Unbiased Rational Decision Making in Multiple-adversary Environments

Anat Hashavit
Unbiased Rational Decision Making in Multiple-adversary Environments

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Anat Hashavit

Submitted to the Senate of the Technion — Israel Institute of Technology
Tishrei 5772 Haifa October 2011
The research thesis was done under the supervision of Prof. (Shaul) (Markovitch) in the Computer Science Department.

I would like to express my gratitude to my advisor, Prof. Shaul Markovitch, for his guidance during my studies. I am thankful to Shaul for guiding me in my first steps in the world of research, for the many fruitful discussions, and for the high standards he always required. Shaul always found the time to meet and discuss this research, even under the many constraints of both our schedules.

I would also like to thank Ariel Raviv, Sonia Liberman, Omer Levi, Erez Karpas, Kira Radinsky, Haggai Toledano, Assag Glazer, and Gal Naimark, my colleagues from the Learning and Reasoning lab for many valuable discussions.

Finally, I would like to thank my husband David, for his love and support during my studies.

The generous financial support of the Technion (and someone else if you got a scholarship) is gratefully acknowledged.
Contents

Abstract 1
1 Introduction 2
2 Problem Analysis: Multiple-Adversary, Binary-Utility Games 5
3 The Max-Prob Algorithm for an Unbounded, Unbiased-Rational Agent 13
4 The Max-Prob Algorithm for a Resource Bounded Agent 20
5 Learning the Heuristic Distribution 25
6 Empirical Evaluation 30
   6.1 Experimental Methodology ................................. 30
   6.2 Multiple Competitor Comparison ............................ 33
   6.3 Pairwise Comparisons ...................................... 35
7 Empirical Study on the Learning Algorithm 37
8 Related Work 39
9 Discussion 44
A Game Rules for Chinese Checkers and Abalone 47
   A.1 Chinese Checkers ........................................... 47
   A.2 Abalone .................................................... 48
Bibliography 51
Abstract in Hebrew
List of Figures

2.1 Example of pre-final states in both a two-player game and a multiple player game. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent $a_i$. .................................................. 6

2.2 Example of a multiple-adversary binary-utility game tree. The values in the brackets below the leaf nodes are vectors of utility values and the value next to each mid-game tree node is agents 1’s expected utility value, for that node, under the assumption of unbiased rational opponents. ................. 8

2.3 Example of a game played according to the generalized mini-max strategy. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent with the same index. Nodes $a$ and $b$ are MIN nodes and will thus propagate a value of 0 to both nodes $a$ and $b$. ............. 10

2.4 Example of a game played according to the Paranoid strategy. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent with the same index and the single value below them is the Paranoid value for agent 1. Nodes $a$ and $b$ are MIN nodes, and will thus propagate a value of -1 to node $a$ and a value of -2 to node $b$. 11

2.5 Example of a game played according to the MaxN strategy. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent with the same index. At node $a$ agent 2 will choose to propagate the values of either node $c$ or $d$ and at node $b$, the values of node $e$, $f$ or $g$. 12
3.1 A description of the UMP algorithm. The algorithm computes $P_w(s', a)$ for all successors of the current state and selects the move leading $a$ to a state with maximal $P_w$. If several such moves exist, it selects one uniformly at random. . . . . 15

3.2 Max-Prob. Example of a complete tree. The values in the brackets are the agent’s winning probabilities. . . . . . . . . . . 16

4.1 Max-Prob. Example of a partial tree evaluation. The values in the round brackets under the leaves are vectors of intervals, representing the bounds of the uniform distribution for each agent, where the value at index $i$ corresponds to the agent with the same index (counting from 1). . . . . . . . . . . . . 22

4.2 A description of the BMP algorithm. The algorithm computes $\hat{P}_w(s', a)$ for all successors of the current state and selects the move leading $a$ to a state with maximal $\hat{P}_w$. If several such moves exist, it selects one uniformly at random. . . . . 24

5.1 Data-flow of the probability learning process, and how the learned model is incorporated to the BMP algorithm. . . . . 29

7.1 The performance of BMP as a function of learning resources. 38

A.1 A Chinese Checkers game board at the start of a six-players game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to. . . . . . . . . . . . . 48

A.2 An example of the center of a Chinese Checkers board in the middle of a game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to and the black circles are empty slots player 1’s pawn can move to. 49

A.3 The board of the game of Abalone at the start of the game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to. . . . . . . . . . . . . 50
A.4 An example of a part of the Abalone board in the middle of the game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to. The arrows show some of the directions in which the marbles can move.
Abstract

In binary-utility games, an agent can have only two possible utility values for final states, 1 (win) and 0 (lose). We define an unbiased rational agent as one that seeks to maximize its utility value, but is equally likely to choose between states with the same utility value. In particular, it prefers winning over losing but is indifferent as to which winning (or losing) state is chosen. This induces a probability distribution over the game tree, from which an agent can infer its probability to win. A single adversary binary game is one where there are only two possible outcomes, so that the winning probabilities remain binary values. In this case, the rational action for an agent is to play minimax. In this work we focus on the more complex, multiple-adversary environment, where an agent is met with at least two adversaries. We propose a new algorithmic framework where agents try to maximize their winning probabilities. We begin by theoretically analyzing why an unbiased rational agent with unbounded resources should take our approach and not that of the existing Paranoid or MaxN algorithms. We then expand our framework to a resource-bounded agent, where winning probabilities are estimated using both manual and machine learning techniques, and show empirical results supporting our claims.
Chapter 1

Introduction

An intelligent agent is an entity that tries to perceive its environment and acts upon it in order to achieve an assigned goal. Which action to take is typically decided by lookahead search in the space of possible world states. The agent uses its perception to determine the current state of the world and evaluates the outcome of alternative action sequences, eventually selecting a sequence it believes will end at a goal state. The actions of other agents operating in the same environment might, however, change the state of the world. Therefore, when performing lookahead search, the agent must consider the actions of other agents, especially in an adversarial environment, where other agents can benefit from preventing the agent from achieving its goal.

When performing lookahead search in an adversarial environment, the outcome of the other agents’ actions is somewhat uncertain. There are two main approaches for solving these uncertainties. The first assumes we know nothing about the opponents, who are thus also assumed to be taking actions which lead to the worst outcome from our perspective. The second assumes the opponents are rational and are therefore trying to maximize their own utility value. In zero-sum two-player games, both approaches lead to the well-known minimax algorithm (Shannon, 1950).

With more than two agents, however, the two approaches lead to different algorithms. The first leads to the generalized minimax algorithm, where we propagate the maximal utility value of the agent at its own decision nodes and its minimal value at the opponents’ decision nodes. In a game with more than two agents, this strategy is overly conservative and in fact assumes
irrational behavior of the other agents as there may be cases where they will not prefer to maximize their own utilities.

The second approach leads to the MaxN algorithm (Luckhart & Irani, 1986), which propagates vectors of utilities. Each agent is assumed to select an action maximizing its own utility value. However, under this assumption, some uncertainty remains regarding the outcome of the opponents’ actions, as there may be several actions that lead to states that are equivalent from the perspective of the acting agent but distinct from the perspective of the other agents. This phenomenon is intensified for games with only a few possible utilities (such as win and lose). To account for this remaining uncertainty, a rational agent must attempt to maximize its expected utility value. However, as we will show in the next chapter, the MaxN strategy does not facilitate an expected utility maximization mechanism.

In this paper we present a novel algorithm for decision making in a multiple-adversary environment. We start by defining a multiple-adversary environment where the utility of a final state for an agent can be either 1(win) or 0(lose). We then define an agent that seeks to maximize its utility value, but is equally likely to choose between states with the same utility value. We call such an agent an unbiased rational agent. We show that a multiple-adversary environment with unbiased rational agents induces a probability distribution over the possible outcomes of the game. We claim that an unbiased rational agent acting in this environment should strive to maximize its winning probability. We then present a new algorithm, Max-Prob, for unbiased rational, lookahead-based decision making in a multiple-adversary environment and show that existing algorithms may act suboptimally in such a setup. We first present the algorithm for an unbounded environment, where the agents have enough resources to explore the full game tree. We prove that in such a setup, our algorithm defines a mixed strategy that is also a Nash equilibrium. In addition, we present a variation of the algorithm for a resource-bounded setup, where we show how an agent’s winning probability can be computed using the distribution of a heuristic evaluation function’s values at the end of the game. We estimate this distribution using both human knowledge and an example-based machine learning method that infers the distribution of the final states’ heuristic values. The examples are retrieved from logs of past games. Finally, we empirically evaluate our algorithm in a variety of setups and show
its superiority over existing methods.
Chapter 2

Problem Analysis:
Multiple-Adversary, Binary-Utility Games

The process where multiple agents commit actions in the environment in order to achieve their assigned goal is often referred to as a game. We define a binary-utility game as one consisting of the following elements:

1. A set of agents $A = \{a_1, ..., a_n\}$.
2. A set of game states, $S$.
3. An initial state $s_i \in S$.
4. A set $F \subseteq S$ of final states.
5. A function, $\text{Turn} : S \rightarrow A$, which designates the acting agent in the given state. We assume that only one agent acts at each state.
6. A function, $\text{Succ} : S \rightarrow 2^S$, which returns all the states directly succeeding $s$, under the constraints of the game, s.t the graph induced by $\text{Succ}$ is acyclic.
7. For each agent $a \in A$, a binary utility function associated with that agent, $U_a : F \rightarrow \{0, 1\}$, which returns, for a final state $s_f \in F$, a binary score, denoting whether the agent has achieved its goal. Additionally,
Figure 2.1: Example of pre-final states in both a two-player game and a multiple player game. The values in the brackets are vectors of utility values, where the value at index \(i\) corresponds to the agent \(a_i\).

for convenience, we will denote by \(U(s_f)\) the vector containing all agents’ utility values: \((U_{a_1}(s_f), ..., U_{a_n}(s_f))\).

We further define an **adversarial binary-utility** game as one where for each final state \(f \in F\):

\[ |A| > \left| \{ a \in A | U(f,a) = 0 \} \right| > 0. \]

This condition, that at each final state there is at least one winner and one loser, is a minimal condition for conflict between the agents.

We call a state \(s \in S\) that satisfies \(\text{Succ}(s) \subseteq F\) a **pre-final** state. Let \(a \in A\) be an agent, and \(s \in S\) be a pre-final state. We define the set of winning states for agent \(a\) as \(W(s,a) = \{ s' \in \text{Succ}(s) | U(s',a) = 1 \}\). \(L(s,a)\) is similarly defined as the set of losing states. In addition, we define the set of utility vectors associated with \(W(s,a)\) as \(U_W(s,a) = \{ U(s') | s' \in W(s,a) \}\) and similarly the set of utility vectors associated with \(L(s,a)\) as \(U_L(s,a)\). Since a rational agent is defined as an agent that seeks to maximize its expected utility value (von Neumann & Morgenstern, 1947), in a pre-final state, such an agent selects a state from \(W(s,a)\) if possible and one from \(L(s,a)\) otherwise.

In a two-agent adversarial, binary-utility game, where the possible utility vectors are either \((1,0)\) or \((0,1)\), \(0 \leq |U_W(s,a)| \leq 1\) and \(0 \leq |U_L(s,a)| \leq 1\), for all agents and all pre-final states, removing all uncertainties regarding the outcome of a rational agent’s decision. However, this is no longer true
when the number of agents is higher than two.

An example of this difference can be seen in Figure 2.1. The numbers in the brackets under the leaves are the utility vectors of the agents. The left side of the figure shows a game tree for a two-player binary-utility game, where agent 1 is the one to act. It can be seen that although \( W(r, a_1) \) contains more than one state, \( |U_W(r, a_1)| = 1 \). Assuming agent 1 is rational, it will select a state from \( W(r, a_1) \). No matter which one of the states in \( W(r, a_1) \) agent 1 will choose, the game will end with agent 1 having a utility value of 1, and agent 2 a utility value of 0.

The right side of the figure shows a game tree of a three-player binary-utility game. Again, agent 1 is the one to act. It can be seen that \( U_L = \emptyset \) and therefore \( |U_L(r, a_1)| = 0 \), and \( U_W(r, a_1) = \{(1,0,1),(1,1,0)\} \) and therefore \( |U_W(r, a_1)| = 2 \). Agent 1 will select an action leading it to a state in \( W(r, a_1) \), and therefore finish the game with a utility value of 1. The utility values of agents 2 and 3, however, are uncertain. If agent 1 will choose node \( a \), agent 2 will finish the game with a utility value of 0 and agent 3 with a utility value of 1, but if agent 1 will choose node \( b \), agent 2 will finish the game with a utility value of 1 and agent 3 with a utility value of 0.

The reason for this uncertainty, despite the assumption of a rational agent, is that the original definition of rationality poses no restrictions as to the preferences of an agent choosing between states where its expected utility is identical. Given \( N \) states where the utility values of a rational agent are equal, and given no additional prior knowledge regarding the agent, predicting which state will be chosen by the agent is equivalent to the problem of assigning probabilities to \( N \) mutually exclusive events, again without any prior knowledge. This problem has been widely discussed: the earliest and most basic reference to it was by Pierre Simon Laplace, who stated that events about whose existence we are equally uncertain should be assigned equal probabilities. This observation was later called the principle of insufficient reason, or, as defined by Keynes, the principle of indifference:

"If there is no known reason for predicating of our subject one rather than another of several alternatives, then relatively to such knowledge the assertions of each of these alternatives have an equal probability." (Keynes, 1921)
Figure 2.2: Example of a multiple-adversary binary-utility game tree. The values in the brackets below the leaf nodes are vectors of utility values and the value next to each mid-game tree node is agents 1’s expected utility value, for that node, under the assumption of unbiased rational opponents.

The principle of maximum entropy further generalizes the above observations by stating that in a state of uncertainty, the probabilities assigned to each event should be the least biased ones, hence maximizing the entropy of the system (Jaynes, 1957a, 1957b).

We therefore define an unbiased rational agent as one that seeks to maximize its expected utility value, but is equally likely to choose between states in which that expected utility value is the same. Given no prior knowledge of the opponents’ behavior other than rationality, the assumption that they are behaving in an unbiased rational manner coincides with the principle of indifference and with the principle of maximum entropy. In a pre-final state, an unbiased rational agent selects a state from \( W(s,a) \) with equal probability if \( W(s,a) \) is not empty, and selects one from \( L(s,a) \) with equal probability otherwise. In all other states, consistent with the definition of rationality, an unbiased rational agent must select a state maximizing its expected utility value and, if several such states exist, select one of them uniformly at random.

For example, let \( A = \{a_1, a_2, a_3\} \) be a set of agents, playing a binary utility game, whose game tree is presented in Figure 2.2, and let \( a_2 \) be an unbiased rational agent. The expected utility value of agent 1, at the root
of the tree, $\hat{U}_{a_1}(r)$, can be computed by the following equation:

$$\hat{U}_{a_1}(r) = P_1(a) \cdot \hat{U}_{a_1}(a) + P_1(b) \cdot \hat{U}_{a_1}(b),$$

where $P_1(a)$, $P_1(b)$ denote the probability of agent 1 to select nodes $a$ or $b$ respectively, and $\hat{U}_{a_1}(a)$, $\hat{U}_{a_1}(b)$, are agent 1’s expected utility values at nodes $a$ and $b$ respectively. Agent 1 is a rational agent and therefore seeks to maximize $\hat{U}_{a_1}(r)$, which means that it must assign positive probabilities only to succeeding states which maximize its expected utility value. Nodes $a$ and $b$ both represent pre-final states and agent 2 is an unbiased rational agent. Therefore, at node $a$, agent 2 will select node $c$ or node $d$ with equal probability, and at node $b$ one of nodes $e$, $f$, or $g$, with equal probability. The expected utility value of agent 1, for the game presented in Figure 2.2, is therefore $P_1(a) \cdot \frac{1}{2} + P_1(b) \cdot \frac{2}{3}$, meaning that if agent 1 is rational, it must assign $P_1(a)$ a value of 0, and $P_1(b)$ a value of 1.

In other words, a rational agent in this scenario, must always choose an action that will lead it to node $b$ and maximize its expected utility value. Any strategy in which agent 1 does not constantly prefer node $b$ over node $a$ will not be rational, as it will not maximize the agent’s expected utility value. As we will now show, neither the generalized minimax, nor its variation, the Paranoid algorithm, nor the MaxN algorithm, are rational under the assumption of unbiased rational opponents.

The generalized minimax algorithm will play this game as shown in Figure 2.3. The values in the brackets are vectors of agents’ utility values. Nodes $a$ and $b$ are both MIN nodes, and therefore the algorithm will propagate a value of 0 to both nodes $a$ and $b$, and will then select one randomly. It does not strictly prefer node $b$ over node $a$, and is therefore not rational under the assumption of unbiased rational opponents. Nevertheless, if no assumptions are made, generalized minimax provides the highest guaranteed utility value.

The Paranoid algorithm (Sturtevant & Korf, 2000) assumes that all other agents have formed a coalition against the remaining agent, thus reducing the game to a two-player zero-sum game, where the agent’s paranoid utility value at a final state $s_f \in F$ is calculated by subtracting the utility value of the coalition from that of the agent:
Figure 2.3: Example of a game played according to the generalized minimax strategy. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent with the same index. Nodes $a$ and $b$ are MIN nodes and will thus propagate a value of 0 to both nodes $a$ and $b$.

$$U_{par}^{a_i}(sf) = U_{a_i}(sf) - \sum_{a_j \in A - \{a_i\}} U_{a_j}(sf).$$

The Paranoid algorithm will play this game as shown in Figure 2.4. The values in the brackets are vectors of agents' utility values and the single value below them is the Paranoid value for agent 1. Since both nodes $a$ and $b$ are MIN nodes, a value of -1 will be propagated to node $a$ and a value of -2, to node $b$. An agent playing Paranoid will therefore always choose to move to node $a$. Since agent 2 is an unbiased rational agent, if agent 1 plays according to the Paranoid algorithm it will have an expected utility value of \(\frac{1}{2}\), lower than the maximal possible value of \(\frac{2}{3}\). The Paranoid algorithm is therefore not rational under the assumption of unbiased rational opponents. Nonetheless, it has a distinct advantage over algorithms that preserve the N-players structure of the game, since, unlike them, it can use the highly effective $\alpha\beta$ pruning method to optimize the lookahead process.

The MaxN algorithm will play this game as shown in Figure 2.5. We assume the agent is aware that the other agents are unbiased and rational agents and will therefore use a uniformly distributed tie breaker when pre-
Figure 2.4: Example of a game played according to the *Paranoid* strategy. The values in the brackets are vectors of utility values, where the value at index $i$ corresponds to the agent with the same index and the single value below them is the *Paranoid* value for agent 1. Nodes $a$ and $b$ are MIN nodes, and will thus propagate a value of -1 to node $a$ and a value of -2 to node $b$.

dicting their actions. At node $a$, $MaxN$ will propagate the values of either node $c$ or $d$ with probability $\frac{1}{2}$ each to node $a$, and at node $b$ it will propagate the values of one of nodes $e$, $f$ or $g$ with probability of $\frac{1}{3}$ each. There are thus the following four possible outcomes of the propagation, with their associated probabilities:

1. Only node $a$ predicts a win for agent 1 (probability of $\frac{1}{6}$ to occur).
2. Nodes $a$ and $b$ both predict a win for agent 1 (probability of $\frac{1}{3}$ to occur).
3. Nodes $a$ and $b$ both predict a loss for agent 1 (probability of $\frac{1}{6}$ to occur).
4. Only node $b$ predicts a win for agent 1 (probability of $\frac{1}{3}$ to occur).

We make a further relaxing assumption in favor of $MaxN$: that in a case of a tie at the root level, $MaxN$ will always select the correct node (node $b$). This means $MaxN$ will select node $a$ for outcome 1, which has a probability of $\frac{1}{6}$ to occur and node $b$ for the other three outcomes, which
Figure 2.5: Example of a game played according to the $MaxN$ strategy.
The values in the brackets are vectors of utility values, where the value at
index $i$ corresponds to the agent with the same index. At node $a$ agent 2
will choose to propagate the values of either node $c$ or $d$ and at node $b$, the
values of node $e$, $f$ or $g$.

have a probability of $\frac{5}{6}$ to occur. The expected utility value of agent 1,
however, will depend on the actual move agent 2 makes and not on agent
1’s predictions of agent 2’s actions. The expected utility value of agent 1 at
the root node is therefore:

$$\hat{U}_{a_1}(r) = P_1(a) \cdot \hat{U}_{a_1}(a) + P_1(b) \cdot \hat{U}_{a_1}(b) = \frac{1}{6} \cdot \frac{1}{2} + \frac{5}{6} \cdot \frac{2}{3} = \frac{23}{36} < \frac{24}{36} = \frac{2}{3}$$

Thus, like generalized minimax and the $Paranoid$ algorithms, under the
assumption of unbiased rational opponents, $MaxN$ also does not maximize
the agent’s expected utility value and is therefore not rational.

In the next chapters we present the $Max-Prob$ algorithm for rational
decision making in an environment of unbiased rational agents. We will
present two versions of this algorithm, one for an agent that has unbounded
resources and can thus expand the entire search tree and another for an
agent with bounded resources that cannot expand the entire tree.
Chapter 3

The Max-Prob Algorithm for an Unbounded, Unbiased-Rational Agent

It is easy to see that in a binary-utility game, the expected utility value of an agent, \( a \), playing a game from state \( s \), is equivalent to its winning probability at that state. We denote this winning probability as \( P_w(s, a) \), and claim that under the assumption that all agents are unbiased rational agents, it can be recursively computed by the following equation:

\[
P_w(s, a) = \begin{cases} 
U_a(s) & s \in F \\
\max\{P_w(s', a) | s' \in \text{Succ}(s)\} & \text{Turn}(s) = a \text{ and } s \notin F \\
\sum_{s' \in C_{\text{max}}(s)} P_w(s', a) \times \frac{1}{|C_{\text{max}}(s)|} & \text{Turn}(s) \neq a \text{ and } s \notin F
\end{cases}
\]  

(3.1)

where

\[C_{\text{max}}(s) = \arg\max_{s' \in \text{Succ}(s)} P_w(s', \text{Turn}(s)).\]

Claim 1. Let \( A \) be a set of unbiased rational agents playing a binary-utility game, then for each agent \( a \in A \) and a game state \( s \in S \), \( P_w(s, a) \), as computed by Equation 3.1, is the winning probability of agent \( a \), for a game
played from state $s$.

**Proof.** We will prove the above claim by simple induction on the depth of the game tree.

**base:** The basic case is a tree of size 0. Such a tree consists of one node, which represents a final game state, $s$. Since $s$ is a final game state, there is no uncertainty regarding the outcome of the game: agents can either win or lose. According to Equation 3.1, the value assigned to each agent at this node is the agent’s utility value, which is by definition 1 if the agent has won the game, and 0 if it has lost, and therefore equal to its winning probability. Thus the claim holds for the basic case of a tree of depth zero.

**step:** Let us now assume the claim holds for trees of depth $N$ and lower and prove that it holds for trees of depth $N+1$. According to the induction assumption, for each state $s' \in $ Succ($S$), and for each agent $a \in A$, the winning probability of agent $a$ is equal to the value computed by Equation 3.1.

We’ll first prove the claim for Turn($s$) = $a$. $a$ is a rational agent and will therefore move to a state maximizing its winning probability. The acting agent’s winning probability will therefore be the maximal winning probability of its successors, exactly as defined by Equation 3.1.

Now let us prove the claim holds when Turn($s$) $\neq a$. Let $a'$ denote the agent Turn($s$). The winning probability of $a$ for this case depends on the move $a'$ will make. In addition to being rational, $a'$ is also unbiased, meaning that if several maximizing successors exist, it will choose one of them uniformly at random. The winning probability of agent $a$ at state $s$ should therefore be the average on the values assigned to it over all successors of $s$ maximizing $a'$’s winning probability, again, exactly as defined by Equation 3.1.

We now present the Unbounded Max-Prob algorithm, henceforth denoted $UMP$. $UMP$ is a lookahead-based decision-making procedure for an unbiased rational agent $a \in A$ that has unbounded resources and acts in an environment containing similar agents. Such an agent playing a binary-utility game computes $P_w(s', a)$ for all successors of the current state $s$ and moves to a state with maximal $P_w$. If several such states exist, it selects one uniformly at random.

The pseudocode of $UMP$ is presented in Figure 3.1. The $UMP$ procedure is the top level procedure that finds the best succeeding state for agent $a$
Procedure UMP($s, a$)
\[
C \leftarrow \text{Succ}(s) \\
P_w^a \leftarrow \text{Max}\{P_w(c)[a] \mid c \in C\} \\
C_{max} \leftarrow \{c \in C | P_w(c)[a] = P_w^a\} \\
\text{Return randomSelection}(C_{max})
\]

Procedure $P_w(s)$
\[
\text{If } s \in F \text{\ If } s \in F \\
\text{Foreach } a \in A \text{\ Foreach } a \in A \\
P_w[a] = U(s, a) \\
\text{Return } P_w \\
C \leftarrow \text{Succ}(s) \\
\text{curr} = \text{Turn}(s) \\
P_w[\text{curr}] \leftarrow \text{Max}\{P_w(c)[\text{curr}] \mid c \in C\} \\
C_{max} \leftarrow \{c \in C, P_w(c)[\text{curr}] = P_w[\text{curr}]\} \\
\text{Foreach } a \in A \setminus \text{curr} \text{\ Foreach } a \in A \setminus \text{curr} \\
P_w[a] \leftarrow \sum_{c \in C_{max}} \frac{1}{|C_{max}|} P_w(c)[a] \\
\text{Return } P_w
\]

Figure 3.1: A description of the UMP algorithm. The algorithm computes $P_w(s', a)$ for all successors of the current state and selects the move leading $a$ to a state with maximal $P_w$. If several such moves exist, it selects one uniformly at random.

at state $s$, and the $P_w$ procedure, is a recursive procedure that calculates and then propagates the winning probabilities of all agents at a given state, according to Equation 1.

A running example of the algorithm on the game tree discussed in Chapter 2 can be seen in Figure 3.2. The values in the brackets represent the $P_w$ values of the agents. There are three agents playing, with agent 1 invoking the UMP procedure. The vectors at the nodes represent the $P_w$ values of all three agents. At node $a$, agent 2 is the one to move. Since it cannot win at this state and is indifferent as to which of the other two agents will, it randomly selects either node $c$ or $d$. Therefore, the winning probability of agents 1 and 3 is 0.5. Similarly, the winning probabilities for node $b$ are
Figure 3.2: *Max-Prob*. Example of a complete tree. The values in the brackets are the agent's winning probabilities.

0.66, 1 and 0.66 for agents 1, 2, 3 respectively. Therefore, according to UMP, agent 1 will select the move leading it to node b, where its probability of winning is higher.

Note that, unlike other algorithms such as *MaxN*, where every value in a mid-game tree node coincides with a value of a leaf node, our algorithm can have mid-game tree-node values that do not coincide with any of the leaf nodes, for example, nodes a and b.

From a game theoretic perspective, the UMP algorithm defines a behavior strategy for an extensive form game, from which an equivalent mixed strategy can be constructed. In addition, as we will prove in this chapter, the mixed-strategy constructed based on UMP is also a Nash equilibrium solution in mixed strategies.

For an agent $a_i$ playing an extensive form game, a pure strategy $x_i$ assigns, for each state $s \in \{s|\text{Turn}(s) = a_i\}$, exactly one succeeding state $s' \in \text{Succ}(s)$. We define $X_i = \{x^i_1, .., x^i_k\}$ to be the set of all pure strategies available for agent $a_i$ and $X = X_1 \times X_2 \ldots \times X_n$ to be the set of all possible pure strategy profiles for agents $a_1, .., a_n$ respectively, where a pure strategy profile $x = (x_1, .., x_n)$ is an assignment of strategies to agents, s.t. $x_i \in X_i$. In addition, we define $u_i : X \rightarrow \{0, 1\}$ to be a function which returns for a strategy profile $x \in X$ the utility value of agent $a_i$, playing a binary-utility game where all agents play according to profile $x$.

A mixed strategy $p_i$ is a probability distribution over the pure strategies in $X_i$, such that $p_i(x_j)$ is the probability assigned to the pure strategy $x^i_j$. 
by \( p_i \). We define by \( \Delta_i \) the (infinite) set of all mixed strategies available for agent \( a_i \). A mixed strategy profile \( p = (p_1, ..., p_n) \) is an assignment of mixed strategies to the agents s.t. \( p_i \in \Delta_i \).

Let \( x^i_j(s) : S \to S \) be the result of acting according to the pure strategy \( x^i_j \) at state \( s \), and let \( X^{s \to s'}_i = \{x^i_1, ..., x^i_k\} \) be the set of all pure strategies available for agent \( a_i \) for which \( x^i_j(s) = s' \). We denote by \( p_i^{s \to s'}(s \to s') : S \times S \to [0, 1] \) the probability of agent \( a_i \) moving from state \( s \) to its successor \( s' \) under mixed-strategy \( p_i \):

\[
p_i^{s \to s'}(s \to s') = \sum_{x^i_j \in X^{s \to s'}_i} p_i(x^i_j).
\]

For an agent \( a_i \), a behavior strategy \( b_i \) defines, for each game state \( s \in S \), a probability distribution over the possible actions \( a_i \) can take at state \( s \). We denote by \( b_i(s \to s') \) the probability of agent \( a_i \) moving from state \( s \) to its successor \( s' \) under the behavior strategy \( b_i \).

Given an agent \( a \), and its behavior strategy \( b \), we say that a mixed strategy \( p \) is equivalent to \( b \), if agent \( a \)'s possible actions at each of its decision nodes are chosen independently and according to the same probability distribution as defined by \( b \). The probability this mixed-strategy assigns a pure strategy \( x \), is the product of the probabilities assigned by \( b \) to the set of actions defined by \( x \):

\[
p(x) = \prod_{(s,s') \in S, x(s) = s'} b(s \to s').
\]

**Theorem 1** Let \( G = (X_1, ..., X_n, u_1, ..., u_n) \) be an \( N \)-player extensive form game, and let \( p^* \) be the mixed strategy profile in which all players play according to a mixed strategy equivalent to the behavior strategy defined by the UMP algorithm. Then \( p^* \) is a Nash equilibrium solution for mixed strategies.

**Proof.**

In order for **UMP** to constitute a mixed-strategy Nash equilibrium, the

---

1When speaking of extended form games, one often refers to information sets rather than game states. An information set of a player is a set of game-tree decision nodes that correspond to the same game state. For consistency with our previous definitions, we will continue to refer to game states.
following inequality must hold:

\[ E_i(r, p^*) \triangleq \{p_1^*, \ldots, p_i^*, \ldots, p_n^*\} \geq E_i(r, p'^* \triangleq \{p_1'^*, \ldots, p_i'^*, \ldots, p_n'^*\}) \forall i, 0 \leq i \leq n, p^* \neq p', \]

where \( E_i(r, p) \) denotes the expected utility value of an agent \( a_i \), playing game \( G \) according to profile \( p \), from an initial state \( r \). \( E_i(r, p) \) can be computed by the following formula:

\[ E_i(r, p) = \sum_{x=(x_1, \ldots, x_n) \in X} p_1(x_1) \times \cdots \times p_n(x_n) \cdot u_i(x). \]

The expected utility value of an agent \( a \), playing a game according to a mixed strategy \( p \), from an initial state \( s \), is determined by the action the agent will take at state \( s \). The action an agent takes at a given state, is a random variable whose probability distribution is defined by \( p^\rightarrow \). Therefore, in addition to the previous formula, the expected utility of \( a \), at state \( s \), can also be recursively computed as follow:

\[ E_s(a_i, p) = \sum_{s' \in Succ(s)} E_{s'}(a_i, p) \cdot p_i^\rightarrow(s \rightarrow s'). \tag{3.2} \]

Since, in binary-utility games the expected utility value of an agent is equal to its winning probability, it is equivalent to prove that \( P_w(r, a_i) \geq E_i(r, p'^*) \). Assume by contradiction that \( UMP \) is not an equilibrium. Then there exists a mixed strategy \( p_i' \neq p_i^* \), for which \( P_w(r, a_i) < E_i(r, p'^* \triangleq \{p_1'^*, \ldots, p_i'^*, \ldots, p_n'^*\}) \). This means that there must be at least one decision node of \( a_i \) which is a root of a game-tree describing a sub-game of \( G \), and for which \( p^* \) is also not an equilibrium. Let us examine the shallowest such game-tree. Denote the root of this game-tree by \( r' \).

All of \( r'' \)'s successors are roots of trees also representing sub-games of \( G \). However, for these sub-games, \( p^* \) is still an equilibrium. Otherwise, there must exist another decision node of \( a_i \), \( r'' \), which is a root of a sub-game of \( G \) for which \( p^* \) is not an equilibrium. In addition, since \( r'' \in Succ(r') \), the game-tree rooted at \( r'' \) must be shallower than the one rooted at \( r' \), which stands in contradiction to the definition of \( r' \) and is thus impossible.

According to equation 3.1, the value of \( P_w(r', a_i) \) will be equal to the maximal value of \( r'' \)'s successors with respect to \( a_i \). If \( UMP \) is not an equi-
librium, $E_i(r', p')$ is supposed to have a value higher than that of $P_w(r', a_i)$. As can be easily deduced from equations 3.1 and 3.2, the expected value of an agent at a parent node cannot exceed the maximal value assigned to it by that node’s successors. This means that if $p^*$ is not an equilibrium, then there must be at least one successor of $r'$, $c$, for which $P_w(c, a_i) < E_i(c, p')$, but as we have already shown, $p^*$ is still an equilibrium for all of $r'$’s successors. This means that $E_i(r', p')$ cannot have a higher expected value than that of $P_w(r', a_i)$, which contradicts our initial assumption and proves that $UMP$ is an equilibrium.

■
Chapter 4

The **Max-Prob** Algorithm for a Resource Bounded Agent

It is well known that expanding the complete game tree is computationally impossible for most if not all games. Thus, a partial game tree is usually explored. In this chapter we will show a version of the Max-Prob algorithm that is suitable for bounded, unbiased rational agents. As a bounded, unbiased rational agent usually cannot expand the search tree down to the level of the final states (the leaves of the game tree), our challenge is to find a way to estimate the $P_w$ values of internal game-tree nodes that are leaves of the search tree.

Our solution is based on traditional heuristic evaluation functions, but with an additional requirement. We define a trait of heuristic functions for games, called *win-distinguishing*, and restrict our algorithm to use only such heuristic functions. A *win-distinguishing* heuristic function is one that at the end of the game assigns the winners of the game, and only them, an equal value that is higher than that of all other agents.

**Definition 1 (win-distinguishing)** Let $a_i \in A$ be an agent. A heuristic function $h^{a_i} : S \rightarrow \mathbb{R}$, is win-distinguishing iff $\forall s \in F, \ U(s, a_i) = 1 \iff h^{a_i}(s) \geq h^{a_j}(s) \ \forall i \neq j$.

Although this definition may seem restrictive, many common heuristics are *win-distinguishing*. Examples include the inverse distance to camp heuristic, used for Chinese Checkers, the material value advantage heuris-
tic used for Chess and several heuristic functions used for trick taking card games.

Our goal is to use heuristic evaluation functions in order to estimate $P_w$, the probability of winning the game. If we use a \textit{win-distinguishing} heuristic function, then the winning probability of an agent $a$ at a state $s$ is the probability of this agent to have the highest heuristic value at the end of the game played from state $s$.

Let $s \in S$ be a game state. We denote the set of all final states reachable from $s$ under unbiased rational play as $F'_s$. Given an agent $a$, and a heuristic function $h^a$, we denote by $h^a_{F'_s}$ the random variable corresponding to the distribution of $h^a$’s values over the set of all heuristic values $H^a_{F'_s} = \{h^a(s) | s \in F'_s\}$.

The winning probability of an agent $a$, at state $s$, for a specific value $x \in H^a_{F'_s}$, is the probability of agent $a$ to finish the game with a heuristic value of $x$, while all other agents finish the game with a lower value. The overall winning probability of an agent $a$, at state $s$, is the accumulation of agent $a$’s winning probabilities, for each of its possible final heuristic values. This probability can be computed according to the following equation:

$$P_w(s, a) = \sum_{x \in H^a_{F'_s}} P(h^a_{F'_s} = x, h^{a'}_{F'_s} \leq x, \forall a' \neq a). \quad (4.1)$$

If we make a relaxing assumption of independency of the agents’ heuristic values at the end of the game, we can evaluate $P_w(s, a)$ as follows:

$$\hat{P}_w(s, a_i) = \sum_{x \in H^a_{F'_s}} P(h^a_{F'_s} = x) \cdot \prod_{a' \neq a} P(h^{a'}_{F'_s} \leq x). \quad (4.2)$$

For a continuous distribution, we can evaluate $P_w(s, a)$, given only the probability density function of $h^a_{F'_s}$ for every agent in the game. We denote this density function by $f^a_{h^a}$.

$$\hat{P}_w(s, a) = \int_{x=-\infty}^{x} f^a_{h^a}(x) \cdot \prod_{a' \in A \setminus \{a\}} \int_{y=-\infty}^{x} f^{a'}_{h^{a'}}(y) dy \, dx. \quad (4.3)$$
The external integral sums agent $a$’s winning probabilities, for each of its possible heuristic values. This is in fact the probability of the agent to reach a final state in which it has some heuristic value $x$, while all other agents have a lower heuristic value. The internal integral is the cumulative distribution function associated with $f_{h_a}$. It evaluates $P(y \leq x)$, where $y$ is the value associated with agent $a'$ and $x$ is the value associated with agent $a$. This formula applies to all forms of continuous distribution functions. In case of a density function that is not integrable, as in the case of a normal distribution, for example, numerical methods can be applied in order to estimate the value of the integral.

Considering the above observations, we suggest Bounded Max-Prob, henceforth denoted BMP, an adaptation of the algorithm presented in Chapter 3. The algorithm will remain the same except for the evaluation method of a search tree leaf, which will first produce a vector of the agents’ heuristic values, and then infer from these values the probability of each agent to win. The values propagated up the tree will be the estimated winning probabilities. The algorithm is listed in Figure 4.2.

An example of how BMP assigns values in a partial game tree is presented in Figure 4.1. For simplicity, we assume that $f_{h_a}$ is uniform over an interval $[l_{h_a}(s), u_{h_a}(s)]$. The values in the round brackets under the leaves are vectors of intervals, representing the bounds of the uniform distribution.
for each agent, where the value at index $i$ corresponds to the agent with the same index (counting from 1). For the case of a uniform distribution, the winning probability of each agent is estimated by the accumulation of winning probabilities for each possible value in the range $[l_{h^a}(s), u_{h^a}(s)]$. The internal integral will therefore be computed according to the cdf of the uniform distribution:

$$
Pr(h^a(f) < x) = \begin{cases} 
0 & x > u_{h^a}(s) \\
\frac{x - l_{h^a}(s)}{u_{h^a}(s) - l_{h^a}(s)} & x \in [l_{h^a}(s), u_{h^a}(s)] \\
1 & x < l_{h^a}(s)
\end{cases} \quad (4.4)
$$

At node $c$ the value of agent 1 will always be higher than that of all the other agents. Therefore, it will have a winning probability of 1 while all the other agents will have a winning probability of 0. Each of the other leaf nodes contain intervals that intersect with each other. We will need to compute the winning probabilities, according to equations 4.3 and 4.4. At node $d$, for example, agent 3’s final heuristic value will always be lower than that of agent 1. As for agent 2, its final value will always be lower if agent 1’s value will fall in the interval $[5, 6]$, but not necessarily for the interval $[4, 5]$, and so

$$
\hat{P}_w(b, a_1) = \int_{x=4}^{5} \frac{1}{6} \cdot \frac{x - 3}{5 - 3} \cdot 1 \, dx + \int_{x=5}^{6} \frac{1}{6} \cdot 1 \cdot 1 \, dx = \frac{7}{8}.
$$

The value for agent 2 can be similarly computed, and so can the values at nodes $e$ and $f$. The winning probabilities are then propagated up the tree according to the BMP algorithm. At node $a$, agent 2 has only one maximal child, but for node $b$ both children maximize the agent’s winning probability and so the expected probability values propagated for agents 1 and 3 will be the average of their values at nodes $e$ and $f$. 

23
**Procedure** BMP\((s, a, \text{depth}, h)\)

\[
C \leftarrow \text{Succ}(s)
\]

\[
\hat{P}_w^a \leftarrow \text{Max}\{\hat{P}_w(c, \text{depth}, h)[a] \mid c \in C\}
\]

\[
C_{\text{max}} \leftarrow \{c \in C \mid \hat{P}_w(c, \text{depth}, h)[a] = \hat{P}_w^a\}
\]

**Return** randomSelection\((C_{\text{max}})\)

**Procedure** EstimateWinProb\((a, A, h)\)

// Compute agent \(a\)’s winning probability
// according to equation 4.2 or 4.3.

**Procedure** \(\hat{P}_w(s, \text{depth}, h)\)

If \(s \in F\)

Foreach \(a \in A\)

\[
\hat{P}_w[a] = U(s, a)
\]

**Return** \(\hat{P}_w\)

If \(\text{depth} = 0\)

Foreach \(a \in A\)

\[
\hat{P}_w[a] = \text{EstimateWinProb}(a, A, h)
\]

**Return** \(\hat{P}_w\)

\[
C \leftarrow \text{Succ}(s)
\]

\[
\text{curr} = \text{Turn}(s)
\]

\[
\hat{P}_w[\text{curr}] \leftarrow \text{Max}\{\hat{P}_w(c, \text{depth} - 1, h)[\text{curr}] \mid c \in C\}
\]

\[
C_{\text{max}} \leftarrow \{c \in C \mid \hat{P}_w(c, \text{depth} - 1, h)[\text{curr}] = \hat{P}_w[\text{curr}]\}
\]

Foreach \(a \in A \setminus \text{curr}\)

\[
\hat{P}_w[a] \leftarrow \sum_{c \in C_{\text{max}}} \frac{1}{|C_{\text{max}}|} \hat{P}_w(c)[a]
\]

**Return** \(\hat{P}_w\)

Figure 4.2: A description of the BMP algorithm. The algorithm computes \(\hat{P}_w(s', a)\) for all successors of the current state and selects the move leading \(a\) to a state with maximal \(\hat{P}_w\). If several such moves exist, it selects one uniformly at random.
Chapter 5

Learning the Heuristic Distribution

The performance of BMP in a resource bounded environment highly depends on the accuracy of $P_w$’s estimation. As a result, it also depends on the estimation of the final heuristic value’s distribution. This distribution can be estimated using domain-specific knowledge and, for a parametric distribution, optimized by parameter tuning. Alternatively, it can be inferred using machine learning techniques. A distribution can be learned by several methods, depending on the nature of the distributed values. Because most heuristic functions assign a finite and relatively small set of values to final states, we propose to perform a structured induction process, the result of which will be a model that will produce for a state $s$, an agent $a$, and its heuristic function $h^a$, a prediction of the distribution of $h^a$’s values over the set $H^a_{F'}$, as defined in Chapter 4.

The learning process will consist of several stages. First, a set of instances will need to be collected and labeled. The labeled instances can then be used by an induction process to create the probability prediction model, which in turn will be incorporated into the decision making process. The same scheme can be applied in order to infer the parameters of a continuous distribution, assuming we know its class. The rest of this chapter discusses each of these stages with respect to a given agent and its associated heuristic function $h^a$.

The examples for the induction process are labeled feature vectors representing various states in the game. In an ideal learning process, the label
of each example would be the distribution of $h^a$’s values over the set $H^a_{F_s}$. However, since constructing $H^a_{F_s}$ usually requires unbounded resources and is thus impossible, we propose a method that will use examples with estimated labels. We first describe how to produce the set of instances and their estimated labels, and then show how to build and then use a distribution prediction model, taking into account the problems that arise due to the use of estimated labels.

We propose the following procedure for constructing a set of instances along with their associated estimated labels:

1. Construct a set of game logs $L$, where each log $l \in L$ is a list of states $l = \langle s_1, ..., s_n \rangle$, $s_i \in S, s_n \in S_f$, describing a self-played game of a resource-bounded, unbiased rational agent, maximizing its associated expected heuristic value (instead of a winning probability, which is still unavailable).

2. Let $f : S \rightarrow \mathbb{R}^n$, be a feature extraction function that converts states to feature vectors. Note that this function does not have to be injective. Generate for each log $l \in L$ the multiset of pairs $M_l = \{\langle f(s_i), h^a(s_n) \rangle | 0 < i < n \}$. Let $M_L = \bigcup_{l \in L} M_l$, denote the multiset sum of all pairs collected from $L$.

3. Let $V_l$ be the set of feature vectors associated with the non-final states in $l$, and let $V_L$ be their union. Construct for each feature vector $v \in V_L$ the multiset $D_v = \{(h, m(h)) | (v, h) \in M_L \}$, where $m(h)$ denotes the number of occurrences of $h$ in $D_v$.

4. Create the estimated label $P_v$ for each feature vector $v \in V_L$, s.t, $P_v(h) = \frac{m(h)}{|D_v|}$, where $|D_v| = \sum_{(h, m(h)) \in D_v} m(h)$.

If we are interested in inferring the parameters of a continuous distribution, such as expected value or standard deviation, then at this point there needs to be an additional stage where these parameters are computed from the distribution. In this case the examples of the induction process will still be the feature vectors of states, but their labels will be tuples of the distribution parameters’ values. Alternatively, a separate classifier can be used for each parameter.
The quality of the labels estimated using our method highly depends on the number of examples used to construct it, which in turn depends, among other things, on the function used to convert states to feature vectors. If we use an injective conversion function, each state will be represented by a unique feature vector. The size of the labeled example set supplied to the induction process will be equal to the number of unique sampled game states, and the number of examples used to construct the estimated labels of these examples will be the number of times the same state has been sampled. If we use a noninjective conversion function, several states will be represented by the same feature vector. The size of the labeled example set supplied to the induction process will be smaller, but the number of instances used to infer each example’s label will be larger, hopefully resulting in higher quality labels.

A good conversion function should therefore be one that produces high quality labels, by achieving an optimal balance between the number of labeled examples used for induction and the ones used for labeling. It should also produce feature vectors with high predictive power and should be easy to compute, especially when the induced model is used by a time-bounded agent.

Once the examples have been tagged, an induction algorithm can be used to construct the distribution model. Theoretically, any induction model that can handle structured labels for a discrete distribution, or continuous labels for a continuous distribution, can be applied to this problem. However, when estimating feature labels as we did, the quality of these estimates will probably not be uniform over the set of examples.

Since we are dealing with labels of different qualities, we needed an induction method that will easily allow for higher quality labels to receive more weight in the classification process. In our experiments, we therefore chose to use the K-Nearest Neighbor algorithm (Aha, Kibler, & Albert, 1991), and assign each neighbor, \( N = (v, P_v) \), a weight \( w(N) \) that is proportional to \( |D_v| \), the number of examples used to construct its label. Let \( v \in V_L \) be an unlabeled feature vector. Let \( N_1, ..., N_k \) be its \( k \) nearest neighbors. The
inferred label of $v$, $P_v$, will be the weighted average of its neighbors’ labels:

$$P_v(h) = \frac{\sum_{i=1}^{k} P_i(h) \cdot w(N_i)}{\sum_{i=1}^{k} w(N_i)}.$$ 

Once the distribution prediction model is constructed, it can be used by $BMP$ during the decision making process. The algorithm will receive a state for which the agent’s action is required. A standard search tree will then be expanded according to the agent’s available resources. When the agent will reach a state it can no longer expand due to limited resources (leaf state of the search tree), the estimated winning probabilities will need to be computed, according to Equation 4.2 for a discrete distribution or Equation 4.3 for a continuous distribution. The components required for these calculations will be retrieved from the distribution prediction model. Once the estimations of the winning probabilities are computed, the decision process will continue according to the $BMP$ algorithm, and the agent’s action choice will be produced. A data-flow diagram of both the learning process and the decision making system is shown in Figure 5.1.
Figure 5.1: Data-flow of the probability learning process, and how the learned model is incorporated to the BMP algorithm.
Chapter 6

Empirical Evaluation

In order to assess the performance of the BMP algorithm, we compared it to several competing algorithms over multiple domains and multiple test setups. The results of these experiments are discussed in this chapter.

6.1 Experimental Methodology

Lookahead-based decision-making algorithms are most commonly tested by evaluating their performance in turn-taking, extensive form games. We also chose to use games as a testing platform. Since we proposed BMP as a general decision making algorithm, rather than a game specific one, we had to assess its performance on multiple test domains. The experiments were therefore conducted on four different games, two board games and two card games.

We will now provide a short description of the domains used for testing. A fuller description of the complex rules of the board games is given in Appendix A.

1. Chinese Checkers

Chinese Checkers is a strategy game for up to six players. The winner of the game is the first player who moves all ten of her pawns from her home camp, located at one pit of a hexagram, to her destination camp, located at the opposite pit. Our heuristic was simple and consisted of the accumulated distance of the pawns from their destination camp.
2. *Abalone*

Abalone is another strategy game. Its board is a hexagon containing 61 slots. In the three-player version that we play, each player starts the game with eleven marbles arranged in two adjunct rows, one containing 5 marbles and another containing 6. The winner of the game is the first player who manages to push six of her opponents' marbles out of the game board. The heuristic function we used contained the number of opponents’ pawns the player pushed off the board and the pawns’ arrangement on the board (vicinity to the edges, number of neighbors around a marble, etc.).

3. *Perfect-information Simplified Spades*

Spades is a trick taking card game for which the spades suite is always a trump. The game has both a partnership and a solo version; we will focus on the solo version. Originally, the game is a hidden information game. We used the perfect information version, where the players’ hands are visible to all. The game ends when one player reaches a predefined score, or in some versions, after a fixed number of hands. In our version a game consisted of a single hand. In the original version of the game, the scoring is affected by the number of tricks a player took and the bid she placed at a preliminary phase. We implemented a previously suggested simplification which omits the bidding phase and awards 1 point for every trick taken (Sturtevant, 2003). The heuristic function we used was constructed from the number of tricks taken and the composition of the player’s hand.

4. *Perfect-information Hearts*

Hearts is another hidden information, trick taking card game. Each heart card taken is worth 1 penalty point and the queen of spades is worth 13. The winner of the game is the one with the lowest number of penalty points at the end of the game. As in Spades, the game consisted of a single hand, and the perfect information version of it was used. Our heuristic function consisted of the number of penalty points taken by the players and the composition of the player’s hand.

5. *T-Spades*

T-Spades is a tournament version of the the single-hand version of
spades presented above. A tournament game consists of 100 single hand games. The score of a player at the end of the game is the number of tricks she took throughout the entire game. As in the single hand version, the winner is the one with the highest score.

6. **T-Hearts**

T-Hearts is a tournament version of the the single-hand version of hearts presented above. A tournament game consists of 100 single hand games. The players’ penalty points are accumulated throughout the game. The winner is the player with the lowest number of penalty points at the end of the game, again exactly as in the single hand version.

We compared the performance of BMP to that of three other algorithms. In each game, the same heuristic function was used for all agents and a fixed depth limitation on the search tree was imposed. The competing algorithms were MaxN, Paranoid and the recently developed MP-Mix algorithm (Zuckerman, Felner, & Kraus, 2009), which alternates between playing MaxN, Paranoid, and an offensive strategy that attacks the leading agent. It does so by examining the difference between the heuristic value of the leading agent and that of the runner up. If the agent invoking the algorithm is the leading agent and the said difference is higher than a predefined threshold, \( T_d \), a paranoid strategy is taken. Otherwise, if the leading agent is another agent and this difference is higher than a second threshold, \( T_o \), an offensive strategy is taken. In all other cases, the MaxN strategy is played.

For the BMP algorithm, we assumed a continuous and uniform heuristic distribution in all but 1 setup, in which this distribution was inferred using machine learning techniques. Each of our heuristic functions can be bounded by a constant number. We used these numbers as the upper bounds of the uniform distribution and the agent’s heuristic value at the current state as the lower bound. We broke ties only at the root level, where we preferred states with a higher heuristic value. We did this for all games except hearts, where the mid-trick heuristic value of a hand was shown to be uninformative to misleading. In addition, for the card games, we did not allow the search to end in the middle of a trick.

In the board games, each experiment compared the algorithms’ performance on 100 games for each of the possible player orderings. The card
game setup was a bit more complex. Each setup compared the algorithms’ performance on 100 different hands played using two different versions for each game. In the first, we ran 100 single hand games for each player ordering, and in the second (the tournament version of each game), a single 100 hands tournament game for each player ordering. We made sure the same 100 hands were played for every ordering so as to minimize the effect of chance.

For the $M_p$-Mix algorithm, we performed parameter tuning on the card game domains. For the non-tournament setups, we tried 196 combinations of the offensive and defensive thresholds for each game. The values ranged from 0 to the maximal possible heuristic value, in increments of 100 for spades and 200 for hearts. We performed parameter tuning for $M_p$-Mix in the tournament setup as well, although not as exhaustively as in the non-tournament setup, due to computational bottleneck and limited resources. We used the optimal settings found for the non-tournament setups as a basis, and performed some manual tuning from there. We could not parameter-tune the board games due to their computational complexity, so we set the thresholds to values that indicate a notable improvement in an agent’s status. In Abalone, this value was equal to the reward gained by pushing an opponent’s marble off the board and in Chinese checkers, a three slot distance gap.

### 6.2 Multiple Competitor Comparison

This experiment comprehensively compared all 4 competing algorithms. A total of 2400 games was run for each testing domain. For the card games setup, the depth of the search tree was limited to 8 plies and all four algorithms participated in each game. The board games have a much higher branching factor than the card games and so their setup was slightly different. Each game tested three of the four competing algorithms, and the depth of the tree was limited to 4.

The results are presented in Table 6.1. The values outside the brackets are win percentages and the values inside the brackets are the algorithms’ performance ranks according to win percentages for each game. We ran the Friedman test on the results and found them to be statistically significant for a p-value of 0.05.
<table>
<thead>
<tr>
<th>Game/Algorithm</th>
<th>BMP</th>
<th>MaxN</th>
<th>Paranoid</th>
<th>MP-Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese Checkers</td>
<td>52.00 (1)</td>
<td>21.50 (3)</td>
<td>24.50 (2)</td>
<td>2.00 (4)</td>
</tr>
<tr>
<td>Abalone</td>
<td>45.71 (1)</td>
<td>5.00 (4)</td>
<td>44.04 (2)</td>
<td>5.25 (3)</td>
</tr>
<tr>
<td>Spades</td>
<td>25.77 (1)</td>
<td>24.78 (3)</td>
<td>24.48 (4)</td>
<td>24.97 (2)</td>
</tr>
<tr>
<td>Hearts</td>
<td>29.86 (1)</td>
<td>25.61 (3)</td>
<td>18.80 (4)</td>
<td>25.73 (2)</td>
</tr>
<tr>
<td>T-Spades</td>
<td>40.00 (1)</td>
<td>20.00 (3)</td>
<td>16.00 (4)</td>
<td>24.00 (2)</td>
</tr>
<tr>
<td>T-Hearts</td>
<td>79.17 (1)</td>
<td>12.50 (2)</td>
<td>0.00 (4)</td>
<td>8.33 (3)</td>
</tr>
<tr>
<td><strong>Average Win Percentage</strong></td>
<td><strong>45.42</strong></td>
<td><strong>18.23</strong></td>
<td><strong>21.30</strong></td>
<td><strong>15.05</strong></td>
</tr>
<tr>
<td><strong>Average Rank</strong></td>
<td>1.00</td>
<td>3.00</td>
<td>3.33</td>
<td>2.83</td>
</tr>
</tbody>
</table>

Table 6.1: Algorithm performance for each domain, multiple comparison setup. The values outside the brackets are win percentages and the values inside the brackets are the algorithms’ performance ranks according to win percentages for each game.

It can be seen that the BMP algorithm outperforms its opponents in all tested domains. If we look at average win percentages, we see that the Paranoid algorithm performed best after BMP, followed by MaxN, and then Mp-Mix. If we look at average ranking however, BMP still outperforms its opponents but the runner up in this case is the Mp-Mix algorithm, followed by MaxN and then Paranoid.

In the board game platforms, BMP performed a lot better in Chinese checkers than in Abalone, and in the card games it performed better in Hearts than in Spades. These differences are probably due to the difference between the predicted winning probability of the agents and the actual ones. In the card game domain, BMP performed best for tournament games. One explanation for this is that while in a single hand game, an agent’s heuristic value and its probability to win can be highly correlated, this correlation diminishes when many hands are played. Therefore, when a single hand game is played, an agent can focus only on increasing its own utility value, but when many hands are played, the agent must consider the heuristic values of the other agents as well.

The Mp-Mix algorithm performed a lot better in the card game domains, where parameter tuning was performed, than in the board game domains, where the parameters were manually set. This demonstrates the substantial benefit of parameter tuning to the performance of Mp-Mix.
6.3 Pairwise Comparisons

In this set of experiments we compared the performance of BMP to each of the competing algorithms separately. Each test consisted of three competing players, invoking only two different algorithms. Table 6.2 presents all the possible orderings for this configuration. Algorithm A was always BMP and Algorithm B was one of MaxN, Paranoid, or Mp-Mix. The testing domains used for this setup were only the card game domains, and the depth of the search tree was limited to 6 plies. As in the previous setup, we performed the Friedman test and found the results, presented in Table 6.3, to be statistically significant for a p-value of 0.05.

<table>
<thead>
<tr>
<th>Game/Algorithm</th>
<th>BPM - MaxN</th>
<th>BPM - Paranoid</th>
<th>BPM - Mp-Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spades</td>
<td>52.14</td>
<td>50.84</td>
<td>52.29</td>
</tr>
<tr>
<td>Hearts</td>
<td>59.57</td>
<td>50.38</td>
<td>56.97</td>
</tr>
<tr>
<td>T-Spades</td>
<td>66.67</td>
<td>83.33</td>
<td>66.67</td>
</tr>
<tr>
<td>T-Hearts</td>
<td>100.00</td>
<td>57.14</td>
<td>100.00</td>
</tr>
<tr>
<td>Average win %</td>
<td>69.60</td>
<td>60.43</td>
<td>68.98</td>
</tr>
</tbody>
</table>

Table 6.3: Winning percentage for each game, pairwise comparison setup.

As in the previous setup, BMP outperforms all its opponents in all tested domains and, again as in the previous setup, it performed best in the tournament domains. Paranoid posed the toughest competition for BMP, followed by MaxN and Mp-Mix, where Mp-Mix performed slightly better than MaxN.
Paranoid might have performed so well because the coalition assumption is implemented in the algorithm partly by creating a new heuristic evaluation function that is constructed from the heuristic values of all other agents. This new heuristic function can be a lot more informative than the functions used by the rest of the agents.
Chapter 7

Empirical Study on the Learning Algorithm

In the previous chapter we examined the performance of BMP when the distribution of the final heuristic values was assumed to be uniform. In this chapter, we will examine the performance of BMP when this distribution is inferred. To this end, we implemented the learning algorithm presented in Chapter 5.

We chose to use three-player single-hand spades as a testing platform. Our heuristic function implementation for this games allows 17 possible final heuristic values. We thus chose to use equation 4.2 to calculate the winning probabilities, and therefore trained a classifier to infer discrete probability distributions, as opposed to continuous density functions. The heuristic function we used also had another property: for final game states, its sum over the agents’ values was always constant. Therefore, despite the relaxing assumption of independence, we factored out events where the constant sum property did not hold, since these events could not occur. BMP ’s performance was again compared to the 3 algorithms used previously, with the pairwise comparison setup. We factored in a uniform prior distribution component, to compensate for cases of very low certainty regarding the labeling of the instance’s neighbors. We represented the prior distribution component as a dummy neighbor. The weight of this dummy neighbor determined the learning rate of the system. We experimented with several such values. In the results presented here, it was set to 100.
Figure 7.1: The performance of BMP as a function of learning resources.

Figure 7.1 illustrates the win percentage of BMP when playing against MaxN, Mp-Mix and Paranoid. The graphs show a typical learning curve. We start with no examples, so the heuristic distribution is assumed to be uniform. There is a strong decline in performance when we combine learning from a small number of examples, followed by a consistent improvement as the number of training examples grows, which is correlated with the decreasing effect of the uniform component. The learning process converges at approximately 1000 examples.
Chapter 8

Related Work

In 1950, Shannon proposed the minimax algorithm for two-player games (Shannon, 1950). An obvious extension of this algorithm to N-Player games is to treat every opponent’s decision node as a MIN node (Mutchler, 1993). As in the two-player version, this strategy provides a maximal payoff guarantee given the worst case scenario. However, playing the minimax strategy is equivalent to assuming a coalition of N-1 players has been formed to play against the remaining player, an assumption that is overly conservative or even paranoid in a game with more than two-players.

Luckhart and Irani (1986) presented the MaxN algorithm, in which evaluation of a leaf in the game tree produces a vector of N values. Each component in the vector is the utility value of the corresponding player for that state. The algorithm also defines a propagation rule which states that each agent in its turn selects a move maximizing its individual utility value, without considering the utility values of the other agents. This means that there is an unstated assumption that the agents are indifferent to each other’s performance, an assumption that does not hold for adversarial games.

A major drawback of the MaxN algorithm is that the highly effective αβ pruning method, which can be applied to minimax search trees, cannot be applied to MaxN search trees. Although several pruning methods have been proposed, including in the original paper itself (Luckhart & Irani, 1986; Korf, 1989, 1990, 1991; Sturtevant, 2005), none were as effective as αβ. To overcome this problem in multiplayer games, Sturtevant (2000, 2003) proposed the Paranoid algorithm, an implementation of N-players minimax that reduces an N-player game to a two-player zero-sum game, thereby allowing αβ
pruning to be applied to the search tree. This is a distinct advantage of the algorithm; however, the paranoid coalition assumption can lead to suboptimal play (Sturtevant & Korf, 2000). Furthermore, using Paranoid renders all individual opponent modeling schemes unusable, even if available.

Zuckerman et al. (2009) presented the MP-Mix algorithm, which implements a mixed strategy approach. When it is an agent’s turn to act, it examines the difference between the leading agent’s heuristic value and that of the runner up. It then decides accordingly whether to use an offensive strategy, a paranoid strategy, or the default MaxN strategy. This algorithm overcomes MaxN’s inherent problem of the agents’ indifference to each other by explicitly examining the relation between the leading agent and that of the runner up. However, this solves the indifference problem only in part since it does not take into account the status of the non-leading agents, whereas our Max-Prob algorithm takes into account the relative state of all agents by calculating winning probabilities.

There are several approaches to handle resource limitation in a decision making algorithm. One such approach, used by our BMP algorithm, is to expand all reachable states up to a certain depth, thus creating a subtree of the complete game-tree. Alternative approaches include selective-deepening and sampling methods in which searching deeper rather than wider is preferred. One sampling based algorithm that has been successfully applied to multiplayer games is the UCT algorithm. UCT is a Monte-Carlo based game tree search algorithm. In the first stage of the algorithm a tree is explored up to a certain depth. In the next stage, an action is selected and the rest of the game is simulated either by pure random play, or by $\varepsilon$-greedy random play, incorporating some domain specific heuristic evaluation function. A reward value is propagated up the tree and a move maximizing this reward is selected. The reward value is a combination of the average payoff gained by the agent and a statistical component designed to provide information regarding sampling frequency. Cazenave (2008) applied UCT to multiple-player Go, and Sturtevant (2008) showed it outperforms MaxN in a restricted version of Chinese checkers and is also a decent competitor in perfect information hearts and spades. Several enhancements to UCT have been recently proposed as well (Nijssen & Winands, 2011). Unlike pruning methods that only omit exploring a branch in the tree if it is redundant, UCT does not attempt to develop a full width tree. It prefers to explore
deeper than wider, believing that the true payoff of a random play is more informative than the heuristic value of a mid-game tree node. Like MaxN, UCT does not consider the payoff of other agents; its origin is in the multi-armed bandit problem, where an agent’s goal is to maximize its payoff when playing a gambling machine. In the future, it would be interesting to adapt UCT to work for a winning probability maximizer, such as Max-Prob, and compare it to the classical algorithm.

Most multiple-player games are games with imperfect information. Algorithmic solutions for such games have to deal not only with the complexity of a multiple-adversary environment but also with the added complexity of the hidden information. As a result, several algorithms which deal specifically with hidden information games have been proposed, both for general hidden information games (Blair, Mutchler, & van Lent, 1996), and for specific games such as Hearts (Sturtevant & White, 2006; Perkins, 1998), Bridge (Amit & Markovitch, 2006; Frank, Basin, & Bundy, 2000).

Both versions of our algorithm propagate win probabilities. The bounded version of Max-Prob, BMP, also evaluates the leaves of the search tree as a distribution over a range of values and infers winning probabilities from these distributions. Other probabilistic methods, and methods which use ranges of values, have been used mainly as a tool for better incorporating opponent models in the search algorithm, or for the purpose of selective deepening of the search tree. Most such methods were implemented only in single-adversary environments. The B* algorithm proposed by Berliner (1979) defines for each node an interval, bounded by an optimistic and a pessimistic value. These intervals are then used to selectively expand nodes. The interval values of a node are derived from its children’s values and are backed up the tree, in an attempt to narrow the range of the root’s interval until it is reduced to a single value.

Palay (1985) extended the B* algorithm to a version where the evaluation of a leaf produces probability distributions, which are then propagated up the tree using the product propagation rule. Baum and Smith (1997) proposed a Bayesian framework to selectively grow the game tree and evaluate tree nodes. Their algorithm, called BP, describes how to evaluate the uncertainty regarding the evaluation function. Instead of assigning single values to nodes, discrete distributions on the possible values of the leaves are assigned and propagated up the tree, again using the product propagation rule.
rule. These algorithms are similar to the bounded version of the Max-Prob algorithm in that the leaf evaluation procedure does not produce a single value. Unlike BMP, however, they continue to propagate non-discrete values that propagate vectors of discrete winning probabilities deduced from the evaluation of the leaves.

The $M^*_\epsilon$ algorithm (Carmel & Markovitch, 1996b) is an extension of the $M^*$ algorithm (Carmel & Markovitch, 1996a, 1996c), which is a generalization of minimax that uses an arbitrary opponent model to simulate the opponent’s search. The algorithm addresses the problem of an agent that is uncertain as to the evaluation function of its opponents and thus tries to contain it within an error bound, similar to the BMP algorithm. Unlike the Max-Prob algorithm, however, $M^*_\epsilon$ does not convert these intervals to single values but rather propagates intervals of values.

Methods which propagate probabilities not for the sake of selective deepening or as part of an opponent modeling algorithm were discussed mostly in the domain of two-player games. Pearl (1984) suggested assigning winning probabilities to the leaves and propagating them using the product rule. He also suggested a way to compute these probabilities using statistical records for chess, later investigated by Nau (1983) for two-player games.

The Expectimax algorithm (Michie, 1966), is an enhancement of the minimax algorithm for two-player zero-sum games, where chance is involved. The algorithm defines a new type on node, in addition to the existing MIN and MAX nodes, called a chance node. At a chance node, the value propagated up the tree is the expected value of the children node payoffs. An edge from a chance node to a MIN or MAX node represents the result of a probabilistic event, such as a dice roll. The Max-Prob algorithm is similar to Expectimax in that it propagates expected values. It is different in that it is a multiple adversary solution scheme, which propagates vectors of values rather than single values. In addition, the uncertainties in a multiple-adversary environment are not due to external factors such as a dice roll that defines the agent’s possible actions, but are rather due to the decisions of the agents themselves. Therefore, in the Max-Prob search tree, every decision node corresponds to an agent and there are no MIN, MAX, or chance nodes.

In multiple-adversary games, there are uncertainties regarding agents’ actions that do not exist in the simpler single-adversary environment. Some agents can form coalitions, others can be biased against or for another agent,
as can occur for example, in tournament games. Many of these added complexities can be addressed using various forms of opponent models, and indeed several algorithms in the field of opponent modeling have been proposed. Sturtevant and Bowling (2006) present the Soft-maxn algorithm, an extension of MaxN that propagates sets of MaxN vector values instead of choosing just one, in case of a tie or several possible opponent models. The Soft-MaxN concept was later generalized for the case where a probability distribution is known for the candidate opponent models in the Prob-MaxN algorithm (Sturtevant, Zinkevich, & Bowling, 2006). This method combines the idea of Soft-MaxN to propagate sets of vectors and the probabilistic opponent model search for two players proposed by Donkers. Like Max-Prob, Prob-MaxN propagates vectors of expected utility values. However, in the case of Prob-MaxN, these vectors represent the expected utility value, when uncertainty exists regarding the opponents’ strategy in a resource bounded game. Some of the additional work done in the field of multiplayer opponent modeling includes that by Wilson et al. (2011), Amit and Markovitch (2006), and Billings et al. (1998)
Chapter 9

Discussion

This paper presented a new framework for lookahead-based decision making in multiple-adversary binary-utility games. We defined a binary utility game as a game in which an agent can have only one of two possible values, 0 for losing and 1 for winning. In addition, we defined an adversarial binary utility game as one where at each final state, there must be at least one losing and one winning agent. We defined a new type of rational agent called an unbiased rational agent: such an agent seeks to maximize its utility value, but is equally likely to choose between states with the same utility value. We showed that in an environment of unbiased rational agents, the behaviors defined by existing algorithms do not maximize the agent’s expected utility value and are therefore not rational. In light of that, we proposed a new algorithm, Max-Prob. Max-Prob is a lookahead based decision algorithm that strives to maximize the agent’s winning probability, which in binary utility games is equal to the expected utility value.

We presented two versions of the Max-Prob algorithm, one for an agent with unbounded resources, and another for an agent with bounded resources. We proved that the unbounded version, the UMP algorithm, constitutes a mixed-strategy Nash equilibrium. After analyzing the theoretical case of an unbounded resources environment, we continued to present the BMP algorithm for an agent with bounded resources. We introduced a way to evaluate the winning probability of an agent according to its heuristic value and that of its opponents. For this we placed only one restriction on the heuristic function: that at the end of the game it must assign the winners of the game, and only them, a value higher than that of all other agents. We
defined such heuristic functions as *win-distinguishing* heuristic functions. Empirical results were presented on a series of domains and for various experimental setups, and showed that *BMP* outperforms its opponents.

The computation of the winning probability relied on the existence of a distribution function which assigned probabilities to the possible end-of-game values. We used two different methods to compute this distribution. In the first method, given no additional information, we assumed it to be uniform. In the second method, we used records of games played by resource-bounded, unbiased rational agents and used these records to train a classifier, which during the game provided us with the required probabilities.

Since in a multiple-adversary game, uncertainties regarding agent decisions remain even if all agents are guaranteed to be rational, we suggested a mixed strategy approach, unlike the *MaxN* and *Paranoid* algorithms, which presented a pure strategy. Our algorithm is based on the assumption that all agents are unbiased rational agents. This assumption is based on the principle of indifference, suggested and supported by Laplace, Leibniz, Jaynes, and others. Like other methods, our bounded version algorithm uses heuristic functions to evaluate the leaves of the search tree. But unlike other methods, we transform these heuristic values to estimated winning probabilities. Although not accurate, an estimated winning probability is a well-defined notion, as opposed to heuristic evaluations whose notion of ”good” and ”bad” is vague. Furthermore, since winning probability is a well-defined notion, we can use machine learning techniques in order to infer it for *any* given heuristic function. This reduces the dependency of the algorithm on domain-specific knowledge.

Pruning was not presented in this work. But we believe that existing methods for pruning of both vector propagating algorithms (Sturtevant, 2003, 2003, 2005) and probabilistic search trees (Ballard, 1983) can be adapted to work for our *BMP* algorithm.

Sampling methods, can also be adapted to work in the environment described in this paper. One of the main sampling methods used for game tree search is the *UCT* algorithm (Kocsis & Szepesvári, 2006). Although the *UCT* algorithm was originally developed for payoff maximizing agents, it can be applied for unbiased rational agents playing a binary utility game, when the selection of an action at the root level is unbiased rational. The rest of the algorithm can remain as is; however, the low variance of the
utility values in a binary-utility game may make the reward values less informative. An alternative approach would be to combine UCT with the winning probability estimation method presented in Chapter 4, and try to infer winning probabilities through sampling.

In binary utility games there is no reward for being a single winner, or for not being a single loser. Games of this kind, where the agents are encouraged to be greedy, can be modeled as zero sum games. Each agent will have its binary utility value and its greedy utility value. The greedy value of all agents who won will sum up to 1, and the value of all agents who lost to -1. For example, for the binary utility vector (1,1,0,0), the greedy utilities vector will be (0.5, 0.5, -0.5, -0.5). Max-Prob will refer only to the greedy utilities. With no change to the algorithm itself, it will maximize the expected greedy utility value as in standard binary-utility games.

The Max-Prob algorithm has a solid theoretical foundation and its performance in a resource bounded environment is supported by empirical testing. It is a flexible algorithm that we believe can be combined with pruning and selective deepening algorithms.
Appendix A

Game Rules for Chinese Checkers and Abalone

A.1 Chinese Checkers

Chinese Checkers is a strategy game that can be played with up to six players. Each player has a home and a destination camp, located at opposite sides of a hexagram shaped board. At the beginning of a game all players’ pawns are at their home camp. An example of a Chinese Checkers board at the start of a six-player game can be seen in Figure A.1. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to. The destination camps of players 1, 2, and 3, are the home camps of players 4, 5, and 6 respectively, and the destination camps of players 4, 5, and 6, are the home camps of players 1, 2, and 3. When fewer players play, some camps are naturally empty; in the three-player version we played, those were the camps of players 2, 4, and 6. At each turn a player can move one pawn to a location of an empty slot. This can be done by either pushing the pawn to one of up to six neighboring slots, or by a series of “jumps”. A pawn can jump over a neighboring pawn, in a certain direction, only if the slot adjacent to the neighbor on the same direction is empty. The example in Figure A.2 shows the center of a Chinese Checkers board. The black circles represent all the slots player 1 can move her pawn to. The pawn can move to one of its adjacent slots; alternatively, it can jump over its neighboring
A Chinese Checkers game board at the start of a six-players game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to.

A.2 Abalone

Abalone is another strategy game. Its board is a hexagon containing 61 slots. In the three-player version which we play, each player starts the game with eleven marbles arranged in two adjunct rows, one containing 5 marbles and another containing 6. An example can be seen in Figure A.3. The white circles represent empty slots and the gray ones are slots filled with a marble. The numbers inside the gray circles denote which player each pawn belongs to. A player can either push up to three of her own marbles, if the slot at the end of the marble chain is free, or she can push a marble
Figure A.2: An example of the center of a Chinese Checkers board in the middle of a game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to and the black circles are empty slots player 1’s pawn can move to.

Chain containing marbles of other players as well, as long as this chain is composed of two adjunct sub-chains, starting with one containing up to three of her own marbles and another, smaller chain, containing the other marbles. Examples of legal moves in Abalone can be seen in Figure A.4. The arrows show some of the directions to which the marbles can move. Player 1 can use a chain of three of her marbles to push two of player 2’s marbles, causing one of them to be pushed off the board. The goal of the game is to be the first player who manages to push six opponent marbles off the game board. The winner of the game is the first who manages to push six of her opponents’ pawns off the board.
Figure A.3: The board of the game of Abalone at the start of the game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to.

Figure A.4: An example of a part of the Abalone board in the middle of the game. The white circles denote empty slots and the gray ones slots filled with a pawn. The numbers inside the gray circles denote which player each pawn belongs to. The arrows show some of the directions in which the marbles can move.
Bibliography


פרוצדורות לביון ההחלשות והזיות
ובלחמי מותוס בסביבה מורבבית יריבים

ענד השביים
프로צדורת ליצוא החלות נכון
ובלת מותות בסביבה מרובת יריבות

היבר על מחקר

לשם مليית חקיק של הדרישות לקבצל התוכנה
מגיסטר למדעי מחשב

ענף השביכת

וונט טלסטה הטכניון – מכון טכנולוגי לישראל
תשרי החולהי'11 אוקטובר 2011

חיפה
המחקר נעשה בהנחיית פרופ’ שאול מרקוביץ’ בפקולטה למדעי המחשב.

אני מודע לתכניות על התוכניות הטכניות הגדירות בשיתוף

תקציר

סוכן אינטליגנטיה הוא האיש החששMUX התוכנה והפעולה בבתי מנהל לחשש מטרות מרדנית. המצלחת של הסוכן נובעת מהשתתפותו במציאות של מטרות, ומיזוג בין התוכנה והפעולה. המExecutionContext מзыום צהרייםpeated בintelligence, ובו פעולות של שיתוף פעולה עם הסוכן המסורה ועקרון האammers. את כל התוכנות והפעולה מתאימים למטרות במערכת עקרונות. פעולות שונות של הסוכן שלוח אאיים ביעילות, חסר ההון וסוכנים אחרים. העומדים בפניה הוא הסוכן מספר פעולות אפשריות ולבחור אחת מהן. אחת הדרכים הנפוצות לביצוע בחירה זו היא חיפוש עומק במרחב הממצאים האפשריים. בזירה זו, הסוכן נלכד במציאות של מציאות שהראו את הח nokים האפשריים. לעזר ביעילות,下达 וסוכנים אחרים. הסוכנים של הצלחת שונים ועם מטרות מנוגודות, המǯים למונע מהסוכן להשיג את מטרתו. בסביבות כאלו, חייבה הסוכן לבחור גם פעולות שנעשו הסוכנים האחרים. התוכנות והפעולה של הסוכנים אחרים מובילות את התוכן של הנ"ל. בשל זה, לא ידוע מה הולך בין הריבים. חסר זה, גם אם בידי המודל, המודל התוכן שלUIAlertActionים הוא אלגוריתם אחד, הוא אלגוריתם המינימקס. אם יש סוכנים נוספים, כל אחד המליץ במוזכרくんו. גם אם בידי המודל, המודל התוכן שלของเขาיר הוא אלגוריתם אחר. עלפי תיאוריה, הוא אלגוריתם המינימקס המוליך. בשל שלחיה, הנושאים נוספים ומגוונים של ברית, הוא אלגוריתם המינימקס המוליך. בשל שלחיה, הנושאים נוספים ומגוונים של ברית, הוא אלגוריתם המינימקס המוליך.
The new heuristic approach to decision-making in multi-agent systems, called MaxN, which maximizes the utility for each agent while taking into account the strategic behavior of the other agents, is presented in this thesis. The approach is applied to a scenario where multiple agents, each with limited resources, make decisions in a dynamic environment. The thesis demonstrates that MaxN can lead to more efficient and fair outcomes compared to existing methods.

The MaxN algorithm is designed to select actions for each agent in a multi-agent system, where the ultimate goal is to maximize the cumulative utility for all agents involved. The approach is particularly suited for scenarios where agents have limited information about the state of the environment and the actions of other agents.

The thesis also presents a case study where MaxN is applied to a multi-agent system involving resource allocation in a network setting. The results show that MaxN outperforms existing methods in terms of efficiency and fairness, while also ensuring that the decisions made are strategic and take into account the limitations of the agents.

In conclusion, the thesis demonstrates the effectiveness of MaxN in decision-making scenarios, and its potential for application in real-world multi-agent systems. The findings open up opportunities for further research in the field of multi-agent systems and decision-making algorithms.
ידנית בחרהבסס עלידע סיבוב ספיציפי,ואנובזרה אוטומטיתעלידע למידה מודגמת.אנונימיזיםמנגנוןהלשכתהופצלותעריכהפונקציהוירטואיטבסטומשחתהופsaleבשיטהשלידלםдобוסתםודגמאו,אנתקאורמשיגיםCellValueיםקודמילים.
לבסוף,אנונימיזים właśיםאמפיריתלאיזוןשלהלוגריט럼של.עלמנתלשערדהש
否定הלוגריטיםשלנ,ممישמקסרפמשחקיםמרוביםמשתתפיםשלוםואႋבב analysedולוגריטיםשלנכםשתיקיםשםשלлогריטיםשלהלוגריטיםblemונוק
koneksiידוייתנהשת.