There is also another problem: the website has kindly notified you that there are 20 other people looking at that page, so you cannot just go and read every review available to you; you want to reach a conclusion as fast as possible. How do you spend your time as efficiently as possible? Once you've identified that a room is not the worst in its hotel, there is no point spending time reading reviews to determine exactly how good it is. It would be way more beneficial to take a better look at a worse room, because that is the one you might end up getting. So that room actually matters when you try to compare the hotels.

This problem has two defining properties: the data we have access to and the structure of the problem. To start with the type of data. The data are the reviews, they are not a direct value for

the rooms, but rather represent a sample, a noisy sample because of opinions of various reviewers. If there was no uncertainty about the quality of the rooms, the problem would be simple. Another notable feature is the fact that there is a structure in the problem: there are multiple layers at work here. The goal is to pick the hotel whose worst room is the best among the worst rooms of all hotels. There are two layers: first you want to know what the worst room in each hotel is, then you want to know which of those worst rooms is the best. If the goal was just to find the best room of all rooms combined, there would have been only one layer to the problem.

		Amount of layers	
		1	> 1
Data	samples	Best-Arm Identification	Mini-max Action Identification
	values	argmax	Mini-max

Table 1.1: The defining properties of Example 1: the type of data and the structure of the problem. This determines the methods that can be used to solve the problem.

Using these features, the problem in Example 1 would fall in the top-right corner in Table 1.1: Mini-max Action Identification. Whereas the other three are already widely studied, Mini-max Action Identification is new. As far as we can tell, the only related work is the recent [4], which independently studies Mini-max Action Identification under different evaluation criteria. To introduce Mini-max Action Identification, we first introduce the other entries of Table 1.1:

We may simplify Example 1 along two orthogonal dimensions (Table 1.1). First, we may remove the *statistical* aspect (the noise in the reviews), by imagining that each room has a known quality score. Second we may remove the *game-theoretic* aspect (the adversarial per-hotel room selection) by imagining that each hotel has exactly one room.

With noise nor adversary we arrive at the problem of finding the position of the best entry in a list of numbers. This is the "argmax" problem, which can be trivially solved in a single pass over the list. With an adversary but no noise we arrive at the core Mini-max problem studied in game theory. This will be reviewed in Section 1.2 below. With noise but no adversary we instead obtain the so-called Best-Arm Identification problem. This problem has recently seen a lot of progress in the literature on bandit problems. We will review it in Section 1.3. Section 1.4 will introduce the full MMAI problem (the formal setup is the topic of Section 2). We conclude the introduction by sketching the outline of the thesis in Section 1.5.

1.2 Mini-max

To properly explain Mini-max Action Identification, we first have to introduce the simpler problems it generalizes. So first we drop the noise from samples and look at Mini-max (Table 1.2).

		Amount of layers	
		1	> 1
Data	samples	Best-Arm Identification	Mini-max Action Identification
	values	argmax	Mini-max

Table 1.2: Mini-max.

We need a way to deal with the multiple layers: $\text{argmax}_{hotel} \min_{room}$. This alternation of minimizing and maximizing is called Mini-maxing.

One example where this problem is prevalent is games. When determining the best move, the player has to keep in mind what the opponent is going to play. In most cases this will mean that the opponent plays their best move, which would be bad for the player. So the best move would be the best move among the worst outcomes one turn down the road. The intuition here is easiest to explain in the form of a simple game. Imagine a game where two players each get one move, a choice between A and B (Figure 1.1). After both players played their moves, player 1 gets a reward based on what both players picked. Player 2 wants that reward to be as small as possible. Similarly here, player 1 could focus on trying to get the highest reward possible, but it is unlikely that player 2 will allow that to happen, so he will play the other available move. Because of this, player 1 should not even focus on the highest reward, but rather figure out the move which forces the player 2 to give a (relatively) high reward to player 1.

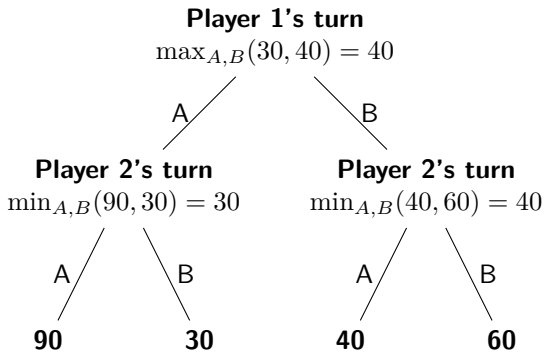


Figure 1.1: An example of a Mini-max problem: Player 1 wants to get the highest possible score, whereas player 2 wants the lowest possible. Player 1 could try and get the 90 score and thus play A, but then player 2 will take move B, so player 1 ends up with a score of 30. So instead player 1 looks at the possible outcomes from the two moves A ($\{90, 30\}$) and B ($\{40, 60\}$), figures out the lowest possible outcomes of each move, and picks the highest of those. The best move for player 1 is then move B.

Example 2. *A good example to demonstrate Mini-max in action is Tic Tac Toe. This is a two-player game where both players have opposite interests; if one wins, the other loses. There are two players: X (he) and O (she). From the perspective of X, a winning board has a value of 1 and a losing board -1. Say the game has been going on for a couple of rounds and the board looks like this:*

X	X	O
O		
O		X

It is X's turn. The way to figure out the best move is to create a tree showing all possible sequences of moves and perform a Mini-max search over it. This tree is displayed in Figure 1.2. X has three possible moves: move 2, 3 and 4. To evaluate the value of those moves, we have to follow the branch in the tree until the game is over. For move 2, it is simple. X wins instantly. This would obviously be the optimal move to take for X. But for the sake of the example, lets see how the values for the other moves are calculated. To do this, we start at the bottom of the tree and work our way up. State 9 and 10 are the only possible outcomes for state 6 and 8 respectively, so X is forced to make

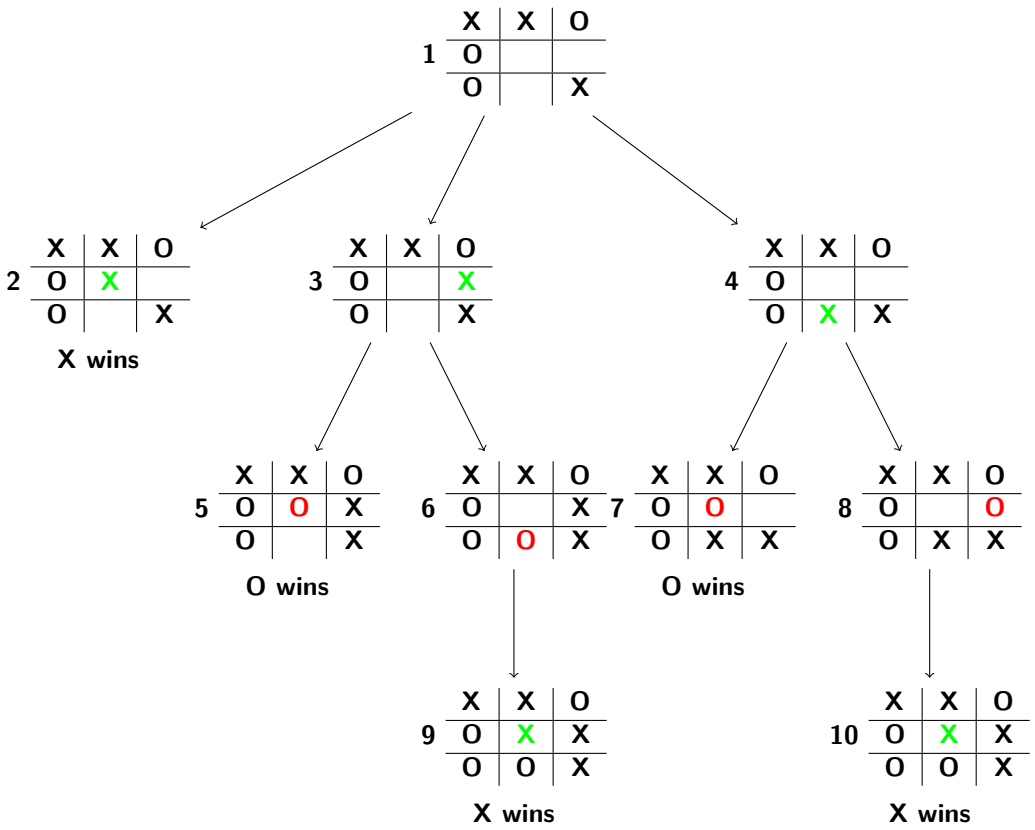


Figure 1.2: Tree of the possible moves in the Tic Tac Toe game in Example 2.

those moves if he is in state 6 or 8. X wins, so the value of states 6 and 8 are both 1. Besides state 6 there is state 5, the other move available for O in state 3. If O makes move 5, she wins, so this has a value for X of -1. Obviously O wants to win as well, so if she has to move in state 3, she would pick move 5. When O is at play, she minimizes the pay-off for X. And X has to keep this in mind. So when evaluating move 3, X knows that O will pick 5 and then win, so he assigns the value -1 to state 3. The same goes for state 4, where O would win with move 7. State 2 now has value 1, state 3 has value -1, and state 4 has value -1. X wants the best move, so picks move 2 and wins.

The difficulty in more complex games like chess is that the amount of moves that each player can make is huge, so the tree, simple in Figure 1.1, would grow exponentially in the depth. This makes applying brute force mini-max in practice quite hard, but there are methods to help with this, like pruning. Another method is to stop after a certain amount of moves, and then approximate the value of those moves: Instead of enumerating the entire game tree, looking at every possible sequence of moves, stop after a certain amount of moves, and use a decent, established (greedy) strategy to play the rest of the game and see who wins (or what the final score is). This then gives an approximation of the quality of the moves made to that point (Figure 1.3). This method is called Monte Carlo Tree Search (MCTS) [5]. There are many algorithms that bring MCTS into practice, some of which presented in [5].

These roll-outs can be considered random samples, as they add some uncertainty to the values of game states. This is also in line with what we would need to solve the problem in Example 1. This is where Mini-max Action Identification comes in. Through the results and estimates gained by

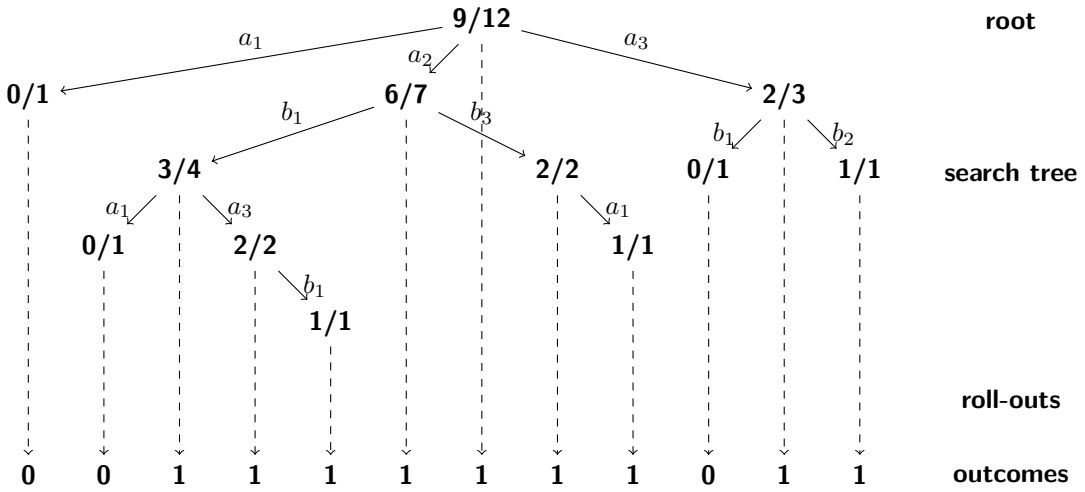


Figure 1.3: Example of Monte Carlo Tree Search, adapted from [5]. The algorithm moves through the game tree for a predefined couple of moves and then applies a generic strategy to play until the end of the game (roll-out), after which it gets an outcome (0/1, loss/win). This is then propagated back to the values of the sequence leading up to the node where the roll-out started.

the roll-out with an established strategy, an estimate is gained for the values of game states, after which a Mini-max search is performed to find the best move.

1.3 Best-arm Identification

If we take a closer look at how to draw conclusions about maximizing (or minimizing) an expectation based on some distribution, removing the game-theoretic aspect, reducing the number of layers to one, we end up in the domain of Best-Arm Identification (Table 1.3). Because a lot of the setting discussed in Section 2 is based on what is currently done in Best-Arm Identification, we will introduce this as well.

		Amount of layers	
		1	> 1
Data	samples	Best-Arm Identification	Mini-max Action Identification
	values	argmax	Mini-max

Table 1.3: Best-Arm Identification.

To illustrate Best-Arm Identification, let us consider another example:

Example 3. *Consider a clinical trial. You want to compare the effects of different drugs on patients. You can give each patient one drug, and can then measure their response by taking their vitals. Your goal is to find the best drug. Not all patients react similarly to the drugs, so there is some randomness involved. Also, after you have given a patient one drug, you cannot use any other drugs for that patient, so you cannot know what their response would have been. You apply the drugs sequentially, so you know the effects of all the drugs you previously applied to the previous patients. Which drug do you give next if you want to identify the best drug overall?*

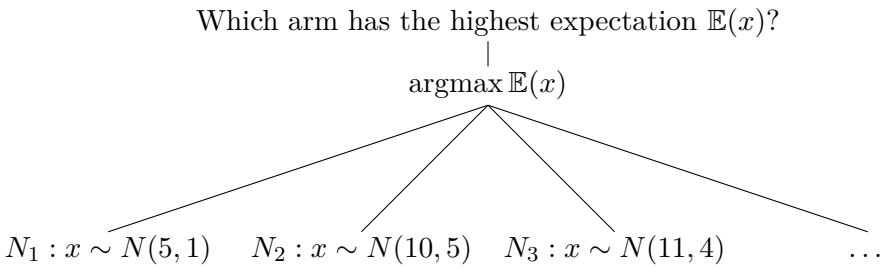


Figure 1.4: An example of a best-arm identification problem: which arm has the highest expectation for x? Each arm has a different distribution N_j on the value for x, which can be sampled by the algorithm and used to evaluate the expected value for x.

The problem of deciding which drug to apply next is widely covered in the field of Best-arm identification [7]. The goal of Best-Arm Identification is to identify the option that has the highest parameter or expectation of some response variable (Figure 1.4). Again the quality of the arms can be sampled with algorithms that decide where to look next. To make this problem mathematically precise, there are two distinct ways to approach the problem: fixed-budget and fixed-confidence. The setting determines the stopping rule of the algorithm, which has a lot of implications on the way the algorithm works.

In fixed-confidence the algorithm continues until it knows through probability theory, typically by means of bounds, that it has at most a δ chance to be wrong in its recommendation for the best arm, where δ is some predefined parameter. The algorithm is more efficient if it reaches this confidence δ in as low as possible amount of samples. After all, the more efficiently it allocates the drugs, the faster it'll be able to reach a conclusion. Alternatively, in fixed-budget, the algorithm receives a budget T, which represents the amount of samples it is allowed to draw, or how many patients are available. The quality of the algorithm then is determined by the quality of the recommendations; how often does the algorithm identify the correct arm?

1.4 Mini-max Action Identification

Now we have properly introduced Best-Arm Identification and Mini-max, we combine the two into Mini-max Action Identification by replacing the optimization objective of Best-Arm Identification with the Mini-max method of alternating maximization and minimization (Table 1.4).

		Amount of layers	
		1	> 1
Data	samples	Best-Arm Identification	Mini-max Action Identification
	values	argmax	Mini-max

Table 1.4: Mini-max Action Identification.

One thing many MCTS-algorithms do in their quest to find the best move, is already trying to play as optimal as possible while figuring out the best move. We think that by letting go of the desire to only sample good sequences, and instead only caring for a good conclusion on the best move, we can improve on current algorithms. Therefore we apply Best Arm Identification to the MCTS methods described in Section 1.2 to provide a way to determine the Mini-max move based on sampled data.

The algorithm moves through the tree of states (Figure 1.5), up until some predefined number of moves, and then uses an established strategy to roll-out the game until the end. This is abstracted to drawing a sample from that game state. The goal of Mini-max Action Identification is to 'sample' the moves efficiently until it has an estimate of what the Mini-max move is.

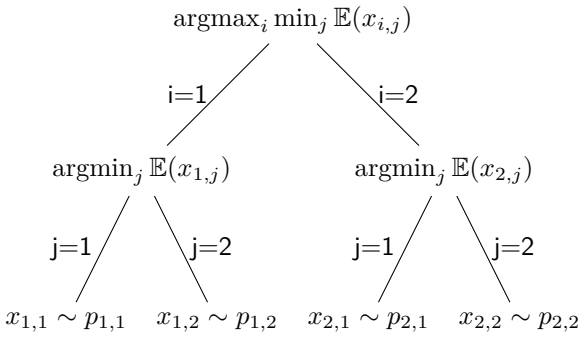


Figure 1.5: Tree structure of the Mini-max Action Identification problem (See Example 1: i are the hotels and j are the rooms within hotels).This is a combination of Mini-max and Best-Arm Identification. If you remove the randomness of the samples, this problem is the same as in Figure 1.1. Alternatively, the argmin_j and argmax_i can be seen as individual examples for Best-Arm Identification (See Figure 1.4).

1.5 In this work

In this thesis we will present some sampling methods to be used to determine the Mini-max action. In Section 2 we will define the setting and provide additional information, as well as present the backbone sampling algorithm. In Sections 3 and 4 we will present two elaborate algorithms: the Optimal algorithm and the Bayesian Expected Regret respectively. In Section 5 we present some more practical algorithms and make a comparison between them. Section 6 elaborates on the question 'what are the worst-case parameters?' We do this by showing that the parameters follow a certain pattern. In Section 7 we give a recap of the thesis and make some recommendations for future work.

2

Mini-max Action Identification

The problem introduced in Section 1 can be seen as an combination of Best-arm Identification and Mini-max. In this Section we will define the setting in which the Mini-max Action Identification algorithm operates.

2.1 The game

This thesis focuses on optimizing the Mini-max Action Identification algorithm involved in figuring out the mini-max best move problem in games from noisy leaf evaluations (see Example 1). To focus as much as possible on the sampling rule part of the algorithm, we abstract the random play-out by some parameter $\{p_{i,j}\}$ for each node, representing the probability of winning. The intuition behind this replacement is that the sample generated by random play-out has inherently some chance to win based on the quality of the move pair, so drawing a sample from a Bernoulli distribution with that same parameter should give the algorithm the same information. For simplicity, we assume that samples from the terminal nodes are i.i.d.

Additionally, we reduce the amount of moves available to the bare minimum, namely two, ending up in the same game tree as described in Figure 1.5. Each of the four terminal nodes receives a $\{p_{i,j}\}$, a win-chance. Player 1 uses Mini-max Action Identification to get an estimate of the $\{p_{i,j}\}$ of each of the terminal nodes. The algorithm receives some budget of T samples. We use the fixed-budget setting instead of the fixed-confidence (Section 1.3), because it makes more sense to have a time-based restriction on the move, rather than an error-based. The reason for this is that games generally limit the time players have to think about their moves. This is more in line with fixing the budget than fixing the confidence. The budget provided can be spent on sampling one variable from any of the four terminal nodes' distributions. After the budget has been spent, the algorithm makes a recommendation based on the sampled results. Player 1 plays the move recommended by the algorithm. Player 2 then plays his move, but does not sample. Instead, player 2 is assumed to be all-knowing, he knows the true parameters behind the terminal nodes and will always pick the move that minimizes the win-chance. Afterwards the recommendation is evaluated to measure the performance of the algorithm. If it recommended the sub-optimal move, some loss will be assigned, more on that in Section 2.3.

2.2 The Min-max Action Identification algorithm

On initialization, the algorithm receives a sampling budget T. This specifies the amount of samples the algorithm is allowed to draw before its recommendation. The algorithm uses the function `someSampleRule` to determine which node to sample from, based on all the previous sampling results. Similarly, when the budget T is spent, the function `someReccomendationRule` returns the recommended arm i, again based on all the results of the samples. The core algorithm is shown in algorithm 1.

Data: The set of true parameters $\{p_{i,j}\}$, unknown to player 1. A sampling budget T.
Result: A recommendation for player 1 for the mini-max arm.
initialization;
for $t = 1$ **to** T **do**
  $\{i, j\}_t \leftarrow$ `someSampleRule`;
  $x_{i_t,j_t} \sim Bern(p_{i_t,j_t})$;
end
`someRecommendationRule`;

Algorithm 1: The core of a Min-max action identification algorithm. The functions `someSampleRule` and `someRecommendationRule` are different for different algorithms and determine which node to sample from and which node to pick at the end respectively.

A very basic example of a sampling algorithm is the Equal-algorithm (Algorithm 2). This algorithm spreads the available budget equally over all the arms. As recommendation it suggests the move with the best expectation based on the Maximum Likelihood Estimator (MLE) of the parameter $\hat{p}_{i,j}$.

`someSampleRule` $\leftarrow$ function$(t)\{$
  $i \leftarrow t \mod 2 + 1$
  $j \leftarrow \lceil 0.5t \rceil \mod 2 + 1$
  return({i,j})
$\}$
`someRecommendationRule` $\leftarrow$ function$(x_{i_t,j_t})\{$
  foreach $\{i, j\}$ **do**
    $\hat{p}_{i,j} \leftarrow$ mean$(\{x_{i_t,j_t} | i_t = i$ and $j_t = j\})$
  end
  if $\min_j \hat{p}_{1,j} = \min_j \hat{p}_{2,j}$ **then**
    $i \leftarrow$ Bern$(0.5) + 1$
    `/* Tie: resolve uniformly at random. */`
  else
    $i \leftarrow \text{argmax}_i \min_j \hat{p}_{i,j}$
  end
  return(i)
$\}$

Algorithm 2: The Equal-algorithm's functions: it spends its budget equally over all combinations of $\{i, j\}$ and recommends based on $\hat{p}_{i,j}$.

2.3 Expected regret

In order to compare different algorithms, we need to find a measure that can quantify the performance of these methods. One popular objective is the error-rate (the probability that the chosen move $I = \texttt{someRecommendationRule}$ is not the optimal move $i^* = \text{argmax}_i \min_j p_{i,j}$: $\mathbb{P}(I \neq i^*)$). This does not take the severity of the error into account. To see this, consider a game in the form of Figure 1.5, where two arms are very close, or even equal ($\min_j p_{1,j} \approx \min_j p_{2,j}$): The Equal-algorithm (Algorithm 2) would have an error-rate approaching 0.5 as $|\min_j p_{1,j} - \min_j p_{2,j}| \to 0$. However, as the difference gets smaller, the negative effects of picking the wrong move i are smaller as well.

This is where the concept of regret comes in. Instead of having a 0/1-loss, measuring no penalty if the correct arm is picked and a unit penalty if the wrong one is, the regret can be used. The regret is set to be equal to the difference in quality between the optimal move i^* and the chosen one I:

$$R(I) = \max_i \min_j p_{i,j} - \min_j p_{I,j} = \min_j p_{i^*,j} - \min_j p_{I,j} \tag{2.1}$$

This way the errors made by the algorithm are scored based on how far the arms (the optimal one and the one picked) are actually apart. It is easy to see that if $i^* = I$, $R(I) = 0$. As I is random because of the randomness in the samples drawn by the algorithm and a possible randomness in the algorithm's recommendation, we can write the expectation on the Regret as follows:

$$\mathbb{E}(R) = \sum_I R(I)\mathbb{P}(I) \tag{2.2}$$

In the case of a simple game with two moves, like in Figure 1.5, $\mathbb{E}(R)$ depends on the error-rate $\mathbb{P}(I \neq i^*)$ and the possibly incurred regret $\delta = |\min_j p_{1,j} - \min_j p_{2,j}|$.

When comparing algorithms, it is most interesting to look at the worst-case scenario. In other words, what is the worst expected regret? Formally this would be $\max_{\{p_{i,j}\}} \mathbb{E}(R)$, maximizing the Expected regret over $\{p_{i,j}\}$, the set of parameters $p_{i,j}$. This measure is a guarantee that the algorithm performs better or equal to that value. Alternatively, we could use the expectation $\mathbb{E}_{\{p_{i,j}\}} \mathbb{E}(R)$, which requires some prior distribution on $\{p_{i,j}\}$, turning it into a Bayesian problem. We prefer to use the worst-case Expected regret, because this gives a guarantee on the performance of the algorithm in all cases and does not require us to put a prior on $\{p_{i,j}\}$.

To get an idea of how the Expected Regret works, consider the the Equal-algorithm (Algorithm 2) modified to work with Best-Arm Identification with a budget of $T = 20$. Instead of sampling equally over four terminal nodes, it samples over two arms instead. In this example the algorithm spreads the budget T equally over both arms, resulting in 10 samples for each arm. Its recommendation is based on $\text{argmax}_i \hat{p}_i$. $\hat{p}_i$ is the MLE of p_i, which is X_i/n_i, where X_i is the amount of successes in the Bernoulli trials and n_i the sample size (10 in this example). The algorithm picks incorrectly if $\hat{p}_{i^-} > \hat{p}_{i^*}$, where $i^* = \text{argmax}_i p_i$ and $i^- \neq \text{argmax}_i p_i$. Because $\hat{p} = X/n$, where X is the number of won games (successes in the binomial distribution):

$$\hat{p}_{i^-} \geq \hat{p}_{i^*} \iff X_{i^-} \geq X_{i^*}$$

The error-rate therefore is:

$$\mathbb{P}(X_{i^-} \geq X_{i^*}) = \sum_{k=0}^{n}\sum_{j=0}^{k}\Bigg[\underbrace{\binom{n}{j}(p_{i^-}+\delta)^j(1-(p_{i^-}+\delta))^{n-j}}_{\text{Probability of } \frac{j}{n} \text{ successes in } i^*}\underbrace{\binom{n}{k}{p_{i^-}}^k(1-p_{i^-})^{n-k}}_{\text{Probability of } \frac{k}{n} \text{ successes in } i^-}\underbrace{(1-0.5(\mathbf{1}_{\{j=k\}}))}_{\text{Randomize ties}}\Bigg] \tag{2.3}$$

All instances where $k \geq j$

With regret:

$$R(I) = \begin{cases} 0 & I = i^* \\ \delta = p_{i^*} - p_{i^-} & I = i^- \end{cases} \tag{2.4}$$

The Best-Arm simplification of Equation 2.1 drops the min terms.

$$\mathbb{E}(R;\delta,p_{i^-},n) = \delta\sum_{k=0}^{n}\sum_{j=0}^{k}\Bigg[\binom{n}{j}(p_{i^-}+\delta)^j(1-(p_{i^-}+\delta))^{n-j}\binom{n}{k}{p_{i^-}}^k(1-p_{i^-})^{n-k}(1-0.5(\mathbf{1}_{\{j=k\}}))\Bigg] \tag{2.5}$$

A plot of this function with $n = 10$ can be seen in Figure 2.1. On the diagonal, where $p_1 \approx p_2$ the regret factor δ in Equation 2.5 is dominant, pulling towards 0, whereas when p_1 and p_2 are further a part, the error-rate term takes over. The variance of samples $x \sim Bern(p)$ are equal to $p(1-p)$. This is highest with $p = 0.5$. This means that the estimates are most uncertain if $p = 0.5$, thus allowing more room for errors. Of course, then they are both equal to 0.5, the regret becomes 0, so they have to be different. The maximum of the expected regret is centered around 0.5, so these would be the hardest p_i to differentiate between. In Figure 2.1 the highest Expected Regret is at the parameters $p_1 = 0.416$ and $p_2 = 0.584$, as well as their mirror: $p_1 = 0.584$ and $p_2 = 0.416$.

We will use this algorithm as a building block in Section 5.

2.4 Mini-max in games

Mini-max Action Identification is combining Best-Arm Identification with Mini-max, so instead of looking for $\max_j \mathbb{E}(x_j)$, the maximum expectation for x_j, we want to pick the mini-max option: $\text{argmax}_i \min_j \mathbb{E}(x_{ij})$. Here x_{ij} is the value of the arm, or game, resulting if player 1 plays move i and player 2 plays move j (Figure 1.5). In boardgames the value generally ends up being 0/1, depending on whether it ends in a win or a loss, but if the algorithm stops at a certain level and then moves over to a reasonable default policy, the value becomes a Bernoulli variable with a certain probability $p_{i,j}$ to end up in a win. This parameter will then be the parameter from the Bernoulli distribution from which the Mini-max Action Identification algorithm samples.

The fixed-budget setting makes more sense as a problem formalization within games than fixed-confidence. This also goes for Example 1. There is only a limited amount of time to be spent looking at reviews, in other words, you have a budget of T amount of minutes to spend looking before the rooms are taken. Within games, often there is a soft budget limit, players in most competitive games have a limited amount of time to think about their moves. While time spent does not have to be linear in the amount of samples drawn, it makes more sense to use than fixed-confidence.

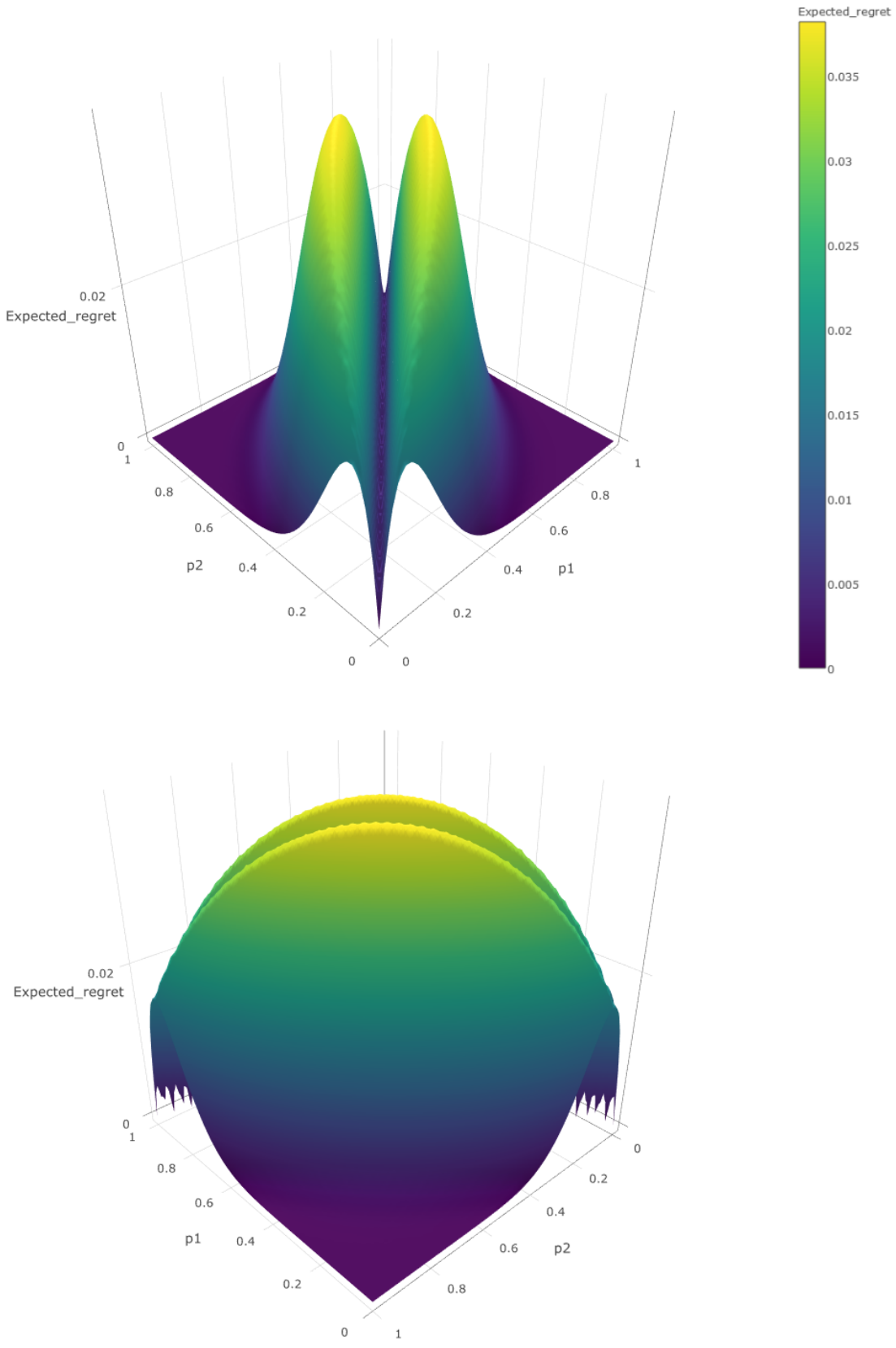


Figure 2.1: Expected regret plots (viewed from two perspectives) of Equal-algorithm (Algorithm 2) adjusted for two-arm Best-Arm Identification with a budget of $T = 20$.

3

The Worst-case Optimal algorithm

For fixed budget algorithms, there are two defining properties. The sampling rule and the recommendation rule. These decide how the algorithm acts while sampling and finding the best recommendation. There is of course a multitude of statistics from the results from earlier samples that might be used to decide which arm to sample from in the next iteration, but it is not immediately obvious how to design a good algorithm.

So instead, we do a search along all of the possible algorithms one could use as sampling- and recommendation rules. The way to approach this idea is to regard the search for an algorithm as a game in itself: with two players. Player 1 tries to find the best move, versus Player 2, Nature, which picks the values $\{p_{i,j}\}$ of the arms. Using game theory, the strategy to such a game can be optimized using a Linear Programming solver. Using this method, we can find an optimal strategy to finding the best move i against the worst-case $\{p_{i,j}\}$, in other words, the worst-case optimal algorithm. In Section 3.1 we define the problem. In Sections 3.2 to 3.8 we define the game and present simplifications to the problem to make its size manageable. In Section 3.9 we will show the performance of this optimal algorithm and in Section 5 we will use this algorithm in the comparisons.

3.1 Adversarial problem

The goal of the Mini-max Action Identification algorithm is to minimize the Expected regret, whereas the 'Opponent's' goal is to maximize this value. The problem can then be formalized to the following:

$$\min_{\substack{\text{Algorithm strategy} \\ (mixed)}} \max_{\{p_{i,j}\}} \mathop{\mathbb{E}}_{\substack{Samples \\ Recommendation}} R(I) \tag{3.1}$$

Where the samples and recommendation follow the protocol of Section 2.2 and:

$$R(I) \equiv \max_i \min_j p_{i,j} - \min_j p_{I,j} \tag{3.2}$$

Using the 'Minimax Theorem' [9] we can rewrite Equation 3.1 to:

$$\max_{Q} \min_{\substack{\text{Algorithm strategy} \\ (pure)}} \mathop{\mathbb{E}}_{\{p_{i,j}\}\sim Q} \mathop{\mathbb{E}}_{\substack{Samples \\ Recommendation}} R(I) \tag{3.3}$$

Where Q is the mixed strategy for the 'Opponent'.

3.2 The Algorithm-game

We define the game as follows: Player 1, the algorithm, tries to find the best move i (Figure 1.5) by sampling from the arms $\{p_{i,j}\}$, to minimize the Expected regret. Player 2, the opponent, picks the values of the arms $\{p_{i,j}\}$. The algorithm does not know the opponent's choice, so it has to consider all possible combinations of $\{p_{i,j}\}$. After the $\{p_{i,j}\}$ are picked, the algorithm plays alone. Each move available to the algorithm will represent one sample to be taken. Therefore, in every node of the game tree where the algorithm is at play, it has the same choices $C_{i,j} = \{1, 2\}^2$, which represent the arms to sample from. These moves are alternated by chance moves $\in \{0, 1\}$, representing a loss or a win returned from that sample respectively. We define a to be the total number of arms to sample from: $a = 4$ generally. In the case of the Best-Arm identification example (Section 1.3), $a = 2$.

In the end, the pay-off for the game is calculated based on the recommendation, expressed in expected regret. We define the game to be zero-sum; As the goal of the algorithm is to minimize the expected regret, the goal of the opponent becomes to maximize this. The solution found for this problem not only gives us an optimal algorithm to solve the best-arm or minimax identification, but also provides us with a distribution on worst-case $\{p_{i,j}\}$.

We begin this analysis enumerating all possible algorithms, and then move on using a series of simplifications provided by [8].

3.3 Definitions

The game can be represented in a tree (Figures 3.1 and 3.2). The root of the tree defines the start of the game. Each node in the tree represents either a move for player 1, a move for player 2 or a chance move. Chance moves in conventional games represent things like shuffling a deck, rolling a die, etc., and in this setting it represents a draw from the arm chosen by the preceding move for player 1, with the parameters chosen in the first move by player 2. Each node x, or leaf, in the tree represents the recommendation, which in turn corresponds with a penalty $h(x) \in \{0, 1\}$. The regret is zero when the recommendation is right, so the pay-off is the regret (Equation 2.1).

An example for a node x is as follows: there is a particular set $\{p_{i,j}\}$ picked by the opponent. For each sample in T, there is an arm $\{i, j\}$ picked to sample from, along with a win or a loss returned as sample. Then finally, based on the samples there is a recommendation I, which is either the correct arm ($I = \text{argmax}_i \min_j p_{i,j}$) and the pay-off is $h(x) = 0$, or it is the incorrect arm ($I \neq \text{argmax}_i \min_j p_{i,j}$) and the pay-off is $h(x) = \delta$ (Equation 2.1).

The algorithm is unaware of the $\{p_{i,j}\}$ picked by the opponent, so therefore cannot distinguish between the subtrees after the first move (Figure 3.1). Formally, all corresponding nodes between the subtrees belong to the same information set u. Player 1 cannot tell nodes $x \in u$ apart, so

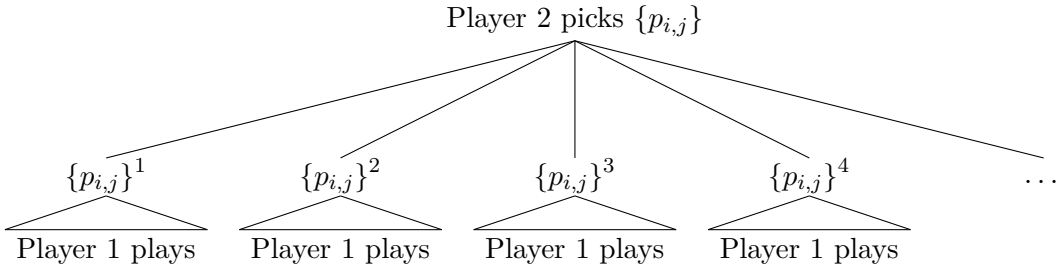


Figure 3.1: Game tree representation of the first move: the opponent picks a set of $\{p_{i,j}\}$ and the algorithm samples from the arms with those $\{p_{i,j}\}$.

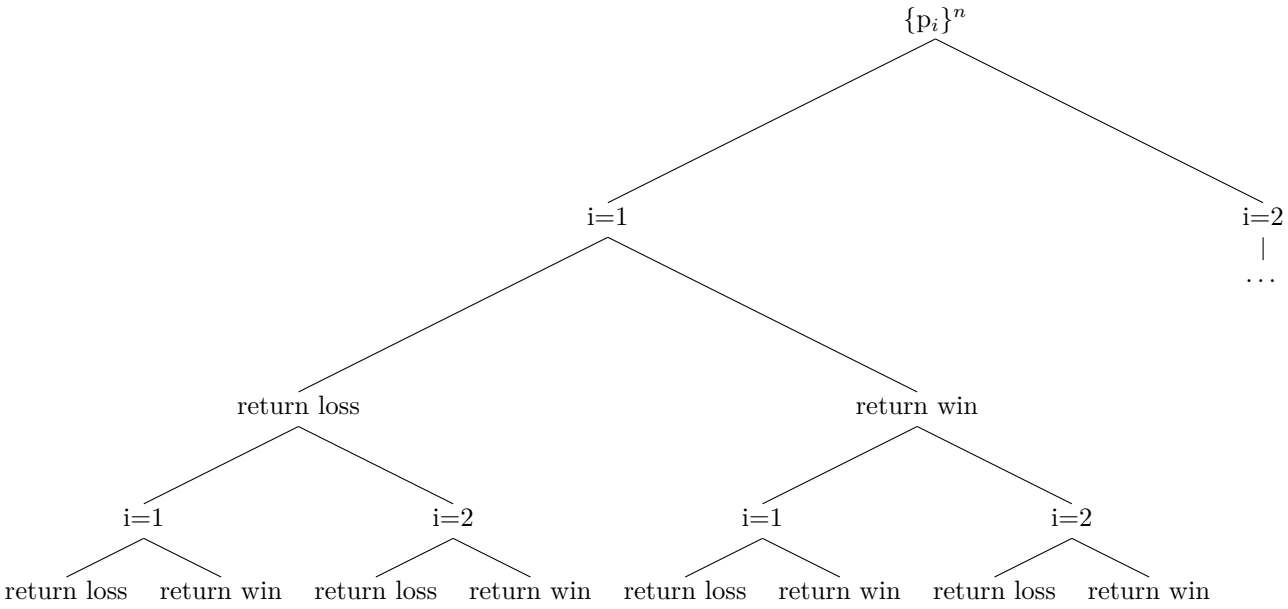


Figure 3.2: The game tree representation after player 2 took their turn picking $\{p_i\}$. This example is with $a = 2$; a Best-Arm example.

their choices in $C_{x\in u}$ should all be the same: C_u. There is no more interaction with the opponent after he has picked $\{p_{i,j}\}$, except for the roll-out of the chance moves. Because this is already incorporated in the chance moves, there is no need to account for this in the information sets, so we drop that. Therefore the choices in node x are $C_x \in \{1, \ldots, a\}$ (We switch to a $\{1, \ldots, a\}$ notation from $\{(1,1), \ldots, (2,2)\}$ for convenience in notation. Furthermore, the dimensionality is not important, except for the recommendation).

We denote a strategy that decides which actions to take for each specific node x^k by π^k, where k ranges over the players. π^1 is a single value, denoting which vector of $\{p_{i,j}\}$ is picked by player 1, but π^2 is a vector with an entry for every node. We call vectors π^2 of this form pure strategies. The set P^k is the set of all the available pure strategies for player k. To play the game, we allow the players to place weights (summing to 1, nonnegative) on each of these pure strategies, creating mixed strategies μ^k. The expected pay-off of a pair of mixed strategies $\mu = (\mu^1, \mu^2)$ is $H(\mu)$. For any node x, let $Pr_{\mu^1}(x)$ be the total μ^2 weight of all strategies π^1 prescribing exactly the moves for player 1 along the path to x and similarly $Pr_{\mu^2}(x)$. Let $\beta(x)$ be the probability of all chance moves along x. To find $H(\mu)$, we multiply the probability of reaching node x given μ and chance

moves β as $Pr_\mu(x)$. $\beta(x)$ then is the product of all the chance moves along the way to node x. The expected payoff then is $H(\mu) = \sum_{x^T} Pr_{\mu^1}(x) Pr_{\mu^2}(x)\beta(x)h(x) = \sum_{x^T} Pr_\mu(x)h(x)$, where x^T are all the terminal nodes in the tree and $h(x)$ the pay-off in node x.

3.4 Enumerating all algorithms

Using the above definition of the game tree, the goal is to find a mixed strategy μ^1 that minimizes the expected regret (expected because of the randomness incurred by both the chance moves and mixed strategy from the opponent) at the end of the game. As this is a zero-sum game, we can solve this with a Linear Programming solver [15]. It is easy to see however that enumerating all possible strategies π^2 grows exponentially with the size of the tree. The amount of nodes in turn grows exponentially in the level of the tree. For a budget T, without the nature move, the tree has $\sum_{i=0}^{T}(2a)^i$ nodes. $2a$ because each node has a choices, which each can return a win or a loss, resulting in $2a$ new nodes. The amount of different algorithms possible for the tree then becomes $a^{\sum_{i=0}^{T-1}(2a)^i} 2^{2a^T}$ (as the recommendation only has two choices, not a), which becomes unfeasible at T as low as $T = 3$ with $a = 2$.

Of course, the representation can be made more compact. If at x_1, $C_{x_1} = 1$ is chosen as action, the entire tree originating from the other actions in C_{x_1} will not be visited. It is useless to enumerate all the different choices in nodes that will never be visited. From every node, one action can be chosen, which in the chance node resulting from it, produces two child nodes, one with a success, one with a failure. So for each level in the tree, the amount of nodes doubles. The amount of nodes then visited is $\sum_{i=0}^{T} 2^i = 2^{T+1} - 1$. This leaves us with $a^{2^{T+1}-1}$ different algorithms, which is still too big (See Table 3.1). More simplifications will be made further on, but first we will cover how to calculate the optimal mixed strategies μ.

		a	
		2	4
T	1	8	16
	2	128	1024
	3	32,768	4,194,304
	4	2^{32}	7.04 10^{13}

Table 3.1: Number of possible strategies (or approximation) for some values of a and T.

3.5 Linear problem

We now introduce the Linear problem that finds the optimal strategy and its parametrization, so we can use these definitions in the upcoming Sections.

The opponent has to pick $\{p_{i,j}\}$. Because of size constraints, we discretize the available values for $\{p_{i,j}\}$ such that the opponent has the option to pick from m different combinations of $\{p_{i,j}\}$. This makes it so that there are finitely many pure strategies for the opponent. Because the opponent has the option to pick a mixed strategy, we can see the weights placed on each $\{p_{i,j}\}$ as a probability of picking that value. We therefore define the nonnegative weights as $\boldsymbol{z} = (z_1, \ldots, z_m)^T$ with the probability restriction $\sum_{i=1}^{m} z_i = 1$. Similarly, the algorithm has n different strategies to pick from,

with the following nonnegative weights: $\boldsymbol{y} = (y_1, \ldots, y_n)^T$ with $\sum_{i=1}^{n} y_i = 1$. Let $\boldsymbol{A}_{v,w}$ be the Expected regret $H(\pi_v^1, \pi_w^2)$ of the pure strategies v and w. For the linear constraints on $\boldsymbol{z}$ and $\boldsymbol{y}$, we define $\boldsymbol{Ex} = e$ with $\boldsymbol{E}$ as a 1 x m matrix of 1's and e as the scalar 1. Similarly, $\boldsymbol{Fy} = f$ with $\boldsymbol{F}$ as a 1 x n matrix of 1's and f as the scalar 1. Furthermore, let p and q range over scalars. The algorithm tries to find $\text{argmin}_{\boldsymbol{y}} \max_{\boldsymbol{z}} \boldsymbol{z^T A y}$ while simultaneously the opponent looks for $\text{argmax}_{\boldsymbol{z}} \min_{\boldsymbol{y}} \boldsymbol{z^T A y}$. With this equilibrium, according to Wilson (1972) we end up with the following linear problem:

$$\begin{aligned} \underset{\boldsymbol{y},p}{\text{minimize}} \quad & e^T p \\ \text{subject to} \quad & -\boldsymbol{Ay} + \boldsymbol{E}^T p \geq 0, \\ & -\boldsymbol{Fy} = -f, \\ & \boldsymbol{y} \geq 0. \end{aligned} \tag{3.4}$$

The dual according to Wilson (1972) is:

$$\begin{aligned} \underset{\boldsymbol{z},q}{\text{maximize}} \quad & -q^T f \\ \text{subject to} \quad & \boldsymbol{z}^T(-\boldsymbol{A}) - q^T \boldsymbol{F} \leq 0, \\ & \boldsymbol{z}^T \boldsymbol{E} = e, \\ & \boldsymbol{z} \geq 0. \end{aligned} \tag{3.5}$$

The solution for this is easily found numerically, but will not be discussed yet, as there are more simplifications to be done.

3.6 Realization weights

The next step [8] is instead of defining an algorithm which describes the actions taken at each node, and then calculating the optimal weights on each algorithm to take, we place the weights on the different choices C_x for each node in the tree. This way the representation becomes a lot more compact. Instead of $a^{2^{T+1}-1}$ the number of weights $\boldsymbol{y}$ becomes $a\sum_{i=0}^{T}(2a)^i$. This is one less exponent than the previous representation. The hierarchy between the nodes in the tree needs to be specified. Implicitly this is done by changing the restriction $\sum_{i=1}^{n} y_i = 1$ to $y_{x_t} = \sum_{c\in(1,\ldots,a)} y_{x_{t+1,c}}$, where y is the weight of node x_t, where x_t is a node x after t of the T samples have been used. $x_{t+1,c}$ is the node resulting from x_t after sampling move c. The intuition behind this is that y_{x_t} is the chance of the algorithm reaching node x_t when multiplied with $\beta(x_t)$, which is incorporated in the pay-off matrix $\boldsymbol{A}$, the product of the chance moves (in this case sample draws) passed on the way from the root to x_t. Intuitively, the probabilities for each of the possible actions C_{x_t} to be taken from x_t should again sum to y_{x_t}. Therefore $y_{x_t} - \sum_{j=1}^{a} x_{t+1,j} = 0 \;\forall\; x_t \in \{1 : \sum_{l=0}^{t}(2a)^l\}$. These added restrictions are added in the matrix $\boldsymbol{F}$ turning it into a n x an matrix where n is the number of nodes $\sum_{l=0}^{t}(2a)^l$:

$$\boldsymbol{F} = \begin{bmatrix} 1 & 1 & & & & & & & \dots \\ -1 & & 1 & 1 & & & & & \dots \\ -1 & & & & 1 & 1 & & & \dots \\ & -1 & & & & & 1 & 1 & \dots \\ & -1 & & & & & & & \dots \\ & & -1 & & & & & & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad \boldsymbol{f} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

With only non-zero entries displayed. On the rows are the nodes x and on the columns the actions for each node C_x. The corresponding vector $\boldsymbol{f}$ then becomes $\boldsymbol{f} = (1, 0, 0, \dots)^T$. The pay-off matrix $\boldsymbol{A}$ is redefined as the product between the probabilities of the chance moves $\beta(x)$ and the pay-off at that node. Because there is no pay-off until the terminal nodes, all row entries of $\boldsymbol{A}$ corresponding $x_{t,t\neq T}$ are 0.

3.7 Sufficient statistic

The simplification provided above has one property, which is something called perfect recall. This implies that the players know, remember and act according to all the previous actions. This would mean that if action 1 was picked twice, once returning a failure, once returning a success, the order in which those two happened matters (or might matter sometimes) for the action picked from the resulting node. Or at least that a separate variable is made to reflect this difference. Each node x also has some sufficient statistic [3], representing the results from the previous samples. This is a vector of length $2a$: $v(x) = (successes_1, failures_1, \dots, successes_a, failures_a)$.

One more simplification that can yet be made is to disregard the order in which the previous samples leading to node x were taken, and allow branches of the tree to rejoin together if $v(x) = v(x')$. The change made to the linear restrictions on the variables is as follows: $\sum_q x_{i,q} - \sum_{j=1}^{a} x_{i,j} = 0 \ \forall$ $i \in \{1 : \sum_{l=0}^{t} (2a)^l\}$ where $x_{i,q}$ are all the actions from other nodes that can result in reaching x_i in the game tree. An example of two nodes having the same sufficient statistic is shown in Figure 3.3.

In the next section we will prove that this simplification will still result in an optimal solution.

3.8 Validity of the Sufficient Statistic

In Section 3.7 we presented a smaller version of the problem in Section 3.5. In this Section we show that they will give equivalent solutions.

Theorem 1. *The optimal solution from solving the reduced problem in Section 3.7 can be used to find an optimal solution in the complete problem in Section 3.5.*

Proof. Let x and y be two nodes in the above defined game tree where $v(x) = v(y)$. Let M be the Linear problem described in Section 3.5. Let $\boldsymbol{S}$ be an optimal solution for M. Similarly, let M' be the Linear problem described in Section 3.7. Let $\boldsymbol{S'}$ be an optimal solution for M'. Let x_a and x_b be the variables in $\boldsymbol{S}$ corresponding with the weights on the choices C_x, and similarly y_a and y_b corresponding to C_y. In the sufficient statistic model M' adds the weights together $x + y = xy$, $x_a + y_a = xy_a$ and $x_b + y_b = xy_b$.

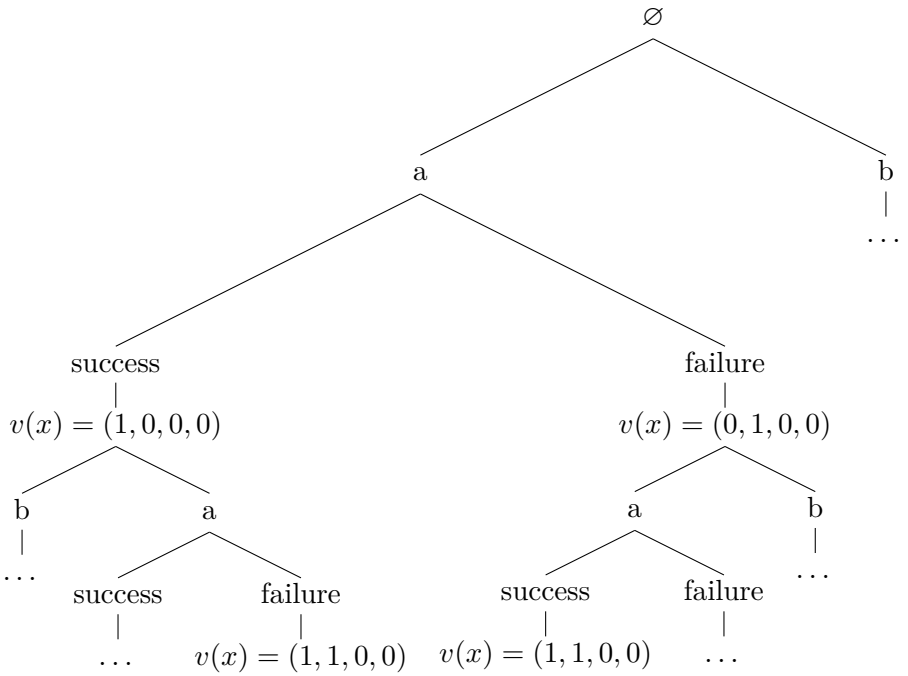


Figure 3.3: An example of two nodes in the tree sharing the same sufficient statistic. As the tree gets deeper, these nodes become more and more common.

$\boldsymbol{S}$ is an optimal solution for M, therefore satisfies the following restrictions:

$$\begin{array}{ccccccc} x & & & = & x_a + x_b & & \\ & & y & = & & & y_a + y_b \\ x & + & y & = & x_a + x_b & + & y_a + y_b \end{array}$$

Therefore, $\boldsymbol{S}$ is a solution $\exists s' \in M'$.

The payoff determining the value of the solution $\boldsymbol{S}$ depends only on the sufficient statistic $v(x)$ of all the nodes x. Therefore $v(x) = v(y) = (xy) \implies h(x) = h(y) = h(xy)$. The expected regret, taking the weights into account, then becomes $h(x_a)x_a + h(y_a)y_a = h(xy_a)xy_a$. This means that the payoff of the optimal solution $h(\boldsymbol{S}) = h(s' \in M') \implies h(\boldsymbol{S}) \geq h(\boldsymbol{S'})$.

Similarly, $\boldsymbol{S'}$ is an optimal solution for M', therefore satisfies the following restrictions:

$$\begin{array}{ccccc} xy & = & xy_a & + & xy_b \\ x + y & = & x_a + x_b & + & y_a + y_b \end{array}$$

There are many solutions that satisfy the restrictions of M, but dividing xy_a and xy_b over x_a, x_b, y_a and y_b according to the ratio $\frac{x}{y}$ always will yield a valid solution:

$$\begin{array}{ccccccc} x_a & & & = & \frac{x}{x+y}xy_a & & \\ & & y_a & = & & & \frac{y}{x+y}xy_a \\ x_a & + & y_a & = & \frac{x}{x+y}xy_a & + & \frac{y}{x+y}xy_a \\ x_a & + & y_a & = & \frac{x+y}{x+y}xy_a & & \\ x_a & + & y_a & = & xy_a & & \end{array}$$

And

$$\begin{array}{lllll} x_a & & & = & \frac{x}{x+y} x y_a \\ & & x_b & = & & & \frac{x}{x+y} x y_b \\ x_a & + & x_b & = & \frac{x}{x+y} x y_a & + & \frac{x}{x+y} x y_b \\ & & & = & \frac{x}{x+y}(x y_a + x y_b) \\ & & & = & \frac{x(x+y)}{x+y} \\ & & & = & x \end{array}$$

Using this, we showed that $\boldsymbol{S'}$ has a corresponding solution $s\exists \in M$. With similar arguments as above, we can see that $h(\boldsymbol{S'}) = h(s \in M) \implies h(\boldsymbol{S'}) \geq h(\boldsymbol{S})$, which along with the earlier statement $h(\boldsymbol{S}) \geq h(\boldsymbol{S'})$ means that $h(\boldsymbol{S'}) = h(\boldsymbol{S}) \implies \boldsymbol{S'} \equiv \boldsymbol{S}$.

□

3.9 Results

Calculations were performed in R, using the package `Rglpk`.

We first used the model described in Section 3.7 on a Best-Arm problem with $a = 2$ and a budget of $T = 20$. The results can be seen in Figures 3.4 and 3.5. One thing to remark is that the strategy for all but a few nodes is entirely placed on one of the two arms, instead of some distribution between the two. This means that, even though the algorithm has the option to define some weights on the choices C_x, it will place all the weights on one action in most nodes.

The goal of this method was to try to find some pattern in the sampling rules of the optimal algorithm. However, looking at Figure 3.5, there is no clear pattern or rule to extract from these results. The results plotted in this Figure are even just from a 2 armed Best-arm setting. The interpretation of the results from a mini-max optimal algorithm with $a = 4$ would be very hard, if not for the fact that it is hard to visualize this.

Looking at the performance of the optimal algorithm (Table 3.2), we see that in the worst-case scenario, the optimal algorithm is slightly better, but it pays for that for other sets of parameters. It is very likely however that the slight improvement is caused by numerical noise. Another remark is that the algorithm is not symmetrical. If we were to exchange the observations between arm 1 and arm 2, we would expect the algorithm's action to change as well. This is not the case, so it seems that there are more than one optimal solutions. The Linear Program solver just picked one. To fix this we force the symmetry between arm 1 and arm 2, by adding constraints of the form: if node x_1 and x_2 are symmetrical, the weights on the choices of move x_1 are equal to their equivalents of x_2. The results of this are shown in Figure 3.4. The worst-case set parameters $\{p_i\}$ is shown in Table 3.3.

The performances of the Optimal algorithm in the Min-max setting does not have the same problems of barely being able to improve on sampling equally as $a = 2$ has (Figure 3.6). With more arms to sample from, it is has more to win by not sampling each arm equally. Compared to the Best-arm results, the performances are worse as a increases, but that is to be expected, as there are more arms to sample from, with the same budget. This means less samples per arm, so it becomes harder to reach a good conclusion.

Sadly the calculation of the Optimal algorithm becomes too computationally intensive as T increases, leaving us with an algorithm for a maximum budget of $T = 6$ in the simplest Min-max scenario.

	Worst-case Optimal algorithm	Equals algorithm
$\max \mathbb{E}(R)$	0.03815815	0.03818470
$Mean(\mathbb{E}(R))$	0.01681627	0.01465692

Table 3.2: Performance of the Worst-case Optimal Algorithm in $a = 2$ and $T = 20$. The performance was compared to the Equal-algorithm adapted for $a = 2$ (Section 2.3). As can be seen in the top row, in the worst-case set $\{p_{i,j}\}$ the Worst-case Optimal Algorithm performs slightly better. When we look at the average performance over a uniform grid of $\{p_{i,j}\}$, we see that it is worse. The Worst-case Optimal Algorithm is indeed better in the worst-case, but pays for that in the rest of the parameter-space.

T	p_1	p_2	Weight
2	0.0000000	0.5102041	0.3310811
	0.4897959	1.0000000	0.2635135
	0.7959184	0.2857143	0.4054054
5	0.3061224	0.0000000	1.454700e-17
	0.3469388	0.6530612	6.530612e-01
	0.6530612	0.3469388	3.469388e-01
10	0.0000000	0.2244898	2.947623e-05
	0.3673469	0.6122449	2.458179e-01
	0.3877551	0.6326531	2.533630e-01
	0.6122449	0.3673469	2.541521e-01
	0.6326531	0.3877551	2.466070e-01
	1.0000000	0.7959184	3.051906e-05
15	0.4081633	0.5918367	0.4081633
	0.5918367	0.4081633	0.5918367
20	0.4081633	0.5714286	4.691306e-01
	0.4285714	0.5918367	5.308692e-01
	1.0000000	0.8775510	1.644140e-07

Table 3.3: Worst-case $\{p_{i,j}\}$ along with their weights for the Worst-case Optimal Algorithm with $a = 2$.

This is not sufficient to use in practice. We can however extrapolate the performances as a base of comparison for other algorithms. Additionally we do get an intuition on the worst case set of $\{p_{i,j}\}$.

In the Min-max $a = 4$ setting, the worst-case set $\{p_{i,j}\}$ could only be calculated for $T = 6$, at which point the differences δ can be relatively big without suffering the lower error-rate. The distribution on the worst-case set can be found in Table 3.4. This is an extreme case; the high difference between $min_j p_{1,j}$ and $min_j p_{2,j}$ is caused by the low budget, leaving the algorithm unable to differentiate between such big differences between the parameters.

T	$p_{1,1}$	$p_{1,2}$	$p_{2,1}$	$p_{2,2}$	Weight
2	0.0000000	0.5000000	0.5000000	0.5000000	0.2424640
	0.5000000	0.5000000	0.0000000	0.5000000	0.2870249
	0.5000000	0.5000000	0.5000000	0.0000000	0.1847969
	0.8333333	0.3333333	0.8333333	0.8333333	0.2123198
	1.0000000	1.0000000	1.0000000	0.5000000	0.0733945
5	0.0000000	0.3333333	1.0000000	0.3333333	0.002875250
	0.1666667	0.6666667	0.5000000	0.5000000	0.016654876
	0.3333333	0.8333333	0.6666667	0.6666667	0.190211561
	0.3333333	1.0000000	0.0000000	0.3333333	0.004388016
	0.5000000	0.5000000	0.6666667	0.1666667	0.093909648
	0.6666667	0.1666667	0.6666667	0.6666667	0.031962007
	0.6666667	0.6666667	0.3333333	0.8333333	0.176373542
	0.6666667	0.6666667	0.8333333	0.3333333	0.050641718
	0.6666667	1.0000000	1.0000000	1.0000000	0.068342128
	0.8333333	0.3333333	0.6666667	0.6666667	0.169562899
	0.8333333	0.8333333	0.5000000	1.0000000	0.064617805
	0.8333333	0.8333333	1.0000000	0.5000000	0.111302297
	1.0000000	0.6666667	1.0000000	1.0000000	0.016941423
	1.0000000	1.0000000	1.0000000	0.6666667	0.002216828
6	0.1666667	0.6666667	0.5000000	0.5000000	0.086782237
	0.3333333	0.0000000	0.3333333	0.3333333	0.003221396
	0.3333333	0.0000000	0.3333333	1.0000000	0.003311834
	0.3333333	0.8333333	0.6666667	0.6666667	0.124142292
	0.3333333	1.0000000	0.3333333	0.0000000	0.006076984
	0.5000000	0.1666667	0.5000000	0.5000000	0.006250593
	0.5000000	0.5000000	0.1666667	0.6666667	0.086337326
	0.5000000	0.5000000	0.6666667	0.1666667	0.064793146
	0.5000000	1.0000000	0.8333333	0.8333333	0.044011895
	0.6666667	0.1666667	0.5000000	0.5000000	0.051353397
	0.6666667	0.6666667	0.3333333	0.8333333	0.126594455
	0.6666667	0.6666667	0.8333333	0.3333333	0.156756307
	0.8333333	0.3333333	0.6666667	0.6666667	0.166241956
	0.8333333	0.8333333	0.5000000	1.0000000	0.036764226
	1.0000000	0.3333333	0.3333333	0.0000000	0.002765150
	1.0000000	0.6666667	1.0000000	1.0000000	0.010733693
	1.0000000	1.0000000	1.0000000	0.6666667	0.023863113

Table 3.4: Worst-case $\{p_{i,j}\}$ along with their weights for the Worst-case Optimal Algorithm with $a = 4$.

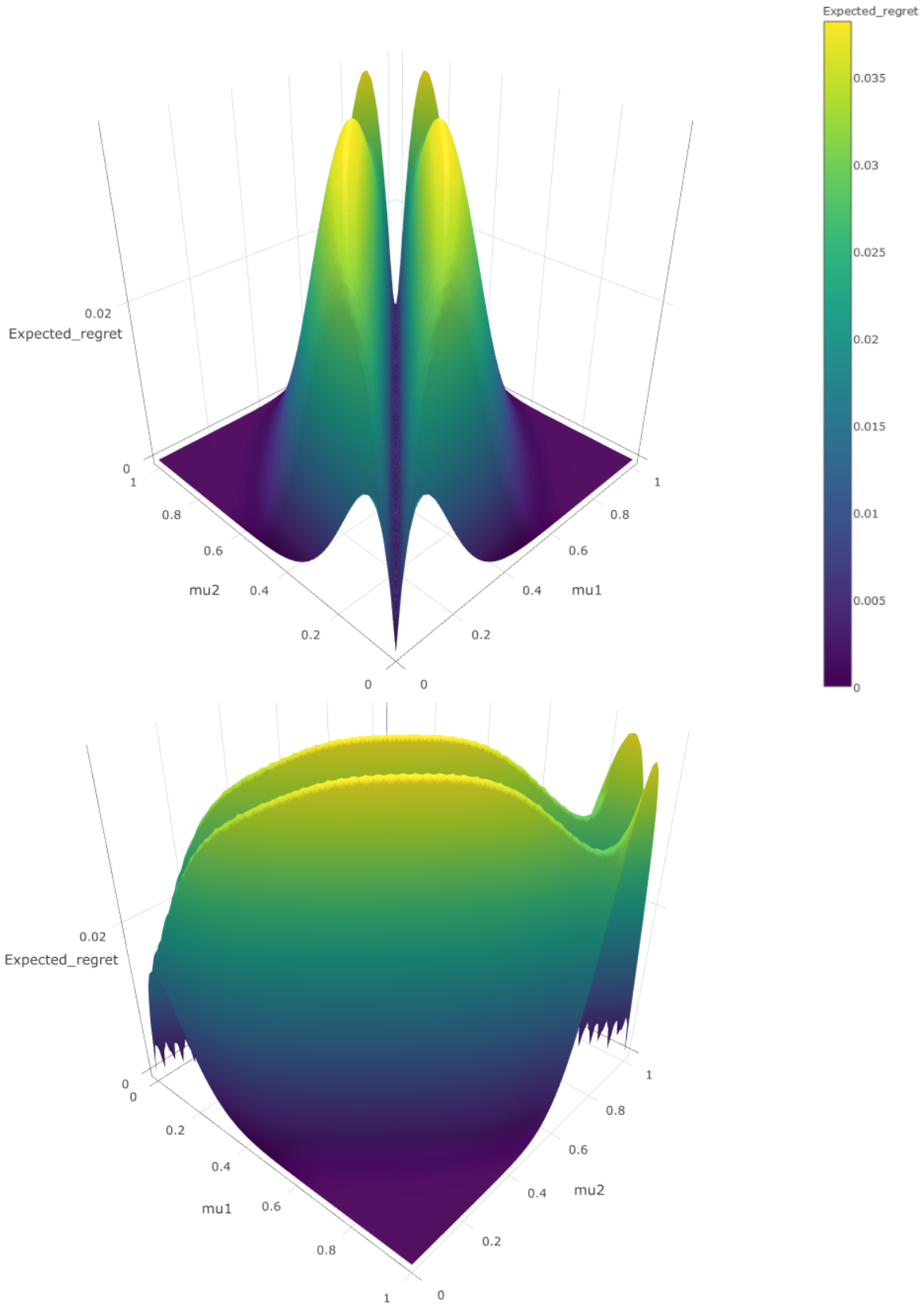


Figure 3.4: Expected regret plots (viewed from two perspectives) of the optimal algorithm for a two-arm Best-Arm Identification problem with a budget of $T = 20$. Compare this to the Expected regret by the Equal-algorithm with same budget in Figure 2.1.

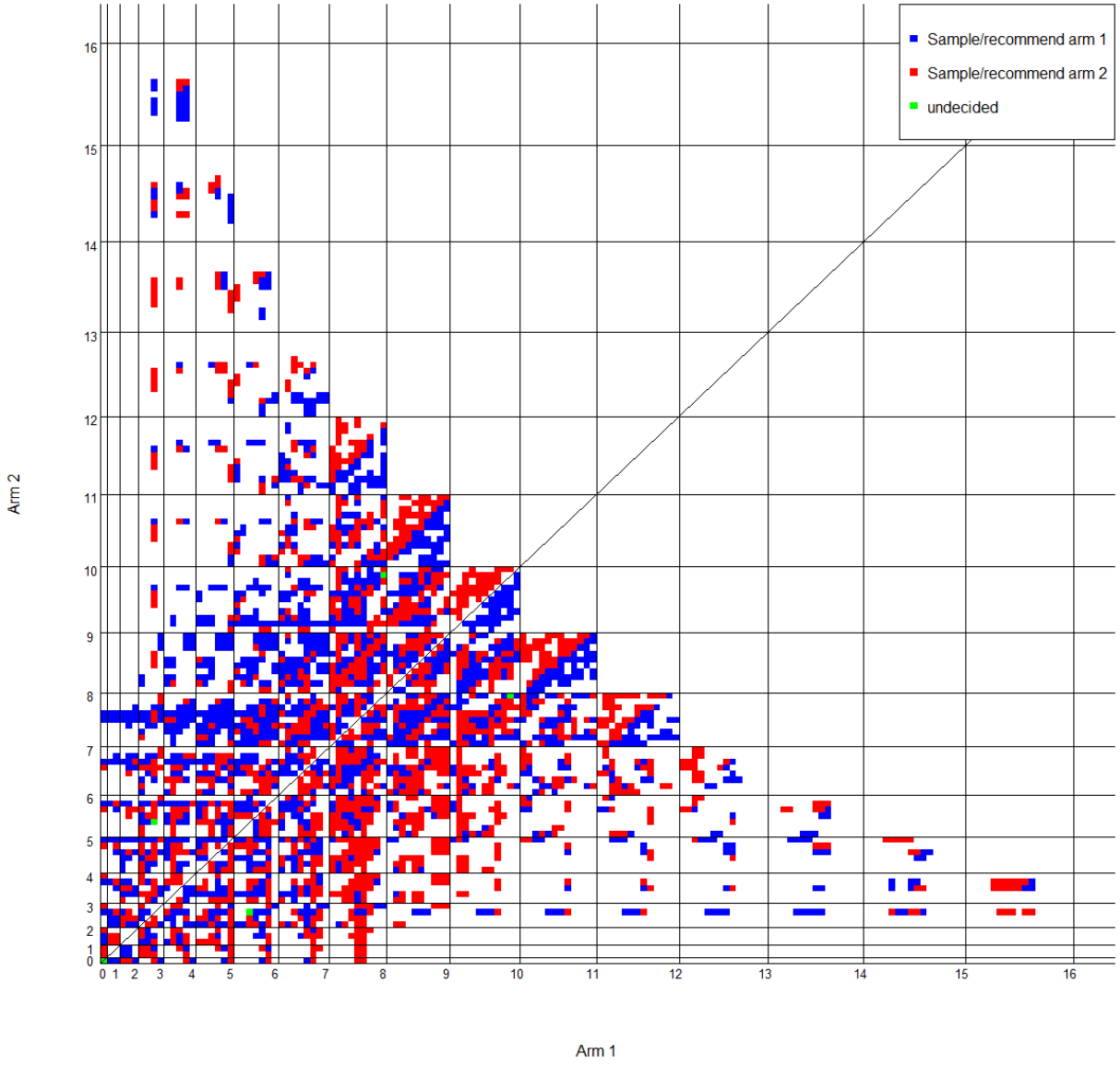


Figure 3.5: Strategy representation of the optimal algorithm for $a = 2$ and $T = 20$. The pixels represent nodes as follows: The boxes represent the samples spent in that arm, from 0 to T in both directions. Within the boxes, each pixel corresponds to the amount of successes obtained in that arm: from 0 to n in both directions. The color of the pixel shows the strategy within that node.

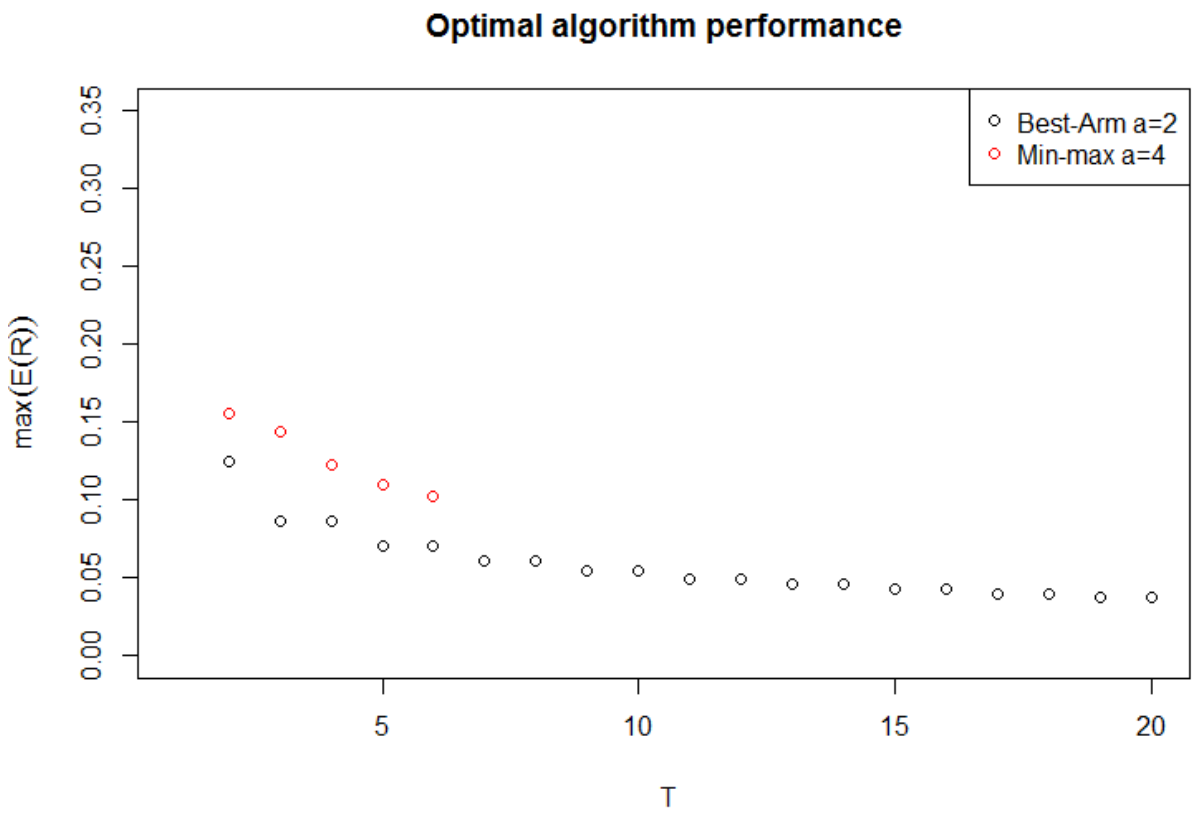


Figure 3.6: Performances of the Optimal algorithm in the worst-case scenarios. Due to computer time, results for higher T are not available. The results for $a = 2$ are in the Best-Arm setting, while the other results are Min-max.

4

The Bayesian algorithm

Our search for the optimal algorithm in Section 3 tried finding the optimal algorithm in the worst case scenario, creating a mini-max game in itself. We found that not only does this game grow exponentially in size, it also grows greatly in computation time, practically limiting us to $T = 6$. If we relax our aim to find an algorithm optimal in the worst case scenario and instead look at optimizing the expectation over all $\{p_{i,j}\}$ under some computationally convenient (and reasonable) prior distribution, the objective becomes easier to compute. We hope that an algorithm that optimizes the expectation, is a decent alternative for one that optimizes the worst case scenario.

The expectation over all $\{p_{i,j}\}$ requires some distribution for those parameters. Recall Equation 3.3 (repeated as 4.1), which maximizes the Regret with respect to some distribution Q on $\{p_{i,j}\}$.

$$\max_{Q} \min_{\substack{\text{Algorithm strategy} \\ (pure)}} \mathop{\mathbb{E}}_{\{p_{i,j}\}\sim Q} \mathop{\mathbb{E}}_{\substack{Samples \\ Recommendation}} R(I) \tag{4.1}$$

Same as Equation 3.3.

If we fix the distribution Q, it will formally take the role of a Bayesian prior on $\{p_{i,j}\}$. The objective then becomes:

$$\min_{\substack{\text{Algorithm strategy} \\ (pure)}} \mathop{\mathbb{E}}_{\{p_{i,j}\}\sim Q} \mathop{\mathbb{E}}_{\substack{Samples \\ Recommendation}} R(I) \tag{4.2}$$

It makes sense to approach this in a Bayesian framework, basing the decisions on the expectation of the regret conditional to a posterior distribution on $\{p_{i,j}\}$. First we will define the priors and decision rules. Then we present the results for comparison in Section 5. While the Algorithm is Bayesian in spirit and construction, the evaluation will still be based on the worst-case Regret, in line with the other algorithms.

4.1 The Prior and Posterior distributions

For numerical convenience we use as Q the fourth power of the conjugate prior for the Bernoulli distribution [11], one prior distribution for each arm in $\{p_{i,j}\}$. This is the Beta distribution. We pick the prior to be uninformative, with parameters a_{prior} and b_{prior}. The posterior distribution is again a Beta distribution, with $a_{pos} = Num.wins + a_{prior}$ and $b_{pos} = Num.losses + b_{prior}$. To test whether the prior distribution seriously affects the algorithm, we tested its performance with three different priors: Beta(0.5, 0.5), Beta(1, 1) and Beta(1.5, 1.5) (Figure 4.1). Beta(0.5, 0.5) is a commonly used prior for binomial parameters, yielding an uninformative distribution that is invariant for all transformations of the parameter [6]. Beta(1, 1) is equal to Unif(0, 1), and as such also uninformative. Finally, Beta(1.5, 1.5) also has mean 0.5, but places more weight around the mean, which is where we would expect the worst-case parameters to be, based on results in Sections 2.3 and 3.9. This way we make the algorithm consider those parameters to be more realistic. The final prior is slightly informative.

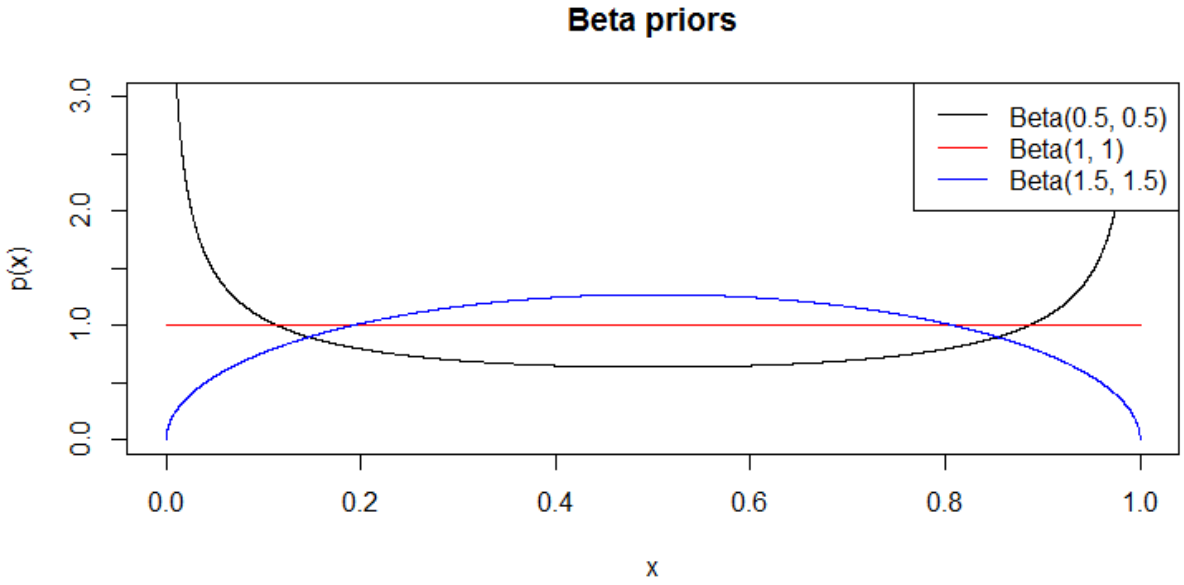


Figure 4.1: Beta prior distributions

The posterior distribution can easily be calculated based on the sufficient statistic value of a node $v(x)$ (Section 3.7). Conditional on this posterior, the expectation on getting a win or a loss is respectively $\frac{a_{pos}}{a_{pos}+b_{pos}}$ and $\frac{b_{pos}}{a_{pos}+b_{pos}}$.

4.2 $\mathbb{E}_{\{p_{i,j}\}}(R; x)$

The final recommendation made by most algorithms presented in Section 5 is based on the MLE of the parameters $\widehat{\{p_{i,j}\}}$ and with that an estimate of the Regret, while the Bayesian framework rather looks at the posterior distribution of those parameters: $\{p_{i,j}\} \sim Q_{pos}$. This means that there is not just one value, but again a distribution of the regret. So preferably we would like to take the expectation of this to make our decisions.

$$\mathop{\mathbb{E}}_{\{p_{i,j}\}}(R_I;x) = \iiiint (\max_i \min_j p_{i,j} - \min_j p_{I,j}) P(\{p_{i,j}\}\,|v(x))\ \mathrm{d}p_{1,1}\ \mathrm{d}p_{1,2}\ \mathrm{d}p_{2,1}\ \mathrm{d}p_{2,2} \tag{4.3}$$

Where x is a terminal node in the game tree (Section 3.3).

The goal is to minimize the regret with respect to I:

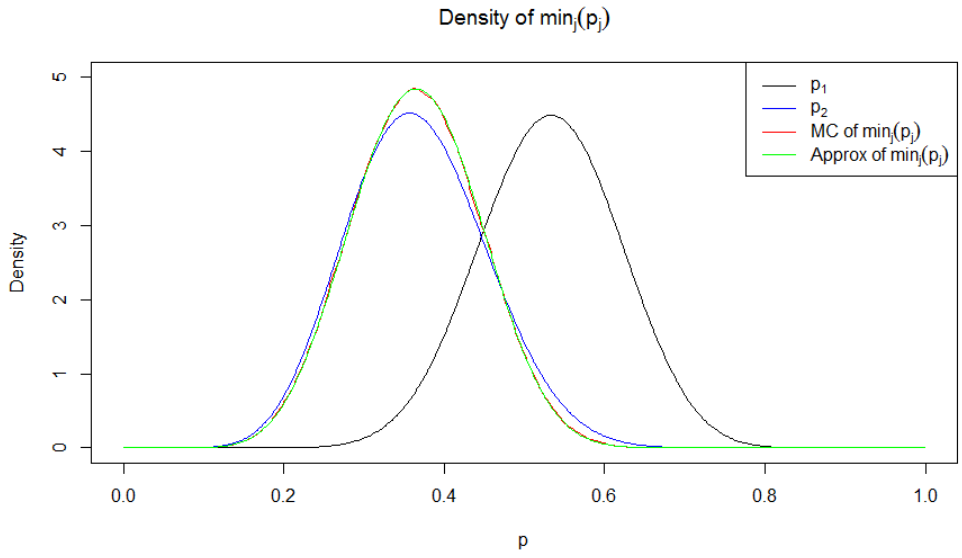
$$\operatorname*{argmin}_I \mathop{\mathbb{E}}_{\{p_{i,j}\}}(R_I;x) \tag{4.4}$$

Because the first term in the integral does not depend on I:

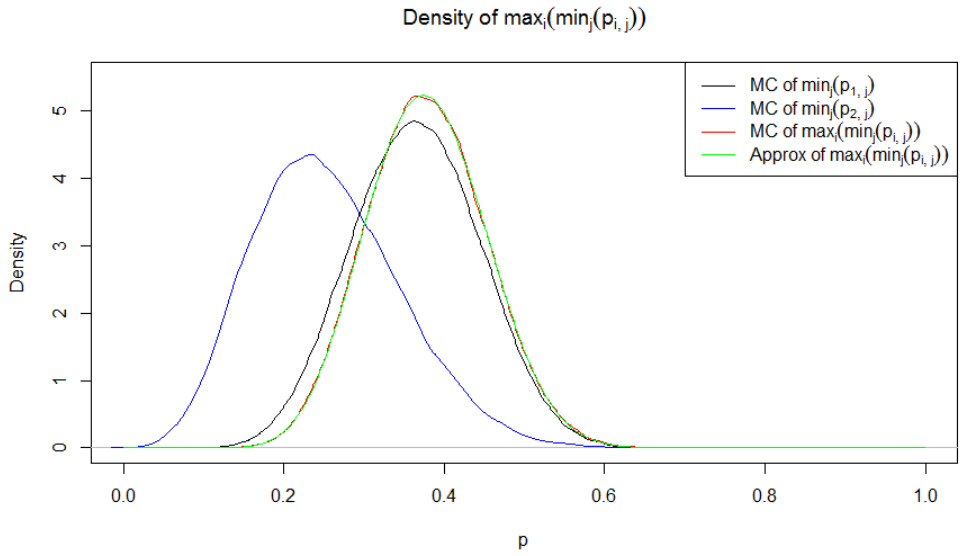
$$\operatorname*{argmax}_I \iint (\min_j p_{I,j}) P(\{p_{i,j}\}\,|v(x))\ \mathrm{d}p_{I,1}\ \mathrm{d}p_{I,2} \tag{4.5}$$

To find a strategy, the integral in Equation 4.3 has to be calculated for each terminal node in the tree (Figure 4.3). This is a huge fraction of all nodes in the tree, so this calculation takes a lot of time. Therefore we approach the integrals with a four-dimensional grid on $\{p_{i,j}\}$, summing over the $\min_j p_{I,j} P(\{p_{i,j}\}\,|v(x))$ for each four-dimensional combination of grid-points, so we can calculate $\mathbb{E}_{\{p_{i,j}\}}(R_I;x)$ in linear time with the one-dimensional grid size.

To show that our approximation of Equation 4.3 yields representable results, we compared its outcomes with the $\mathbb{E}_{\{p_{i,j}\}}(R_I)$ calculated off a Monte Carlo sample ($n = 10^5$) of $\{p_{i,j}\}$ generated from the Beta posterior distributions given the $v(x)$. The results of this are shown in Figure 4.2. In Figure 4.2a we show the distribution of $\min_j p_j$, which makes up part of the integral. For each value of i, these plots are then shown in 4.2b, along with the density of the pairwise maxima. This is again compared to the grid-wise approximation of the density. As can be seen, on a grid of 500 points, this yields a decent approximation, while saving a lot on computation. We find a grid of 500 points to be accurate while still being fast to compute.



(a)



(b)

Figure 4.2: The estimated density of $\min_j p_j$ (a) and $\max_i \min_j p_{i,j}$ (b) evaluated on a grid of 500 values for p. Comparison shown is for the node with sufficient statistic $v(x) = \{17, 15, 11, 19, 5, 15, 2, 1\}$. The results are compared to the results of a Monte Carlo sample of $\{p_{i,j}\}$ drawn from the posterior Beta distributions on this node. $n = 10^5$. The overlap between the approximation and the Monte Carlo results show that the approximation is sufficient.

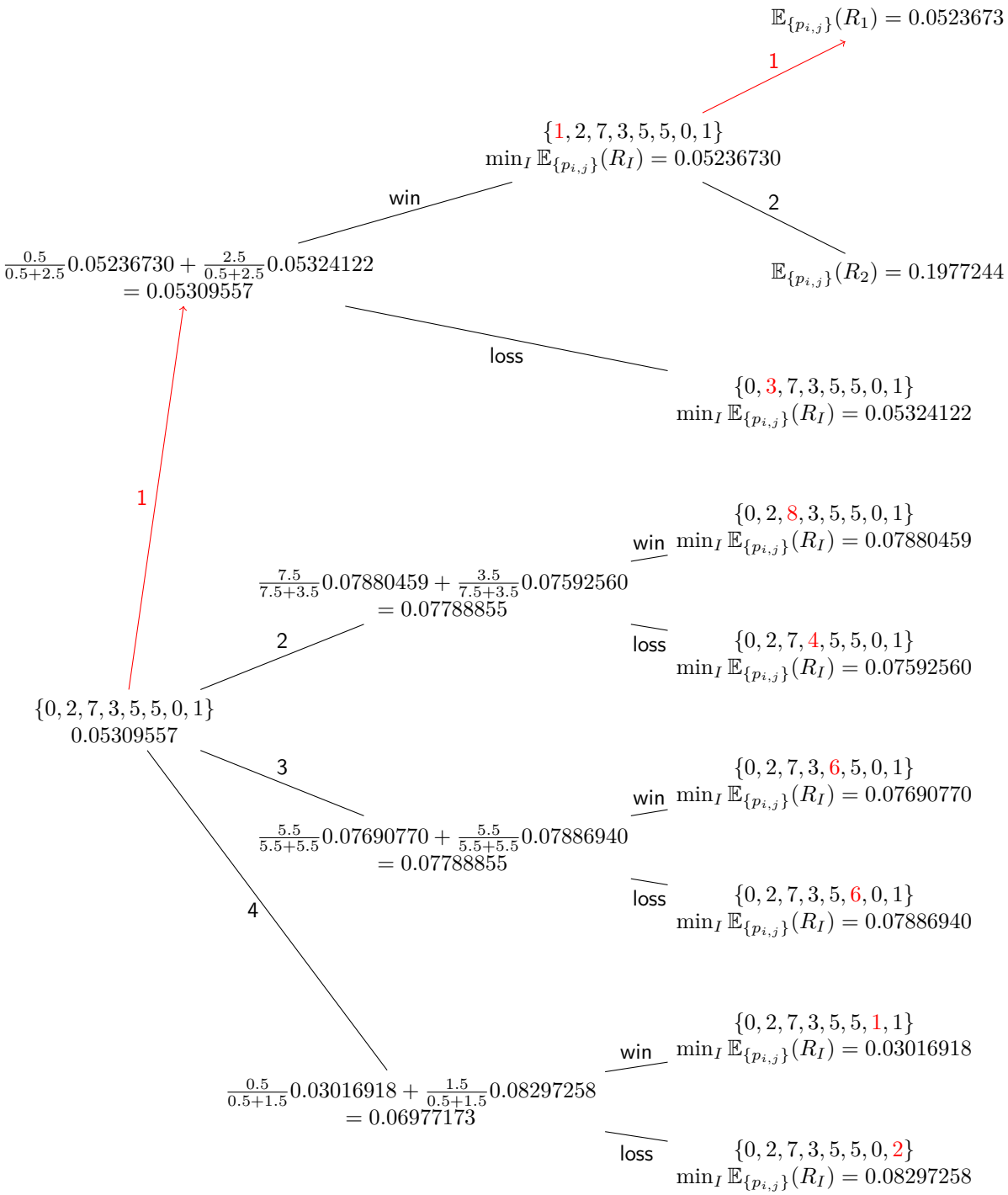


Figure 4.3: An example of the `someRecommendationRule` and `someSampleRule` for the Bayesian Expected regret algorithm. Shown on the left is one node ($\{0,2,7,3,5,5,0,1\}$). The algorithm has one sample left to spend, after which it has to make a recommendation. For each of the eight possible terminal nodes and their two possible recommendations, the expectation of the Regret is calculated according to Equation 4.3.

4.3 Bayesian strategy

When all the terminal nodes have their Expected regret calculated, the values for the rest of the nodes can be calculated in backward fashion. Each node has four arms to sample from, so the algorithm picks the arm which in expectation yields the lowest regret. In order to do that, we need to know the probability of getting a success or a loss from that sample. This can be calculated conditional to the posterior distribution on $\{p_{i,j}\}$ in that specific node:

$$\underset{i,j}{\operatorname{argmin}} \left[\frac{a_{pos}^{i,j}}{a_{pos}^{i,j} + b_{pos}^{i,j}} \underset{\{p_{i,j}\}}{\mathbb{E}}(R_{i,j}^{1}) + \frac{b_{pos}^{i,j}}{a_{pos}^{i,j} + b_{pos}^{i,j}} \underset{\{p_{i,j}\}}{\mathbb{E}}(R_{i,j}^{0}) \right] \tag{4.6}$$

Where i, j is the arm to sample from and $\mathbb{E}_{\{p_{i,j}\}}(R_{i,j}^{0/1})$ the Expected regret value of the resulting node after sampling from arm i, j and receiving a loss or a win respectively.

An example of this calculation is shown for a small part of the game tree in Figure 4.3: Within the node, the recommendation that minimizes this expectation is picked as the chosen recommendation for `someRecommendationRule`. For the topmost terminal node, this calculation is shown, for the other nodes only the value is shown. From these Expected regrets, the expectations on the regrets on the nodes leading up to these terminal nodes is calculated. Conditional on the posterior in the node, the probabilities of getting a win or a loss are used to calculate the expectation of the regret. With a Beta prior, these probabilities are convenient to calculate (Equation 4.6). In this example, the algorithm has the lowest Expected regret if it picks arm 1, so it picks arm 1.

4.4 Results

The strategy for this algorithm is way faster to calculate than the optimal algorithm because it does not need to consider all possible distributions $\{p_{i,j}\} \sim Q$, but only the fixed prior. However, here as well there are limitations. This downside with this algorithm is that, similarly to the optimal algorithm, the entire algorithm has to be calculated in advance. There is no straightforward way to calculate the next step on the fly. This means that the entire strategy has to be stored, which becomes hard as the budget grows. With all the simplifications mentioned in Section 3.7, the tree still becomes too big around $T \approx 30$.

The strategies for this algorithm were calculated in R, for T from 2 to 30, with different prior distributions for $\{p_{i,j}\}$. Then the performance was evaluated over a grid of $\{p_{i,j}\}$, using the worst-case regret. The results are shown in Figure 4.4. In this figure the results are compared to the Optimal algorithm presented in Section 3. Notable is the better performance of the Optimal algorithm. This is of course the promise of the algorithm. Sadly the computational burden of calculating the Optimal algorithm prevents us from comparing the Bayesian Expected Regret algorithm for more realistic T.

The different priors seem to perform similarly as T increases. This makes sense, as the smaller T is, the bigger the influence of the prior. This also explains the relatively big differences for lower T.

As mentioned, this strategy has the problem that it has to be calculated fully in advance as well, again creating a upper-bound on what can be calculated in reasonable time and space. However, it does give some additional information about hard cases of $\{p_{i,j}\}$ (not using the term 'worst-case', because this algorithm makes no such guarantees) and how they change as T increases. The results

Prior	$p_{1,1}$	$p_{1,2}$	$p_{2,1}$	$p_{2,2}$
Beta(0.5, 0.5)	0.5	1	0.65	0.65
Beta(1, 1)	0.43	1	0.57	0.57
Beta(1.5, 1.5)	0.5	1	0.65	0.65

Table 4.1: Worst-case parameters $p_{\{ij,\}}$ for the Bayesian Expected Regret algorithm at $T = 30$.

here confirm the intuition explained in Section 3.9: for $T = 30$, the highest Expected Regret as a result from the algorithm with priors Beta(0.5, 0.5) is from the set $\{p_{i,j}\} = \{0.65, 0.65, 1, 0.5\}$. Now it should be noted that this is based on a grid of 11 values for each $p_{i,j}$, resulting in $11^4 = 14641$ sets. This of course is not the densest grid, but it does give a decent picture of a hard case. The worst-case parameters are shown in Table 4.1. Because the indices i and j do not have a fixed order, the worst-case parameters have symmetries. $\{0.5, 1, 0.65, 0.65\}$ should be no harder than $\{1, 0.5, 0.65, 0.65\}$, $\{0.65, 0.65, 1, 0.5\}$ or $\{0.65, 0.65, 0.5, 1\}$. As can be seen in Table 4.1, the worst-case parameters for Beta(0.5, 0.5) are the same as for Beta(1.5, 1.5).

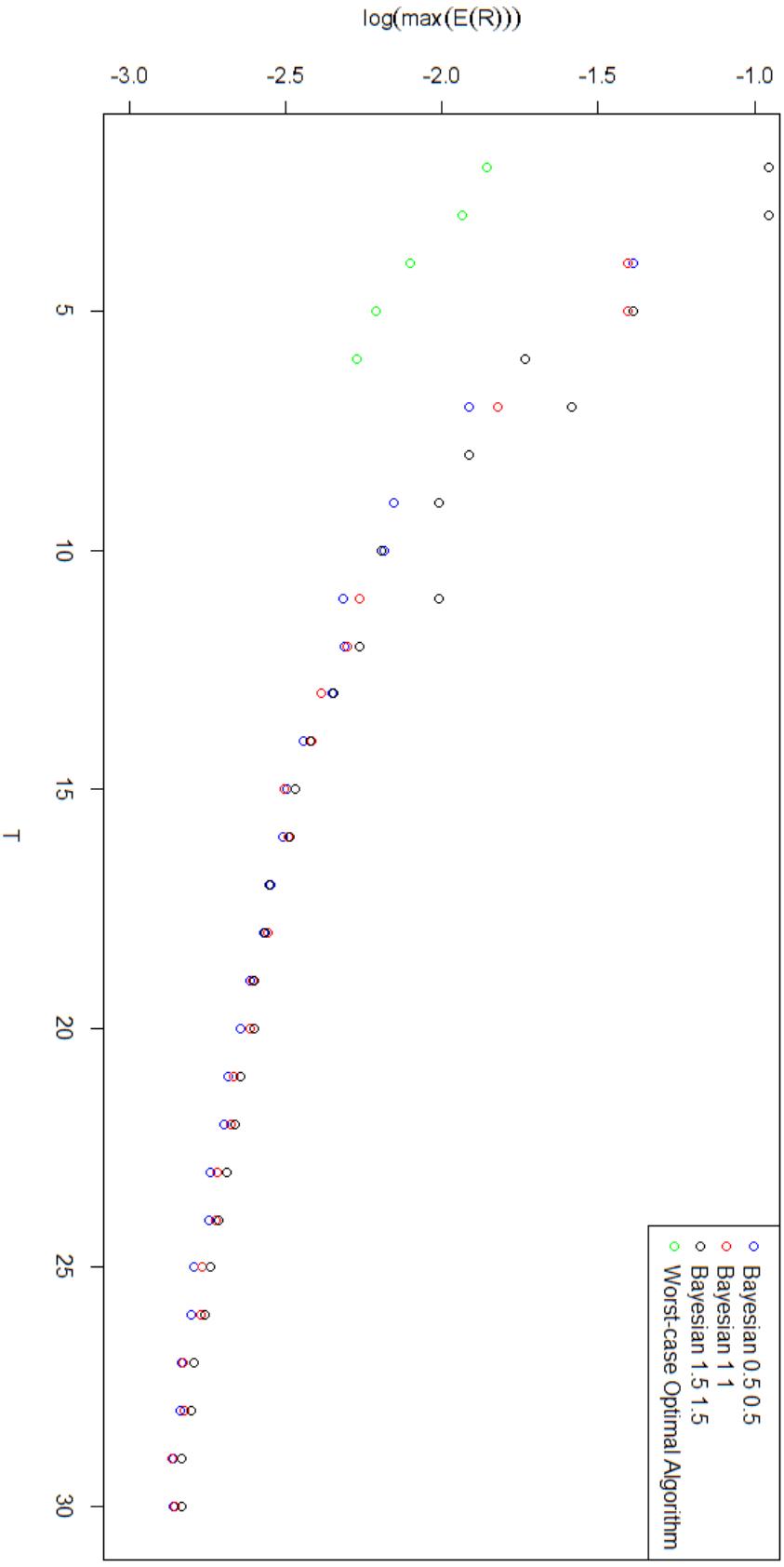


Figure 4.4: Results from the Bayesian Expected Regret Algorithm, compared to the results from the Optimal algorithm (Section 3). From $T = 10$ and onwards the regret seems to decrease exponentially as T increases.

5

Algorithm comparison

In this Section we make a more elaborate comparison of some algorithms used in Mini-max Arm Identification, using the worst-case Expected regret $\max_{\{p_{i,j}\}} \mathbb{E}(R)$ as our evaluation criterion. We will first introduce the subsequent algorithms in the format provided in Algorithm 1. Then we will compare their performance and their worst-case parameters.

5.1 Algorithms

In this subsection we will introduce four new algorithms for comparison.

5.1.1 Equals algorithm

The Equals algorithm was already presented in Section 2.2 (Algorithm 2). This algorithm serves as a logical upper-bound on the performance of an algorithm. The algorithm makes no effort to adjust its sampling based on the results from the samples and does nothing adaptive. Rather, it divides the budget T in 4 parts, and samples each arm accordingly.

5.1.2 Hierarchical algorithm

The first logical step up from the Equals algorithm is the Hierarchical algorithm. The idea is to first expend some of the budget to sample equally, determine which arm is $\min_j \hat{p}_{i,j}$ for each i, and then spend some more samples on those arms specifically. This effectively increases the amount of samples used on the (estimated) lowest arms, increasing the certainty of the estimations for $\hat{p}_{i,j}$. The budget T is divided in 6 parts: equal to the amount of arms $\{i, j\}$ plus the amount of arms $\{i\}$. First each arm gets one part, then the arms with the lowest $\min_j \hat{p}_{i,j}$ then get another part of the budget. These arms will have $^1/_3$ of the budget, instead of the $^1/_4$ of the Equals algorithm. The fraction of the budget spent equally versus on the minimal arms can be tuned further, maybe adaptively, but that is beyond the scope of this work.

Divide budget T in 6 parts R
↓
Sample each arm for one part R
↓
Estimate $\text{argmin}_j \hat{p}_{i,j}$ **for both** i
↓
Sample $\text{argmin}_j \hat{p}_{i,j}$ **for another part R for both** i
↓
Estimate $\hat{p}_{i,j}$ again taking the new samples into account
↓
Estimate $\text{argmax}_i \, min_j \hat{p}_{i,j}$

Figure 5.1: The actions taken by the Hierarchical algorithm.

```
someSampleRule ← function(t, x_{i_t,j_t}){
    if t < (4/6)T then
        /* Samples equally for first 4 parts of the budget.                    */
        i ← t mod 2 + 1
        j ← ⌈0.5t⌉ mod 2 + 1
    else if t = (2/3)T then
        /* Performs last equal sample and determines min_j p̂_{i,j}.           */
        i ← t mod 2 + 1
        j ← ⌈0.5t⌉ mod 2 + 1
        foreach {i, j} do
            p̂_{i,j} ← mean({x_{i_t,j_t} | i_t = i and j_t = j})
        end
        for I in i do
            j_{min,I} ← argmin_j p̂_{I,j}
        end
    else
        /* Samples equally over the estimated min_j p̂_{i,j}.                  */
        i ← t mod 2 + 1
        j ← j_{min,i}
    end
    return({i,j})
}
someRecommendationRule ← function(x_{i_t,j_t}){
    foreach {i, j} do
        p̂_{i,j} ← mean({x_{i_t,j_t} | i_t = i and j_t = j})
    end
    if min_j p̂_{1,j} = min_j p̂_{2,j} then
        i ← Bern(0.5) + 1
        /* Tie:  resolve uniformly at random.                                  */
    else
        i ← argmax_i min_j p̂_{i,j}
    end
    return(i)
}
```

Algorithm 3: The Hierarchical algorithm's functions. See Figure 5.1.

5.1.3 Lower Confidence Bound algorithm

The objective of the Min-max Action Identification algorithm is to find the $\max_i \min_j p_{i,j}$, which means it is sensible to focus as much as possible on sampling the lowest arms on each side. Once the estimator of an arm becomes relatively high, it is unlikely to be the $\min_j p_{i,j}$, so there no point sampling from that arm anymore. The Lower Confidence Bound algorithm implements this by using the 95% confidence interval on the estimated $\hat{p}_{i,j}$. When some $p_{i,j}$ is high, its confidence interval should be high as well, scaled to the amount of samples already spent on that arm. This algorithm is adaptive by sampling the arm for each i that has the lowest lower bound on the 95% confidence interval of $\{p_{i,j}\}$.

Because this interval is calculated after each sample, we need a confidence interval that is accurate when n is low. There is a multitude of confidence intervals for the Binomial distribution, each with different drawbacks, but they all struggle when $\hat{p}$ is extreme and/or n is close to 0. We use the Jeffreys Interval, based on the Quantiles of the Beta function [1]:

$$CI_{100(1-\alpha)\%}(x,n) = \begin{cases} (0, B_{1-\alpha/2}(x+0.5, n-x+0.5)) & \text{if } x = 0 \\ (B_{\alpha/2}(x+0.5, n-x+0.5)), 1) & \text{if } x = n \\ (B_{\alpha/2}(x+0.5, n-x+0.5)), B_{1-\alpha/2}(x+0.5, n-x+0.5)) & \text{otherwise} \end{cases} \tag{5.1}$$

Where $B_q(a,b)$ is the q-th quantile of the Beta distribution with parameters a and b.

5.1.4 One-more Bayesian algorithm

The One-more Bayesian algorithm is inspired by the Bayesian algorithm presented in Section 4, but instead of minimizing the Expected regret over the entire budget, the algorithm samples as if there is only one sample left. This could be seen as a greedy approach, sampling each time from the arm that minimizes the expected regret right there and then. In other words, I have one more sample to spend, how do I do that as efficiently as possible? That question is asked for each and every sample in the budget T. The sample rule would then be an adaptation of Equation 4.6. The recommendation rule is the same as for the Bayesian algorithm (Equation 4.5).

`someSampleRule` $\leftarrow$ function(t, x_{i_t,j_t}){
  foreach $\{i, j\}$ **do**
    $\hat{p}_{i,j} \leftarrow \text{mean}(\{x_{i_t,j_t} | i_t = i \text{ and } j_t = j\})$
  end
  $n_{i,j} \leftarrow \frac{\sum_t x_{i,j_t}}{\hat{p}_{i,j}}$
  $i \leftarrow t \mod 2 + 1$
  for J *in* $\{1, 2\}$ **do**
    $CI_J \leftarrow CI_{95\%}(\hat{p}_{i,J} \cdot n_{i,J}, n_{i,J})$
  end
  $j \leftarrow \text{argmin}_J\, CI_J$
  return({i,j})
}
`someRecommendationRule` $\leftarrow$ function(x_{i_t,j_t}){
  foreach $\{i, j\}$ **do**
    $\hat{p}_{i,j} \leftarrow \text{mean}(\{x_{i_t,j_t} | i_t = i \text{ and } j_t = j\})$
  end
  if $\min_j \hat{p}_{1,j} = \min_j \hat{p}_{2,j}$ **then**
    $i \leftarrow \text{Bern}(0.5) + 1$
    `/* Tie: resolve uniformly at random. */`
  else
    $i \leftarrow \text{argmax}_i \min_j \hat{p}_{i,j}$
  end
  return(i)
}

Algorithm 4: The Lower Confidence Bound algorithm's functions. For $CI_{95\%}(a, b)$ see Equation 5.1.

`someSampleRule` $\leftarrow$ function(x_{i_t,j_t}){
  $a^{i,j}_{pos} \leftarrow 0.5 + \sum_t x_{i,j_t}$
  $b^{i,j}_{pos} \leftarrow 0.5 + n_{i,j_t} - \sum_t x_{i,j_t}$
  $\{i, j\} \leftarrow \text{argmin}_{i,j} \left[\frac{a^{i,j}_{pos}}{a^{i,j}_{pos}+b^{i,j}_{pos}} \mathbb{E}_{\{p_{i,j}\}}(R^1_{i,j}) + \frac{b^{i,j}_{pos}}{a^{i,j}_{pos}+b^{i,j}_{pos}} \mathbb{E}_{\{p_{i,j}\}}(R^0_{i,j}) \right]$
  return({i,j})
}
`someRecommendationRule` $\leftarrow$ function(x_{i_t,j_t}){
  $i \leftarrow \text{argmax}_I \iint (\min_j p_{I,j}) P(\{p_{i,j}\} | v(x_{i,j_t}))\, \mathrm{d}p_{I,1}\, \mathrm{d}p_{I,2}$
  return(i)
}

Algorithm 5: The One-more Bayesian algorithm's functions.

5.2 Performance Evaluation

For the evaluation we again use the $\max_{\{p_{i,j}\}} \mathbb{E}(R)$, based on a grid for fixed $\{p_{i,j}\}$. The evaluation of the Equals algorithm (Section 5.1.1) and the Hierarchical algorithm (Section 5.1.2) can be found in closed form by considering all possible results from the samples and the regret incurred by the algorithm in those occasions (similar to Equation 2.5). This is doable because the Equals algorithm does not sample adaptively and the Hierarchical algorithm does so in a very predictable manner. This means that the amount of samples drawn from each arm is stable, which makes the amount of possible outcomes a relatively small set.

Shown below is the closed form for the evaluation of the Equals algorithm, assuming T is divisible by 4:

$$\mathbb{E}(R; \{p_{i,j}\}, T) = \sum_{i=0}^{\frac{T}{4}} \sum_{j=0}^{\frac{T}{4}} \sum_{k=0}^{\frac{T}{4}} \sum_{l=0}^{\frac{T}{4}} \left[R(\text{argmax}\{\min\{i,j\}, \min\{k,l\}\}; \{p_{i,j}\}) \prod_{x \in \{i,j,k,l\}} B(x, \frac{T}{4}, \{p_{i,j}\}) \right] \tag{5.2}$$

Where $B(x, n, p)$ is the Binomial probability of x successes out of n trials with probability p. $R(I; \{p_{i,j}\})$ is the regret incurred with recommendation I given parameter set $\{p_{i,j}\}$ (Equation 2.1). The Hierarchical evaluation is computed in a similar fashion, but is not shown here because it is more complicated.

The performance with the worst-case set $\{p_{i,j}\}$ would then be:

$$\max_{\{p_{i,j}\}} \mathbb{E}(R; \{p_{i,j}\}, T) \tag{5.3}$$

This closed form compilation is not the case for the Lower Confidence Bound algorithm (Section 5.1.3) and the One-more Bayesian algorithm (Section 5.1.4), as these sample adaptively. To calculate the $\mathbb{E}(R)$ the entire strategy would have to be calculated in a manner similar to the Bayesian algorithm in Section 4. This then again runs into similar problems as that algorithm: the tree becomes exponentially large as T increases. So instead we evaluate these algorithms based on the regret averaged over 1,000 runs of the algorithm for each set $\{p_{i,j}\}$.

5.3 Results

Based on the evaluation described in Section 5.2, the performances of each presented algorithm is performed on a grid of parameters $\{p_{i,j}\}$. The results of this are shown in Figure 5.2 and Table 5.1 along with the performances of the Bayesian algorithm and the Worst-case Optimal algorithm. Because the Worst-case Optimal algorithm could only be calculated for $T \leq 6$, it is omitted in Figure 5.2. As $T = 30$ is approximately the upper limit for the Bayesian algorithm, the performances in Table 5.1 are shown for $T = \{6, 30, 60, 100\}$.

Because of the guarantee the Worst-case Optimal algorithm provides, it is the best at $T = 6$, closely followed by the Hierarchical algorithm. With the absence of results for the Worst-case Optimal algorithm at higher budgets, the Hierarchical algorithm shows the best performance. At $T = 60$ the performance of the Lower Confidence Bound algorithm is slightly better than the Hierarchical algorithm. An important thing to note though is that the performance of the Lower Confidence Bound algorithm is based on a Monte Carlo sample of 1,000 runs of the algorithm. This means that there is some margin of error in the performance estimation. This margin of error is very unpredictable however, because the measure is the maximum of the Expected regrets evaluated over a grid of $\{p_{i,j}\}$. This also means that the measure is biased to high extrema. However, because the worst-case set $\{p_{i,j}\}$ for the Lower Confidence Bound algorithm looks very much alike the others, we do not believe that these measures are outliers.

It seems that the attempts to sample adaptively do not improve much on sampling (hierarchically) equally. In contrary, the One-more Bayesian algorithm performs much worse. The sampling methods of this algorithm are very greedy though, looking only at what is best in the short run, which means that either it works correctly, or it fails miserably. This is because there is no guarantee that it will sample all arms. For example, if sampling the first arm minimizes the expected regret, there is no reason for the algorithm to try the other arms. This way it is possible for the algorithm to completely tunnel-vision on one arm.

The worst-case parameter sets $\{p_{i,j}\}$ are shown in Table 5.2. The worst-case parameter distribution on the Worst-case Optimal algorithm are ommitted, but can be found in Table 3.4. As mentioned in earlier sections, an interesting pattern appears in the worst-case sets. It seems that the min-max arm $\max_i \min_j p_{i,j}$ is equal to the non-minimal arm on the same side. This is further discussed in Section 6.

Algorithm		T = 6	30	60	100
Worst-case Optimal		0.1095045	NA	NA	NA
Bayesian	0.5 0.5	0.17676308	0.05725861	NA	NA
	1 1	0.1767631	0.0574952	NA	NA
	1.5 1.5	0.1767631	0.0590056	NA	NA
Equals		0.1924198	0.06740518	0.04774235	0.03702653
Hierarchical		0.1133918	0.05161268	0.0382073	0.02987815
Lower Confidence Bound		0.2134286	0.0567	0.0381	0.03195239
One-more Bayesian		0.2255	0.133	NA	NA

Table 5.1: The Maximum Expected regret values for various algorithms. The Worst-case Optimal algorithm is elaborated upon in Section 3. The Bayesian algorithms and their priors are discussed in Section 4, the other shown algorithms are discussed in Section 5.1 and 5.2. The green cells show the best performances for that budget (among the available data).

Algorithm		T	$p_{1,1}$	$p_{1,2}$	$p_{2,1}$	$p_{2,2}$	δ
Bayesian	0.5 0.5	30	0.5	1	0.65	0.65	0.15
	1 1	30	0.4285714	1	0.5714286	0.5714286	0.1428572
	1.5 1.5	30	0.5	1	0.65	0.65	0.15
Equals		30	0.5	1	0.7142857	0.7142857	0.2142857
		60	0.4285714	1	0.5714286	0.5714286	0.1428572
		100	0.4452381	1	0.5547619	0.5547619	0.1095238
Hierarchical		30	0.35	0.7142857	0.5714286	0.5714286	0.2214286
		60	0.4285714	0.7142857	0.5714286	0.5714286	0.1428572
		100	0.3833333	0.5880953	0.4785714	0.4785714	0.0952381
Lower Confidence Bound		30	0.5	0.7142857	0.65	0.65	0.15
		60	0.35	0.5714286	0.5	0.5	0.15
		100	0.4452381	0.6333333	0.55	0.55	0.1047619
One-more Bayesian		30	0.6896552	1	0.9310345	0.9310345	0.2413793

Table 5.2: Worst-case parameter set $\{p_{i,j}\}$ for the different algorithms discussed for $T = 30$, $T = 60$ and $T = 100$ where available. The possible regret incurred $\delta = |\min_j p_{1,j} - \min_j p_{2,j}|$ is also displayed.

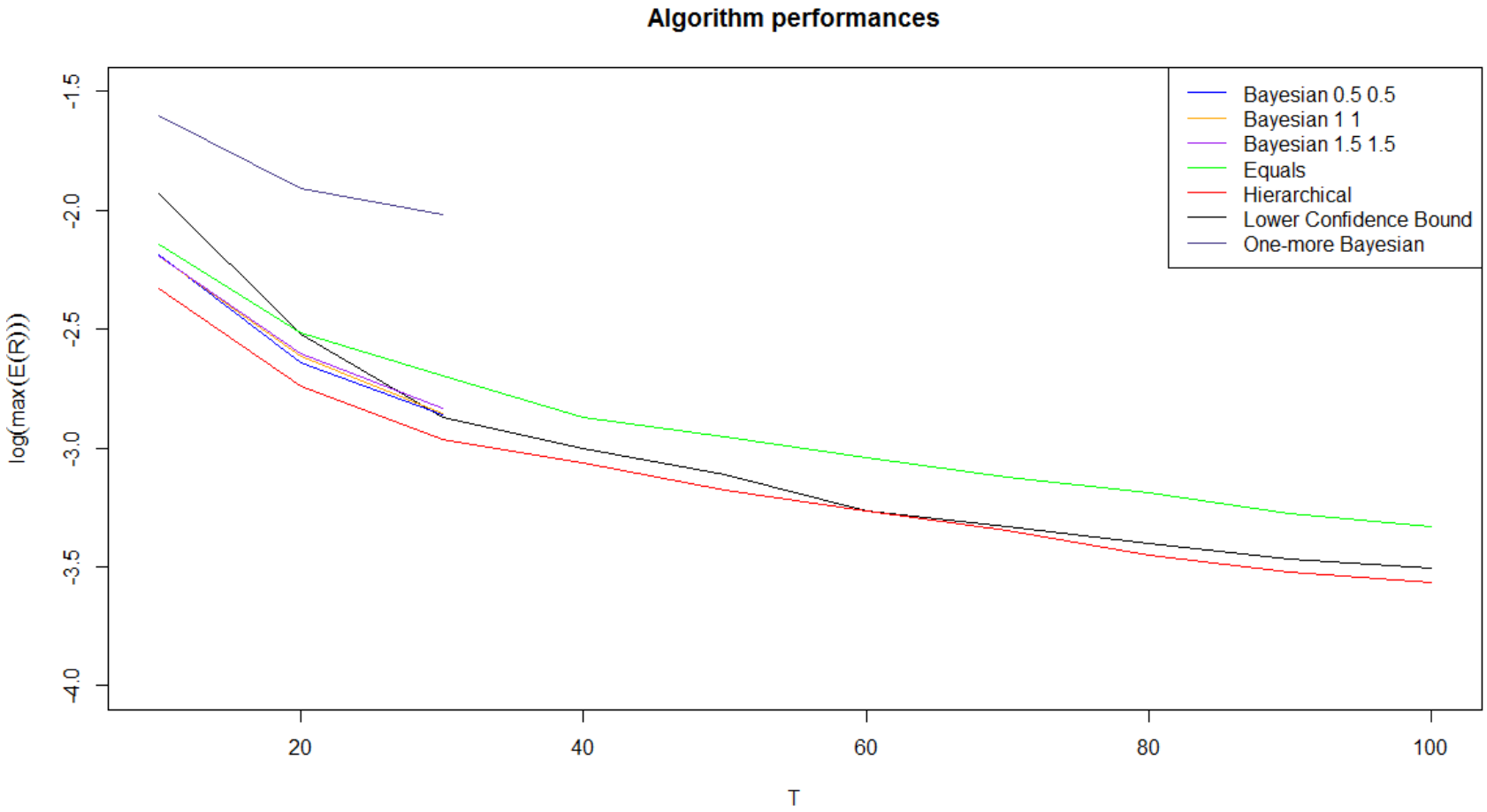


Figure 5.2: Maximum Expected regret values plotted on a logarithmic scale. Shown are the results of several algorithms with budget T in intervals of 10. The Bayesian algorithms and their priors are discussed in Section 4, the other shown algorithms are discussed in Section 5.1 and 5.2.

6

Worst-case set $\{p_{i,j}\}$

Based on the information found on the worst-case sets of $\{p_{i,j}\}$ in Sections 2.3, 3.9, 4.4 and 5.3, we noticed a pattern in the worst-case sets $\{p_{i,j}\}$. In this section we specify the pattern and prove that it is indeed the worst-case set in algorithms with different recommendation rules.

6.1 The pattern

As already described in Section 3.9, the minima on both sides ($\min_j p_{i,j}$) are centered around 0.5. The pattern we found describes the value of the arms that are not the minima. If $p_{1,1} = \min_j p_{1,j}$ and $p_{2,1} = \max_i \min_j p_{i,j}$, then the pattern we find is $p_{1,1} < p_{2,1} = p_{2,2} \ll p_{1,2}$ (Figure 6.1).

For Best-arm identification we already saw in Figure 2.1 that the worst-case set $\{p_{i,j}\}$ is the equilibrium of a big regret on one side, and a high probability for the algorithm to make a mistake (Section 1.3). In the Min-max setting, the worst case set $\{p_{i,j}\}$ is very similar to Best-Arm, if we consider the Equals algorithm (Algorithm 2). The $\min_j p_{i,j}$ follow a similar pattern as the Best-arm worst case set. The interesting thing is the behavior of the regret as the non-minimal parameters change. The intuition behind this is easy to understand if you consider the following: The algorithm makes a mistake if the estimate for $\operatorname{argmax}_i \min_j \hat{p}_{i,j}$ is not actually $\operatorname{argmax}_i \min_j p_{i,j}$. This could happen because of the randomness of the samples drawn, and becomes more likely, again, with a lower sample size T or a smaller difference $\delta = |\min_j p_{1,j} - \min_j p_{2,j}|$. But also the role of the other parameters can be explained with this. Because the algorithm makes its decisions on the Maximum Likelihood Estimates of the parameters, the chance of $\min_j \widehat{p_{i^*,j}}$ to be lower than $\min_j \widehat{p_{i^-,j}}$ becomes bigger when then $p_{i^*,j}$ are all close together (See $p_{2,j}$ in Figure 6.1). The opposite is the case for the arms that are not Min-max (i^-); here the $\min_j p_{i^-,j}$ and the other arms from i^* are so far apart that their confidence intervals do not overlap anymore.

In the upcoming subsections we will provide proof for this pattern with respect to both $p_{2,2}$ and $p_{1,2}$. Seeing as we found this pattern in multiple types of recommendation rules, Bayesian and frequentist, we will show for both settings that this proof applies.

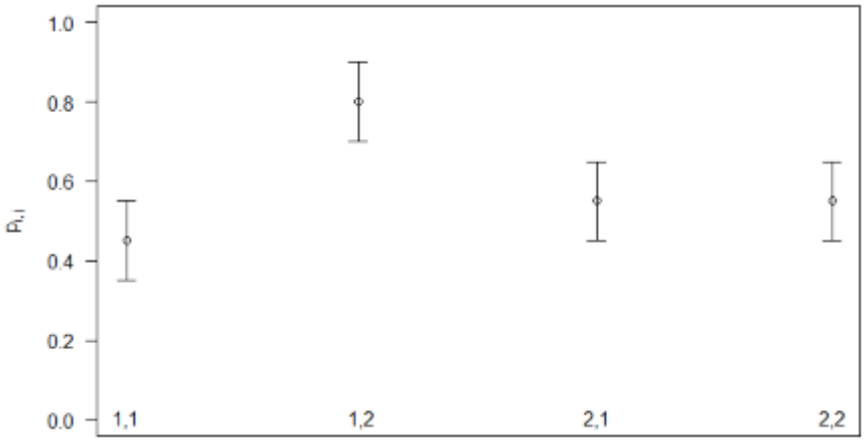


Figure 6.1: Example of a set of $\{p_{i,j}\}$ with arbitrary confidence intervals.

6.2 $p_{2,2}$

We will provide proof for the pattern described in Section 6.1. First we start with $p_{2,2}$:

Theorem 2. *Let* $p_{1,1}$ *be* $\min_j p_{1,j}$ *and* $p_{2,1}$ *be* $\max_i \min_j p_{i,j}$. *Let* $p_{1,1}$, $p_{1,2}$ *and* $p_{2,1}$ *be fixed. Consider an algorithm that samples all arms equally and makes its recommendation on any function that strictly increases with the sufficient statistic. Then* $\operatorname{argmax}_{p_{2,2}} \mathbb{E}(R) = p_{2,1}$.

Proof. Let $p_{1,1}$ be $\min_j p_{1,j}$ and $p_{2,2}$ be $\max_i \min_j p_{i,j}$. Let $p_{1,1}$, $p_{1,2}$ and $p_{2,1}$ be fixed. Because $\min_j p_{i,j}$ are fixed, the regret incurred if the algorithm makes a mistake is fixed as well. So to maximize $\mathbb{E}(R)$ we need to find:

$$\operatorname*{argmax}_{p_{2,2}} \mathbb{P}(\hat{I} \neq I)$$

Let $S_{i,j}$ be the sufficient statistic of n samples drawn from $p_{i,j}$. Let $f(S_{i,j})$ be some function used in the recommendation of the algorithm, strictly increasing in $S_{i,j}$.

Because we change $p_{2,2}$, conditional to $p_{1,1}$, $p_{1,2}$ and $p_{2,1}$ the objective becomes:

$$\operatorname*{argmax}_{p_{2,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}} \sim p_{\{1,1\}} \\ x_{\{1,2\}} \sim p_{\{1,2\}} \\ x_{\{2,1\}} \sim p_{\{2,1\}}}} [\mathbb{P}(S_{1,j} > S_{2,j})]$$

Which, as we take the expectation over $\bar{x}_{\{1,1\},\{1,2\},\{2,1\}}$, equals:

$$\operatorname*{argmax}_{p_{2,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}} \sim p_{\{1,1\}} \\ x_{\{1,2\}} \sim p_{\{1,2\}} \\ x_{\{2,1\}} \sim p_{\{2,1\}}}} [\mathbb{P}(c_{1,j} > c_{2,1} f(S_{2,2}))]$$

Because $f(S)$ is strictly increasing:

$$\operatorname*{argmax}_{p_{2,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{1,2\}}\sim p_{\{1,2\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}}}} \left[\mathbb{P}(f^{-1}(c) > S_{2,2})\right]$$

$$= \operatorname*{argmax}_{p_{2,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{1,2\}}\sim p_{\{1,2\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}}}} \left[cdf_{S_{2,2}}(f^{-1}(c))\right]$$

As S is binomially distributed: $S_{i,j} \sim B(p_{i,j}, n)$: [14]

$$cdf_{S_{2,2}}(x) = I_{1-p_{2,2}}(n-x, x+1)$$

Where $I_x(a,b)$ is the regularized incomplete beta function. This function is strictly increasing with x, therefore $cdf_{S_{i,j}}$ is strictly decreasing in $p_{i,j}$, which implies (because $p_{2,2} \geq p_{2,1}$):

$$\operatorname*{argmax}_{p_{2,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{1,2\}}\sim p_{\{1,2\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}}}} \left[cdf_{S_{2,2}}(f^{-1}(c))\right] = p_{2,1}$$

This implies that:

$$\operatorname*{argmax}_{p_{2,2}} \mathbb{P}(\hat{I} \neq I) = p_{2,1}$$

□

6.3 $p_{1,2}$

The proof for $p_{1,2}$ is very similar to that of $p_{2,2}$ provided in Theorem 2, but then reversing the logic:

Theorem 3. *Let $p_{1,1}$ be $\min_j p_{1,j}$ and $p_{2,2}$ be $\max_i \min_j p_{i,j}$. Let $p_{1,1}$, $p_{2,1}$ and $p_{2,2}$ be fixed. Consider an algorithm that samples all arms equally and makes its recommendation on any function that strictly increases with the sufficient statistic. Then $\operatorname{argmax}_{p_{1,2}} \mathbb{E}(R) = 1$.*

Proof. Let $p_{1,1}$ be $\min_j p_{1,j}$ and $p_{2,2}$ be $\max_i \min_j p_{i,j}$. Let $p_{1,1}$, $p_{2,1}$ and $p_{2,2}$ be fixed. Because $\min_j p_{i,j}$ are fixed, the regret incurred if the algorithm makes a mistake is fixed as well. So to maximize $\mathbb{E}(R)$ we need to find:

$$\operatorname*{argmax}_{p_{1,2}} \mathbb{P}(\hat{I} \neq I)$$

Let $S_{i,j}$ be the sufficient statistic of n samples drawn from $p_{i,j}$. Let $f(S_{i,j})$ be some function used in the recommendation of the algorithm, strictly increasing in $S_{i,j}$.

Because we change $p_{1,2}$, conditional to $p_{1,1}$, $p_{2,1}$ and $p_{2,2}$ the objective becomes:

$$\operatorname*{argmax}_{p_{1,2}} \mathop{\mathbb{E}}_{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}} \\ x_{\{2,2\}}\sim p_{\{2,2\}}}} [\mathbb{P}(S_{1,j} > S_{2,j})]$$

Which, as we take the expectation over $\bar{x}_{\{1,1\},\{2,1\},\{2,2\}}$, equals:

$$\underset{p_{1,2}}{\operatorname{argmax}} \underset{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}} \\ x_{\{2,2\}}\sim p_{\{2,2\}}}}{\mathbb{E}} [\mathbb{P}(c_{1,1} f(S_{1,2}) > c_{2,j})]$$

Because $f(S)$ is strictly increasing:

$$\underset{p_{1,2}}{\operatorname{argmax}} \underset{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}} \\ x_{\{2,2\}}\sim p_{\{2,2\}}}}{\mathbb{E}} \left[\mathbb{P}(f^{-1}(c) < S_{2,1})\right]$$

$$= \underset{p_{1,2}}{\operatorname{argmax}} \underset{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}} \\ x_{\{2,2\}}\sim p_{\{2,2\}}}}{\mathbb{E}} \left[1 - cdf_{S_{1,2}}(f^{-1}(c))\right]$$

As S is binomially distributed: $S_{i,j} \sim B(p_{i,j}, n)$: [14]

$$cdf_{S_{1,2}}(x) = I_{1-p_{1,2}}(n - x, x + 1)$$

Where $I_x(a, b)$ is the regularized incomplete beta function. This function is strictly increasing with x, therefore $cdf_{S_{i,j}}$ is strictly decreasing in $p_{i,j}$, and again $1 - cdf_{S_{1,2}}$ is strictly increasing:

$$\underset{p_{1,2}}{\operatorname{argmax}} \underset{\substack{x_{\{1,1\}}\sim p_{\{1,1\}} \\ x_{\{2,1\}}\sim p_{\{2,1\}} \\ x_{\{2,2\}}\sim p_{\{2,2\}}}}{\mathbb{E}} \left[1 - cdf_{S_{1,2}}(f^{-1}(c))\right] = 1$$

This implies that:

$$\underset{p_{1,2}}{\operatorname{argmax}}\, \mathbb{P}(\hat{I} \neq I) = 1$$

□

6.4 $f(S)$

The proof in Sections 6.2 and 6.3 has one requirement that has to be cleared up. This is the monotonicity of $f(S_{i,j})$ in $p_{i,j}$. Now we will show that this holds in both frequentist and Bayesian recommendation rules.

Theorem 4. *The recommendation rule* $\operatorname{argmax}_i \min_j \hat{p}_{i,j}$ *has an associated* $f(S_{i,j})$ *which is monotone in* $p_{i,j}$ *if* n *is equal for all arms.*

Proof. The associated $f(S_{i,j})$ for the recommendation rule $\operatorname{argmax}_i \min_j \hat{p}_{i,j}$ is:

$$f(S_{i,j}) = \min_j \hat{p}_{i,j}$$

As n is equal for all arms, $\hat{p}_{i,j} \propto S_{i,j}$.

Now we need to find a distribution for $\min S_{i,j}$. If we define the CDF of $\min S_{i,j}$, so in other words $P(\min S_{i,j} < X)$ as $F(X)$, and $cdf(S_{i,j})$ as the CDF of $S_{i,j}$:

$$F(X) = 1 - \prod_j [1 - cdf(X)]$$

The CDF of the binomial distribution is

$$cdf_{S_{i,j}}(x) = I_{1-p_{i,j}}(n - x, x + 1)$$

$$F(X) = 1 - \prod_j \Big[1 - I_{1-p_{i,j}}(n - X, X + 1)\Big] = 1 - \prod_i \Big[I_{p_{i,j}}(X + 1, n - X)\Big]$$

Again, $I_x(a, b)$ is strictly increasing in x, so $F(X)$ is decreasing in $p_{i,j}$, which implies that $f(S_{i,j})$ increases in $p_{i,j}$.

□

Theorem 5. *The recommendation rule* $\operatorname{argmin}_i \mathbb{E}_{\{p_{i,j}\}}(R)$ *has an associated* $f(S_{i,j})$ *which is monotone in* $p_{i,j}$ *if* n *is equal for all arms.*

Proof. The associated $f(S_{i,j})$ for the recommendation rule $\operatorname{argmin}_i \mathbb{E}_{\{p_{i,j}\}}(R)$ is:

$$\iint (\min_j p_{i,j}) P(\{p_{i,j}\}|v(x)) \, \mathrm{d}p_{i,1} \, \mathrm{d}p_{i,2} = \mathbb{E} \min_j p^{i,j}_{posterior}$$

$p^{i,j}_{posterior} \sim \beta(S_{i,j} + a^{i,j}_{prior}, n - S_{i,j} + b^{i,j}_{prior})$, which we can use to define the cdf of $\min_j p_{i,j}$:

$$F(X) = 1 - \prod_j \Big[1 - cdf_{p^{i,j}_{posterior}}(X)\Big]$$

$$cdf_{p^{i,j}_{posterior}}(X) = I_X(S_{i,j} + a^{i,j}_{prior}, n - S_{i,j} + b^{i,j}_{prior})$$

As $p_{i,j}$ increases, so does $S_{i,j}$. $I_x(a, b)$ is both strictly decreasing in a and strictly increasing in b, so strictly decreasing in $S_{i,j}$. Which means that $F(X)$, the cdf of $\min_j p_{i,j}$ is strictly decreasing in $p_{i,j}$.

$$\mathbb{E}(X) = \int_0^\infty (1 - F_X(x)) \mathrm{d}x \quad \text{for } X \geq 0$$

This imlies that if $F_x(X) > F_y(X) \quad \forall\, X \implies \mathbb{E}_x(X) < \mathbb{E}_y(X)$. This means that $\mathbb{E} \min_j p^{i,j}_{posterior}$ is strictly increasing in $p_{i,j}$.

□

6.5 Discussion

The subsections above proved that in algorithms that sample all arms equally and have some specific rules on which they base the recommendations, the worst-case parameter set $\{p_{i,j}\}$ has the following property: If $p_{1,1} = \min_j p_{1,j}$ and $p_{2,2} = \max_i \min_j p_{i,j}$, then the pattern we find is $p_{1,1} < p_{2,1} = p_{2,2} \ll p_{1,2}$. However, the numerical results from the previous sections do suggest that these patterns hold even when some of these restrictions do not hold, especially regarding $p_{2,2}$. The behavior of $p_{1,2}$, being as high as possible ($= 1$), is likely to change as the algorithm becomes more adaptive in its sampling. After a couple of samples, an algorithm might be able to identify $p_{1,2}$ to be (relatively) high, and stop sampling there. This would mean that more samples are being invested in actually relevant arms, allowing for a better estimate of those parameters, improving the accuracy. So one would expect there to be some trade-off, with at one side $p_{1,2}$ being as high as possible to maximize $\mathbb{P}(\hat{I} \neq I)$ and at the other side as low as possible to keep the algorithm interested. This effect can be seen in the results of the Hierarchical algorithm and the Lower Confidence Bound algorithm (Table 5.2).

Further proof for these patterns in algorithms that do not necessarily sample equally is yet to be found, but might be worth investigating.

7

Conclusion

7.1 Recap

We presented a new concept, combining the goal of Mini-max with the sampling methods of Best-Arm Identification. To explore the possibilities of this combination, we went on to create some sampling algorithms as per the setting created in Section 2. We presented two elaborate algorithms:

- The Worst-case Optimal algorithm (Section 3), which guarantees the best performance in terms of worst-case regret: $\max_{\{p_{i,j}\}} \mathbb{E}(R)$. The section elaborates on the steps taken to make the problem more compact to prevent it from exploding exponentially in T. Nonetheless, the problem was too big to solve for budgets higher than 6, rendering it practically useless.
- The Bayesian Optimal algorithm (Section 4) was presented as a simplification of the Worst-case Optimal objective by relaxing the goal of finding an algorithm that provided worst-case guarantees and instead provide an algorithm that does optimally conditional to a fixed (prior) distribution on $\{p_{i,j}\}$. This proved to be a more compact problem, but also struggled with higher budgets, because of the requirement of computing the entire strategy in advance. Computer-time and memory space quickly become a bottleneck. The performance of the Bayesian Optimal algorithm was worse than the Worst-case Optimal algorithm, as is to be expected.

As both elaborate algorithms have issues with computer time and memory space, caused by having to calculate the entire sampling strategy in advance, we created some practical algorithms with more simple strategies. These were compared in Section 5.

- Equals algorithm (Section 5.1.1), which samples each arm equally, regardless of what the outcome of the previous samples were. Then bases its recommendation on the Maximum Likelihood Estimator (MLE) for $\{\hat{p_{i,j}}\}$. This provided a baseline for other algorithms. As all algorithms that sample adaptively and (attempt to) allocate their budget more efficiently, should be able to get a higher performance than this. Else the adaptivity of the algorithm has failed.

- The Hierarchical algorithm (Section 5.1.2), which treats the mini-max problem as three separate problems: two minimizing problems for each side and then one maximizing problem between the two minima. In practice this ended up having the best performance among the algorithms tested.
- The Lower Confidence Bound algorithm (Section 5.1.3), which samples based on the lower bound of the Confidence Interval of the estimates of $\{p_{i,j}\}$. It performed decently, but was not optimal.
- The One-more Bayesian algorithm (Section 5.1.4), an adaptation of the Bayesian Optimal algorithm, spending its samples on the arm that yields the best gain in Expected regret at that point. This algorithm sampled very greedily, which is reflected in its performance: worse than the Equals algorithm.

One of the results gained from the performance evaluations was an idea of 'hard' parameters $\{p_{i,j}\}$. Because what is hard for algorithms to sample efficiently and draw correct conclusions from? All of the worst-case sets $\{p_{i,j}\}$ pointed to a single pattern: If $p_{1,1} = \min_j p_{1,j}$ and $p_{2,2} = \max_i \min_j p_{i,j}$, then the pattern we find is $p_{1,1} < p_{2,1} = p_{2,2} \ll p_{1,2}$. In Section 6 we proved that this is indeed the worst-case set $\{p_{i,j}\}$ in algorithms that sample each arm equally. The results however indicate that this is also the case if the samples are not distributed equally among the arms.

7.2 Recommendations

This thesis has followed a very practical approach to studying Mini-max Action Identification. For future work it might be beneficial to look at it in a more theoretical manner, like exploring the complexity of the problem. Furthermore, fine-tuning the presented algorithms to improve their performances is also an option.

Finally, I would suggest to see if a proof for a broader version of Theorem 2 can be found so that the assumption that the algorithm samples all arms equally can be relaxed. The results do already hint that this is the case after all.

8

References

[1] Lawrence D. Brown, T. Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statist. Sci.*, 16(2):101–133, 05 2001.

[2] Murray Campbell, a. Joseph Hoane Jr., and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[3] R. A. Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 222(594-604):309–368, 1922.

[4] Aurélien Garivier, Emilie Kaufmann, and Wouter M. Koolen. Maximin action identification: A new bandit framework for games. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016, New York, USA, June 23-26, 2016*, pages 1028–1050, 2016.

[5] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, 2012.

[6] Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 186(1007):453–461, 1946.

[7] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *Journal of Machine Learning Research*, 17(1):1–42, 2016.

[8] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 750–759, New York, NY, USA, 1994. ACM.

[9] J. von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[10] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[11] Howard Raiffa and Robert Schlaifer. *Applied Statistical Decision Theory*. 1961.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, pages 1–37, 2015.

[13] Stefan Theussl and Kurt Hornik. *Rglpk: R/GNU Linear Programming Kit Interface*, 2015. R package version 0.6-1.

[14] G.P. Wadsworth and J.G. Bryan. *Introduction to Probability and Random Variables*. A Wiley publication in mathematical statistics. McGraw-Hill, 1960.

[15] Robert Wilson. Computing equilibria of two-person games from the extensive form. *Management Science*, 18(7):448–460, 1972.