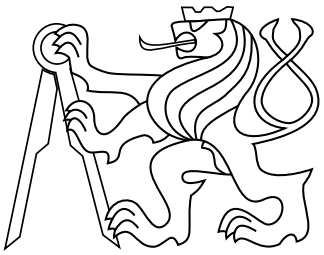
Czech Technical University in Prague
Faculty of Electrical Engineering



Bachelor Thesis

Learning Strategies in Stochastic Zero-Sum Games

David Futschik

Supervisor:
Mgr. Branislav Bošanský, Ph.D.

May 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: David F u t s c h i k

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Učení se strategií ve stochastických hrách s nulovým součtem

Pokyny pro vypracování:

Dvouhráčové nekonečné stochastické hry, ve kterých jsou ohodnocení pouze v terminálních stavech, jsou důležitou třídou her s řadou potenciálních aplikací. Bohužel v současnosti neexistují efektivní a v praxi použitelné algoritmy pro výpočet optimálních strategií pro tuto třídu her. Standardní algoritmy, které využívají iterativní výpočet hodnot/strategií, mohou v nejhorším případě potřebovat až dvojitě exponenciální počet iterací. V oblasti konečných sekvenčních her však v posledních letech vzniklo několik algoritmů, které umožnily řešení velkých her, a které primárně využívají techniky učení pro nalezení optimální strategie. Cílem studenta je proto:
(1) prozkoumat možnosti využití učících algoritmů pro řešení stochastických her a adaptace vybraných přístupů úspěšných v konečných hrách,
(2) experimentálně porovnat nově navržené algoritmy se standardními algoritmy pro řešení nekonečných stochastických modelů na sadě konkrétních her.

Seznam odborné literatury:
[1] Kristoffer Arnsfelt Hansen, Rasmus IbsenJensen, and Peter Bro Miltersen. "The complexity of solving reachability games using value and strategy iteration." Computer Science–Theory and Applications. Springer Berlin Heidelberg, 2011. 7790.
[2] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. "Regret minimization in games with incomplete information." In Advances in neural information processing systems, pp. 17291736. 2007.
[3] Yoav Shoham, and Kevin LeytonBrown. Multiagent systems: Algorithmic, gametheoretic, and logical foundations. Cambridge University Press, 2009.

Vedoucí bakalářské práce: Mgr. Branislav Bošanský, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 1. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: David F u t s c h i k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Learning Strategies in Stochastic Zero-Sum Games

Guidelines:

Twoplayer infinite stochastic games with utilities in the terminal nodes are important class of games corresponding to many realworld problems. However, there are not many practical algorithms for this class of games and typical algorithms based on value (or strategy) iteration can take up to doubly exponential number of iterations to converge. On the other hand, there has been a substantial advancement in algorithms for solving finite sequential games, where algorithms based on learning are among the most successful ones.
The goal of the student is therefore to:

(1) explore learning algorithms applied in finite sequential games and adapt selected algorithms for solving infinite stochastic games,
(2) experimentally compare the computation time of new algorithms with existing value and strategy iteration algorithms on a collection of games.

Bibliography/Sources:

[1] Kristoffer Arnsfelt Hansen, Rasmus IbsenJensen, and Peter Bro Miltersen. "The complexity of solving reachability games using value and strategy iteration." Computer Science–Theory and Applications. Springer Berlin Heidelberg, 2011. 7790.
[2] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. "Regret minimization in games with incomplete information." In Advances in neural information processing systems, pp. 17291736. 2007.
[3] Yoav Shoham, and Kevin LeytonBrown. Multiagent systems: Algorithmic, gametheoretic, and logical foundations. Cambridge University Press, 2009.

Bachelor Project Supervisor: Mgr. Branislav Bošanský, Ph.D.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 14, 2015

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Declaration

I hereby declare that I created the presented thesis independently and that I cited all used sources of information in accord with Methodical instructions about ethical principles for writing academic theses.

V Praze dne ....

Podpis autora práce

Acknowledgements

I would like to take this opportunity to thank my supervisor Mgr. Branislav Bošanský, Ph.D. for his invaluable assistance, patience and exceptional care throughout the writing of this thesis.

Abstrakt

Dvouhráčové nekonečné stochastické hry s nulovým součtem a ohodnocením v koncových stavech jsou důležitým typem her s velkým počtem aplikací, ale jedná se o méně studovanou třídu her. Z tohoto důvodu neexistuje mnoho praktických algoritmů pro jejich řešení. Standardní algoritmy používají iteraci hodnot nebo strategií, ovšem tyto algoritmy mohou v nejhorším případě potřebovat až dvojitě exponenciální počet iterací. Proto hledáme algoritmy s lepší složitostí, nebo alespoň metody vylepšení stávajících algoritmů. První část práce vysvětluje základní pojmy teorie her se zaměřením na řešení her. Následně jsou popsány stochastické hry a standardní postupy při jejich řešení. Abychom mohli aplikovat postupy používané při řešení konečných sekvenčních her, zavádíme pojem serializace stochastických her. Poskytneme algoritmus pro řešení stochastických her založený na kombinaci prvotního odhadu hodnot s hodnotovou iterací. Nakonec provedeme experimentální porovnání nových algoritmů s existujícím algoritmem hodnotové iterace na sadě konkrétních her.

Klíčová slova: Dvouhráčové stochastické hry s nulovým součtem a ohodnocením v koncových stavech, stochastické hry, serializace stochastických her, hodnotová iterace

Abstract

Two-player zero-sum stochastic games with utilities in terminal nodes is an important class of games with many applications, but one that has not been studied in great depth. As such, there are not many practical algorithms for solving this class of games. The two standard algorithms are *value iteration* and *strategy iteration*. However, these algorithms have doubly exponential worst case complexity in number of iterations. Therefore, we are searching for algorithms with lower complexity or methods of improving existing algorithms' performance. First, we explain the most essential basics of game theory with focus on solving games. Then, we describe stochastic games and the standard approaches to solving them. To be able to apply algorithms used in finite sequential games, we introduce the concept of serialization of stochastic games into finite sequential games. We present an algorithm for solving stochastic games based on value estimation combined with value iteration. Lastly, we experimentally compare performance of novel algorithms to the existing value iteration algorithm on a collection of example games.

Keywords: two-player zero-sum stochastic games with utilities in terminal nodes, stochastic games, serialization of stochastic games, value iteration

Contents

Chapter 1

Introduction

In this thesis, we consider two-player zero-sum possibly infinite stochastic games with utilities in terminal nodes. The property of being possibly infinite refers to the individual runs of the game, while the set of game states is finite. This is an important class of games with many applications, but one that has not been studied in great depth.

A direct application of stochastic games is in software design, specifically in network flow control. Stochastic games are used to model dynamic control in queuing networks. Solving the model for the network controller yields an efficient policy which guarantees the best performance under the worst service conditions [1].

Another application is in robot control, where we model the robot (or rather a group of robots) as one of the players, playing against the environment. This is especially effective in environments where the robots only have access to limited means of communication with each other or cannot communicate at all. Stochastic games may be used to represent distributions over other robots' observations about the world, thus minimizing the need for communication. The domain of this application is more complex than the stochastic games studied in this thesis, as they are only partially observable [3].

However, there are not many practical algorithms for solving this class of games. The two standard algorithms are called *value iteration* and *strategy iteration*. The basic idea behind these algorithms is propagating the values from terminal nodes to nonterminal nodes through repeated iteration of values or strategy. However, these algorithms have doubly exponential worst case complexity in number of iterations [5].

Conversely, there has been a significant advancement in algorithms for solving finite sequential games [2]. In this class of games, algorithms based on learning are among the most successful. In this thesis, we explore the possibility of applying learning algorithms to solving stochastic games. Then, we experimentally compare performance of novel algorithms to existing value iteration algorithm on a collection of example games.

Solving stochastic games is significantly harder than solving a sequential finite game for several reasons; firstly, stochastic games are possibly infinite, meaning that an instance of such game is not guaranteed to end. Secondly, nodes of a stochastic game often have circular dependencies.

1.1 Overview

The first part of the thesis is dedicated to introducing the reader to required terminology and theoretical basis. As we are only interested in two-player games, all definitions in this thesis are chosen with that in mind. However, most of them can be (and often are) generalized to n-player games.

Then, we describe the type of games we are interested in and define what *solving a game* means - formally, we want to find a strategy with the best possible average guaranteed outcome provided both players play optimally.

Second part of the thesis focuses on description of an original method of solving the type of games described in the first part. This includes chapters 4 through 6.

The last part of the thesis presents a comparison of the proposed algorithms and the standard method of value iteration. First, the experimental domain is described in Chapter 7. The results and their analysis are then laid out in Chapter 8.

Chapter 2

Theoretical background

In order to properly describe the algorithms and ideas in this thesis, we need to define some crucial terms. We use definitions from [6], most of which have been modified to suit two-player games only. This section describes essentials of game theory and provides necessary theoretical background. First, we define normal-form and extensive-form games, as they are necessary for understanding later algorithms.

Additionally, we need to define what a zero-sum game is, gain basic insight into what a strategy means in different classes of games and present an algorithm to calculate strategy in extensive-form games.

We define best response and Nash equilibrium in order to be able to explain how to solve normal-form games.

2.1 Normal-form game

The key representation of a game in game theory is called Normal-form game (NFG).
Definition: A finite, *2*-person normal-form game is a tuple (N, A, u), where:

- N is a set of two players, indexed by i (and additionally j, where $i \neq j$) $N = \{1, 2\}$.
- $A = A_1 \times A_2$, where A_i is a finite set of actions available to player i. Vector $a = (a_1, a_2) \in A$ is called an *action profile.*
- $u = (u_1, u_2)$ where $u_i : A \mapsto \mathbb{R}$ is a real-valued utility function for player i.

One way (perhaps the most common) to represent normal form games is a matrix. An example of such game would be the *Prisoner's Dilemma* in Figure 2.1.

	Cooperate	Defect
Cooperate	5, 5	0,3
Defect	3, 0	1,1

Figure 2.1: A game of Prisoner's Dilemma

In matrix games the actions of two players correspond to rows and columns of the matrix; the elements of the matrix itself specify the utility of outcomes for respective players. Here, we have two players (row player and column player) who both have the choice of either cooperating with the other player or defecting. Both

players choose their action without having any knowledge about their opponent's choice. After that, they receive utility as specified in the matrix (sometimes also called *Payoff matrix*).

In this thesis, we often refer to a single instance of a normal-form game as a concurrent (move) node (CM).

2.2 Zero-sum game

Definition: A two-player game is *zero-sum* if the following statement holds: for each action (strategy) profile $a \in A_1 \times A_2$ it is the case that $u_1(a) + u_2(a) = 0$.

In other words, a game is zero-sum if first player's gain is exactly equal to the second player's loss.

2.3 Extensive-form game

Normal-form games themselves have no concept of turns, as both players only play once. To introduce sequential play, we must first define extensive-form games.

Definition: A finite, *2*-person extensive-form game (EFG) is a tuple $(N, A, H, Z, \chi, \rho, \sigma, u)$, where:

- N is a set of two players, indexed by i (and additionally j, where $i \neq j$) $N = \{1, 2\}$.
- A is a single set of actions.
- H is a set of nonterminal choice nodes.
- Z is a set of terminal nodes. $(Z \cap H = \emptyset)$
- $\chi : H \mapsto 2^A$ is the action function, which assigns to each choice node a set of possible actions.
- $\rho : H \mapsto N$ is the player function, which assigns to each nonterminal node a player who chooses an action at that node.
- $\sigma : H \times A \mapsto H \cup Z$ is the successor function, which maps a choice node and an action to a new choice node or terminal node such that $\forall h_1, h_2 \in H$ and $\forall a_1, a_2 \in A$, if $\sigma(h_1, a_1) = \sigma(h_2, a_2)$ then $h_1 = h_2$ and $a_1 = a_2$.
- $u = (u_1, u_2)$ where $u_i : Z \mapsto \mathbb{R}$ is a real-valued utility function for player i and $u_1 = -u_2$.

Unlike normal-form game, extensive-form game is best visualized using a tree graph, which inherently encompasses the sequentiality as introduced by σ. See Figure 2.2 for an example with players A and B. In this game, player A is the first player; his choice of action decides the node player B gets to play after that.

Extensive-form games as defined above are often also called perfect information extensive-form games.

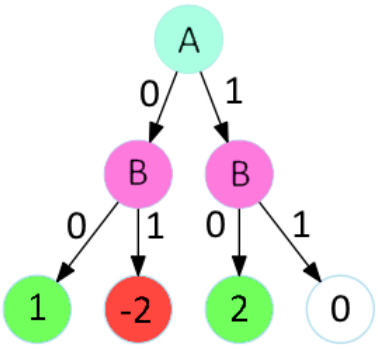


Figure 2.2: An extensive-form game

2.4 Strategy and strategy profile

Definition: Let (N, A, u) be a normal-form game, and for any set X let $\Pi(X)$ be the set of all probability distributions over X. Then the set of *(mixed-)strategies* for player i is $S_i = \Pi(A)$.
Definition: A (mixed-)strategy profile is the Cartesian product of individual strategy sets, $S_i \times S_j$.
Definition: A pure strategy is such strategy that assigns one of the possible actions probability of 1.

Consider again the game in Figure 2.1. Always playing Cooperate would be a pure strategy. Playing Cooperate and Defect with equal probabilities is a mixed strategy.

For extensive-form games, the definition is different, because it may be beneficial for the player to choose a different action at different nodes.
Definition: Let $G = (N, A, H, Z, \chi, \rho, \sigma, u)$ be an extensive-form game. Then the pure strategies of player i consists of the Cartesian product

$$\prod_{h \in H, \rho(h)=i} \chi(h)$$

that is, a pure strategy is a vector which assigns an action to each node where that player is supposed to choose an action.

A mixed strategy for player i is a probability distribution over his set of pure strategies. For example, in Figure 2.2 player A has two possible pure strategies; $\{(0), (1)\}$. On the other hand, player B has four - $\{(0,0), (0,1), (1,0), (1,1)\}$.

For extensive-form games, we can also define the concept of a *behavioral strategy*. Rather than randomizing over pure strategies, a behavioral strategy is a strategy in which the player randomizes in each node independently. For the game in Figure 2.2, $(0.3 \cdot (0,0), 0.7 \cdot (0,1))$ is one example of a mixed strategy for player B, whereas playing action 0 with probability 0.1 and action 1 with probability 0.9 is an example of a behavioral strategy.

2.5 Solving a game

When we talk about solving a game, it means we are searching for a strategy (usually for one player - the maximizer) that gives the player certain guarantees. Consider

a game in which we have to choose a strategy before we are assigned an opponent and before we actually play. We cannot make any assumptions about how the other player is going to play, because if it turned out to be incorrect, we would likely end up with an unfavorable result.

Therefore, we want a strategy that will have the property of being "good" regardless of how our opponent would play. That is, it will guarantee us a minimum utility we can expect to gain and, preferably, this value will be the maximum of all such strategies. That is, we are searching for the *maximin strategy* $\text{argmax}_{s_i} \min_{s_j} u_i(s_i, s_j)$.

The mechanism of finding the *maximin* strategy is different in NFG and EFG, therefore, we describe them separately.

2.6 Best response and Nash equilibrium

Definition: Player i's best response to the strategy s_j is a mixed strategy $s_i^* \in S_i$ such that $u_i(s_i^*, s_j) \geq u_i(s_i, s_j)$ for all strategies $s_i \in S_i$.

Given a strategy of his opponent, a player's best response is such strategy that is not worse than any other strategy available to that player.

Definition: A strategy profile $s = (s_i, s_j)$ is a *Nash equilibrium* if for player i s_i is a best response to s_j and for player j s_j is a best response to s_i.

Nash equilibrium is such couple of strategies that neither player can benefit form changing their strategy. Normal form games do not necessarily have a unique Nash equilibrium, but it can be shown that every finite game has at least one. This is an important result, because, in some sense, Nash equilibrium strategy is the best guarantee a player can have without making assumptions about the other player in games that are zero-sum.

As an example, we can try to find the Nash equilibrium of the previously shown Prisoner's dilemma game. Let us consider only pure strategies. If the row player chooses to cooperate, then the column player's best response would be to defect. But so is the case when the row player defects, therefore, the column player's best response is to always defect. Identical reasoning works for the row player and so defecting is the mutual best response. It is obvious that no player can benefit from assigning cooperation probability higher than zero. Therefore, it is a Nash equilibrium strategy profile.

Nash equilibrium does not necessarily have to exists for pure strategies only, for that reason, we must also consider mixed strategies.In that case, finding Nash equilibria becomes slightly more complicated. In case of zero-sum two-player games, they can be found using linear programming. We use the following formulation:

$$minimize \qquad U_1^* \tag{2.1}$$

$$\sum_{j \in A_2} u_1(a_1^k, a_2^j) s_2^j \leq U_1^* \qquad \forall k \in A_1 \tag{2.2}$$

$$\sum_{j \in A_2} s_2^j = 1 \tag{2.3}$$

$$s_2^j \geq 0 \qquad \forall j \in A_2 \tag{2.4}$$

Solving this linear program yields equilibrium strategy for one of the players as well as the utility for the other player. Similar program can be constructed that reverses the roles of players.

2.7 Algorithms for solving extensive-form games

The standard algorithm for solving perfect-information extensive-form games is called *backward induction*. As the name implies, it involves propagating the tree node values from bottom to top. Since the utilities at terminal nodes are known, we can calculate values of nodes one step higher, by choosing the value most favorable for given player.

We can then effectively replace this node by its value and continue with the calculation, all the way up to the root node. Consider again Figure 2.2. Let us suppose that player B is the *minimizer*, that is, he tries to minimize his utility. By following Algorithm 1 for his two nodes, we get Figure 2.3. After performing another step of backward induction, we arrive at the result of 0 in the root node, meaning that the *value of the game* is equal to 0.

This algorithm also reveals the equilibrium strategy for this game [6]. We can obtain it by keeping track of which action leads to bestUtility (Algorithm 1 lines 14 and 17) and then constructing a pure strategy of this action at given node.

Many improvements to this basic algorithm have been presented, such as $\alpha - \beta$ *pruning*, *Negamax*, *Negascout* and others.

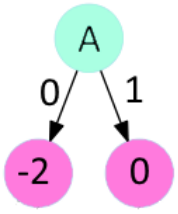


Figure 2.3: Partially solved extensive-form game

2.8 Solving extensive-form games with concurrent moves

CM-EFG is an extensive-form game where one or more nodes have been replaced with a concurrent move node. The basic idea for solving such game is very similar to solving EFGs (Section 2.7). However, when encountered with a concurrent node, calculating its value is not as simple as choosing the value of the best child.

Algorithm 1 Backward Induction

```
1: function BackwardInduction(node)
2:     if node is terminal then
3:         return utility of node
4:     end if
5:     player = ρ(node)
6:     if player is maximizer then
7:         bestUtility = −∞
8:     else
9:         bestUtility = ∞
10:    end if
11:    for all child of node do
12:        childUtility = BackwardInduction(child)
13:        if player is maximizer and childUtility > bestUtility then
14:            bestUtility = childUtility
15:        end if
16:        if player is minimizer and childUtility < bestUtility then
17:            bestUtility = childUtility
18:        end if
19:    end for
20:    return bestUtility
21: end function
```

When solving a normal-form game, we are usually searching for the maximin strategy. Naturally, such strategy is found in a Nash equilibrium of the given game [6]. Several algorithms have been proposed to find Nash equilibria of two-player, zero-sum games, such as solving the linear program shown in section 2.6.

In this case, then, we must calculate the Nash equilibrium strategy and its value. We can then again use that value in place of the node and continue with backward induction.

Chapter 3

Stochastic games

Games we are interested in for this thesis have some special properties. First of all, the games fall into the two-player category. Next, the games we are studying are stochastic, meaning they are, in part, random. An example of this would be a game of tic-tac-toe where the starting player is decided by a coin toss.

A very important property of the games is that they are (possibly) infinite. That is, there exists such vector of strategy profiles $\hat{s} = (s^1, \dots s^q)$ (a profile for each stage game in the studied game) that will ensure the game will never reach a terminal state.

3.1 Definition, example

Definition: A stochastic game is a tuple (Q, N, A, P, Z, u), where:

- Q is a finite set of stage games. These games may correspond to matrix games, be chance nodes or player nodes.
- N is a set of two players, indexed by i (and additionally j, where $i \neq j$) $N = \{1, 2\}$.
- $A = A_1^1 \times \ldots \times A_1^q \times A_2^1 \times \ldots \times A_2^q$, where A_i^k is a finite set of actions available to player i in game $k \in Q$.
- Z is a set of terminal nodes. $Z \neq \emptyset$.
- u is a terminal node utility function u: $Z \mapsto \mathbb{R}$.
- P: $Q \times A \times (Q \cup Z) \mapsto [0, 1]$ is the transition function. $P(q, a, \hat{q})$ is the probability of transitioning from q to $\hat{q}$ after action profile a.

These games therefore contain four basic elements: Matrix games (NFG), player nodes, where one of the players chooses an action, chance nodes, where no player chooses an action but instead the outcome is decided randomly (based on probabilities) and terminal nodes.

Instead of receiving utility after chosing actions, the players are taken to play another stage game based on the chosen actions or an element of randomness in case of chance nodes. This in practice means that the games may be infinite and will only end if the state of the game moves into a terminal node, where each of the players receives utility.

Furthermore, the games could be classified as recursive games with randomness. That is, they are finite sets of "game elements" ($Q \cup Z$), which are either real numbers (utility in terminal nodes) or another game of the set, but not both. It can be shown that if every game element posseses a solution, then the recursive game posseses a solution as well [4].

Another way to look at the games is from a reachability perspective. One player's goal is to reach a certain set of terminal states and the other player's goal is to prevent that from happening (so called safety condition). Additionally, the other player may too have a set of terminal states that they are trying to reach.

Every game has at least one terminal node. At each terminal node player utility is defined (as the games are also zero-sum, utility of player j is equal to minus utility of player i, let us call player i the maximizer and player j the minimizer). Usually, we look at those terminal nodes from perspective of the maximizer. A terminal node that is favorable to that player (utility is greater than zero) is often called a *goal node*, while a terminal node with utility less than zero is called a *trap*.

As an example, consider Figure 3.1, a game called *Dante's purgatory* [5]. Here, player 1's goal is to reach state 3. Player 2 is trying to prevent that while also striving to reach state 4. States 1 and 2 are concurrent nodes, meaning that both players have to choose their action before the next state can be determined. The numbers in the matrices correspond to the result of function P for given combinations of actions.

At state 1 and 2, both players have two actions available, 0 and 1. Note that this game lacks an element of randomness. The values in terminal nodes 3 and 4 are player utilities.

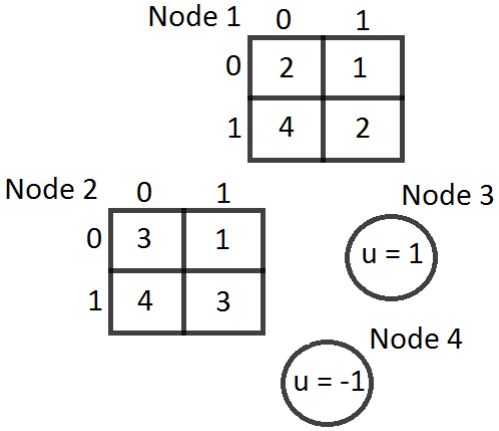


Figure 3.1: Dante's purgatory

3.2 Computing strategy

Computing the values (and thus the strategy) of nodes in a stochastic game is significantly harder than solving normal-form games or extensive-form games, despite having relatively similar structure to EFGs. Backward induction as described in Algorithm 1 will not work in the general case, because the algorithm would possibly never terminate.

If we assume knowledge of node values for all nodes in a game except the one we are trying to calculate, we can calculate the value according to Algorithm 3. In one-player nodes, the player simply selects the successor with the most beneficial value. In concurrent play nodes, we calculate the Nash equilibrium value. For chance nodes, the value is the sum of successor values multiplied by their respective probabilities.

However, because the calculations of values can be dependent on each other, we cannot simply propagate the values as we do in backward induction. Instead, we are forced to iterate the process, thus improving the values in each iteration. There are two best-known algorithms for solving stochastic games. Perhaps the most standard algorithm for doing so is called *value iteration*, described in Algorithm 2. The SolveNode function is described in Algorithm 3.

The iteration needs an initial point to start from (line 2), in most cases we choose to start from 0 for all nonterminal states, but any value would work. Indeed, having a good estimate value before starting to iterate can speed up the process substantially (as we show in Chapter 8).

In each iteration of the algorithm, value of every node is recalculated using vector of values from the previous iteration (line 6). Thus, we only need to store k real numbers, where k is the number of nodes in the game.

Algorithm 2 Value iteration

```
1: S = Set of all nodes
2: values_0 = Initial values
3: i = 0
4: while values_i ≠ values_{i−1} do
5:     for s ∈ S do
6:         values_i[ s ] = SolveNode(s, values_{i−1})
7:     end for
8:     i = i + 1
9: end while
```

The other algorithm is called *strategy iteration*, described in Algorithm 4. In strategy iteration, we iterate *strategically*, meaning we are improving the players' strategy in each iteration, as opposed to only calculating the value of the nodes. $\mu(s_1, s_2)$ on line 7 denotes the probability of player 1 reaching goal, given the players play the strategy profile (s_1, s_2). Maximin strategy (needed on line 14) is the Nash equilibrium strategy for player 1.

In both algorithms, we iterate until the convergence is reached. In practice however, this may be a very lengthy process; usually, we do not iterate until the values are stable, but rather until they only change by a very small amount (ϵ>0). Both algorithms also have the property that as i approaches infinity, the values approach the correct values.

In fact, the worst case complexity of the algorithms is doubly-exponential. This phenomenon can be observed on Dante's purgatory[5]. Therefore, we are searching for algorithms with lower complexity, or at least with better performance in the general case.

Algorithm 3 Calculating node values

```
1: function SolveNode(node, values)
2:     if node is a terminal node then
3:         return utility of node
4:     else if node is a chance node then
5:         value = 0
6:         for all children of node do
7:             value = value + values[child] × probability of child
8:         end for
9:         return value
10:    else if node is a concurrent play node then
11:        matrix = game matrix of node with values filled in
12:        value = NashEquilibriumValue(matrix)
13:        return value
14:    else if node is a maximizer node then
15:        value = -∞
16:        for all children of node do
17:            if values[child] > value then
18:                value = values[child]
19:            end if
20:        end for
21:        return value
22:    else if node is a minimizer node then
23:        value = ∞
24:        for all children of node do
25:            if values[child] < value then
26:                value = values[child]
27:            end if
28:        end for
29:        return value
30:    end if
31: end function
```

Algorithm 4 Strategy iteration

N = Set of all nodes
s_1^0 = strategy for player 1 playing uniformly at each node
i = 0
while true **do**
    s_2^i = optimal best reply by player 2 to s_1^i
    for $n \in N$ **do**
        $values_i[\, n \,] = \mu(s_1^i, s_2^i)$
    end for
    $i = i + 1$
    for concurrent node $n \in$ N **do**
        matrix = game matrix of node with values filled in
        value = NashEquilibriumValue(matrix)
        if value $> values_{i-1}[\, n \,]$ **then**
            $s_1^i[\, s \,]$ = maximin(matrix)
        end if
    end for
end while

Chapter 4

Serializing stochastic games

In order be able to apply algorithms used on sequential games, we have to make the stochastic game finite. One such way is to "serialize" the game.

Serialization of stochastic games is an idea of approximating possibly infinite games with finite games, namely extensive-form games with concurrent moves. This way we obtain a game that we already have theoretical foundations for solving, by using estabilished algorithms and methods described in Chapter 2.

4.1 Serialization

One algorithm to serialize a stochastic game is the Algorithm 5. In order to serialize a stochastic game, we first have to choose the node that will be the root node of the finite approximation. Any nonterminal node can be chosen as the root of the serialization. When using this algorithm, we specify the *depth* of serialization. The algorithm expands the game, starting from the root node. When a node with successors is encountered, all its successors are copied (line 20) and expanded as well (line 12). This process stops when nodes in path from the root have been used *depth* number of times (line 17).

4.2 Description of the algorithm

The algorithm starts by copying the node selected as root of the serialization and putting the copy on in the work queue (lines 4 and 5). Then, until the queue is empty, we take the front of the queue and, if it is not a terminal node, copy its successors and set the copied nodes as its new successors (CopyAndSetAllSuccessors function).

However, if any of the successor nodes have been copied *depth* times already, we instead replace it with a terminal node containing the best known value of that node as its utility (line 16). All copied nodes are put into the work queue after they have been assigned as successors to their parent (line 9). The resulting game is then effectively stored in the root node, as its graph is a tree.

Consider an example game as seen in Figure 4.1. Nodes 1 and 2 could be concurrent move nodes or one-player nodes (their exact type is not relevant for the serialiaziation, only the fact that they have successors is), nodes 3 and 4 are terminal nodes.

Algorithm 5 Serialization of a stochastic game

```
1: depth = User parameter of serialization depth
2: numUsed = Mapping of how many times each node has been used in serialization
3: workQueue = Queue of nodes to process
4: root = CopyNode(root)
5: workQueue.Add(root)
6: while workQueue is not empty do
7:     currentNode = workQueue.RemoveFront()
8:     if currentNode is a terminal node then
9:         continue
10:    end if
11:    successors = CopyAndSetAllSuccessors(currentNode)
12:    workQueue.AddAll(successors)
13: end while
14: function CopyAndSetAllSuccessors(node)
15:    newNodes = empty list
16:    for all successor of node do
17:        if numUsed[successor] ≥ depth then
18:            ReplaceSuccessor(node, successor, MakeTerm(successor))
19:        else
20:            copy = CopyNode(successor)
21:            ReplaceSuccessor(node, successor, copy)
22:            newNodes.Add(copy)
23:        end if
24:        return newNodes
25:    end for
26: end function
```

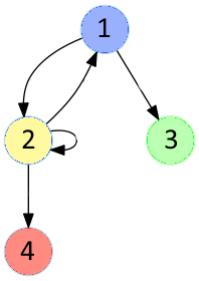


Figure 4.1: Example game

Serialized game created by this algorithm is a tree graph game. Nodes where the graph would naturally continue but the node has been used maximum of allowed times are replaced by the best known value for that node as a new terminal node. See the game from Figure 4.1 serialized into a tree with depth = 3 in Figure 4.2. The white node represents the best known value for nodes 1 and 2.

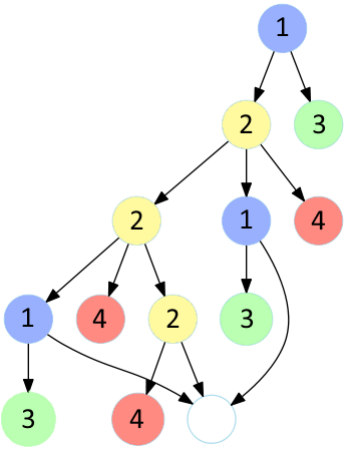


Figure 4.2: Example game serialized to a tree

To calculate value of nodes in the serialized game, we can use the backward induction algorithm (optionally with $\alpha - \beta$ pruning) described in Algorithm 6. Compared to Algorithm 1, this implementation also supports concurrent play nodes and chance node. As the serialized game contains no simple cycles, the algorithm is guaranteed to finish.

4.3 Improvements

However, a closer inspection reveals that many subtrees are often repeated in the serialized game. This could potentially cause the game to substantially increase in size. Naturally, we should try to avoid duplicity where possible, thus, instead of producing a tree graph, we should try to create a directed acyclic graph (DAG), as its properties are identical to a tree as far as calculating the game value is concerned and it does not suffer from the issue of duplicate subtrees.

The algorithm for serializing a stochastic game into a DAG is very similar to the tree serialization, except this time, we keep a list of nodes that have not yet been

Algorithm 6 Backward induction for serialized games

```
1: procedure BackwardInduction(node)
2:     value = 0
3:     if node is terminal then
4:         value = node utility
5:     else if node is a chance node then
6:         for successor of node do
7:             p = probability of successor
8:             value += p × BackwardInduction(successor)
9:         end for
10:    else if node a maximizer node then
11:        max = −∞
12:        for successor of node do
13:            successorValue = BackwardInduction(successor)
14:            if successorValue > max then
15:                max = successorValue
16:            end if
17:        end for
18:        value = max
19:    else if node a minimizer node then
20:        min = ∞
21:        for successor of node do
22:            successorValue = BackwardInduction(successor)
23:            if successorValue < min then
24:                min = successorValue
25:            end if
26:        end for
27:        value = min
28:    else if node is a concurrent play node then
29:        m = matrix of BackwardInduction values of successors
30:        value = NashEquilibriumValue(m)
31:    end if
32: return value
33: end procedure
```

expanded (taken out of the workQueue, algorithm line 12). Whenever we would copy a node in that list, we use that node instead. See Figure 4.3 for the earlier example game serialized to a directed acyclic graph game using Algorithm 7. As we can see, choosing DAG instead of tree saved us an instance of a 1 node (and therefore all subtrees it would spawn).

Algorithm 7 SerializeGameDag

```
depth = User parameter of serialization depth
numUsed = Mapping of how many times each node has been used in serialization
workQueue = Queue of nodes to process
openList = Empty list
root = CopyNode(root)
workQueue.Add(root)
while workQueue is not empty do
    currentNode = workQueue.RemoveFront()
    if currentNode is a terminal node then
        continue
    end if
    openList.Remove(currentNode)
    successors = CopyAndSetSuccessors(currentNode, openList)
    workQueue.AddAll(successors)
end while
function CopyAndSetSuccessors(node, openList)
    newNodes = empty list
    for all successor of node do
        if numUsed[successor] ≥ depth then
            ReplaceSuccessor(node, successor, MakeTerm(successor))
        else
            if successor not in openList then
                copy = CopyNode(successor)
                ReplaceSuccessor(node, successor, copy)
                newNodes.Add(copy)
            end if
        end if
        return newNodes
    end for
end function
```

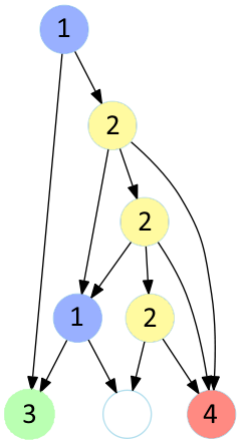


Figure 4.3: Example game serialized to a directed acyclic graph

To calculate the value of nodes, we can again use the backward induction algorithm described in Algorithm 6. When we get the value of the root node, it can be then used as a new best known value for that node in the original game.

4.4 Serialization iteration

Once we can serialize the game from any nonterminal node, we can iterate this process. The complete algorithm serializes the game, using every nonterminal node as a root node, improving the best values as it iterates. See Algorithm 8.

The algorithm's outer loop works exactly like value iteration, but the means of obtaining the value differs. The SerializeGameDag function serializes the game according to Algorithm 7, using s as its root, *depth* as the maximum times any node can be used and $value_i$ as the currently best known values of nodes ($value_i[k]$ for k-th node) for when it has to replace a node by its value.

Algorithm 8 Serialization Value Algorithm

S = Set of nonterminal nodes
$value_0$ = Vector of zeros
depth = Depth constant chosen by user
while $max(|value_i - value_{i-1}|) > \epsilon$ **do**
    for *node* $s \in S$ **do**
        g = SerializeGameDag(s, depth, $value_i$)
        $value_i[s]$ = DFS(root of g)
    end for
end while

Chapter 5

Serialization of concurrent play

Computing the Nash equilibria of matrix (concurrent) games is a costly operation, because it requires us to solve a linear program, which consists of constructing said program and then usually calling an external solver. Naturally, we should think whether there exists a way that would allow us to quickly approximate the value instead. One such approach is to *serialize the concurrent play*[2]. That is, to approximate simulataneous play with a turn based game, effectively producing a perfect-information extensive-form game.

5.1 Serializing a single concurrent stage

There are two different ways to serialize a concurrent play move in a two-player game. One is to allow the first player to play first, thus giving the second player an advantage of knowing his opponent's move before choosing. The other way is where the roles are reversed.

Let us asses the situation from one player's perspective in both cases. When the first player goes first, it is at a clear disadvatage; they will expose their strategy before the other player gets to play. But with this in mind, the player can make such decision that has the ability to prevent the worst of outcomes, essentially creating the best worst-case scenario. By calculating its value, we get the lower-bound value of the original game.

On the other hand, if the player gets to play second, they already know the action taken by the other player and can therefore choose their action accordingly. This produces the upper-bound value of the original game.

5.1.1 Example

Consider Figure 5.1 that shows an example concurrent play, zero-sum game, where A is the maximizer and B is the minimizer. By serializing it, we obtain two games, Figure 5.2 shows the lower bound game for player A, Figure 5.3 shows the upper bound game for player A. By solving those games, we can observe that the lower bound value of the original game is 0, while the upper bound value is 1, it therefore follows, that the actual value of a Nash equilibrium must be between 0 and 1 (it is, in fact, 0).

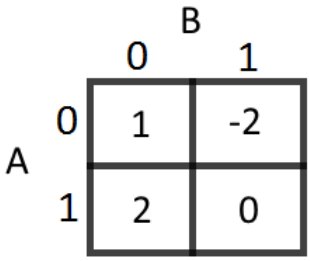


Figure 5.1: Example zero-sum game

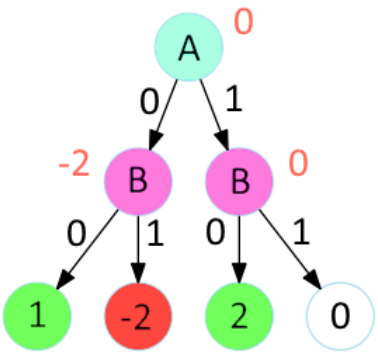


Figure 5.2: Lower bound game serialization

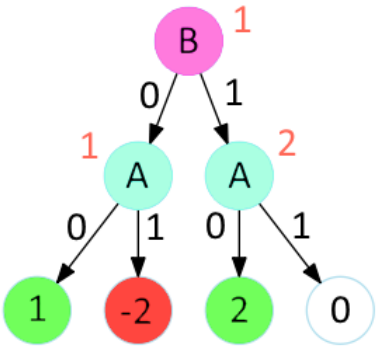


Figure 5.3: Upper bound game serialization

5.2 Notes

Note that when the lower bound value is equal to the upper bound value, then they are also directly equal to the value of a Nash equilibrium of the original game. Since the serialization and calculation of its values is a significantly simplier operation, it allows us to quickly check whether this is possibly the case and thus obtaining the value of a Nash equilibrium almost effortlessly.

Chapter 6

Application of serialization

By applying the concepts described in Chapters 4 and 5, we were able to design an algorithm for solving stochastic games.

The algorithm works in two phases; in the first phase, the algorithm tries to rapidly estimate the values of nodes using serialization of stochastic games and serialization of concurrent play. The second phase refines the values, converging towards the solution using value iteration.

6.1 First phase

The first phase takes advatage of serializing the game, as described in Chapter 4 as well as serializing concurrent play, as described in Chapter 5. In each iteration, a sequence of steps is carried out for each nonterminal node. First, the node is used as the root in a serialization into a DAG game. All concurrent play nodes in this serialized game are then serialized into extensive-form lower and upper bound games. Afterwards, the lower and upper bounds are calculated (using backward induction described in Algorithm 6).

Once we know the lower and upper bound values for the given state, those values are averaged (line 8) and this average is used as the new value of the node. Based on experimental data, this works reasonably well, often producing the correct values (as we will show in Chapter 8). See Algorithm 9 for implementation.

The algorithm iterates until a user-chosen codition is met. For example, this could be a requirement for convergence, although convergence of values in this algorithm does not necessarily bring meaningful information, as the algorithm itself does not converge to the correct values. In practice, this algorithm is best run for either a fixed number of iterations or time, simply because there are no guarantees that the estimate gets better with more iterations.

The main reason we use this algorithm is that it is very quick. It gives us a value estimate for each node, with no guarantee of how close to solution that estimate is (when lowerBound and upperBound are equal, then the value is correct). It is also the reason we should want to choose a rather small depth parameter (2 or 3 in practice), as increasing the depth has a negative effect on the computational time of each iteration due to increasing size of serialized DAG.

Algorithm 9 First phase

```
1: S = Set of all nonterminal states
2: depth = Depth of serialization
3: bestValues = Mapping of best known values of states
4: repeat
5:     for node s ∈ S do
6:         g = SerializeGameDag(s, depth, bestValues)
7:         lowerBound, upperBound = SerializeConcurrentPlay(root of g)
8:         value = (lowerBound + upperBound) × 0.5
9:         bestValues[ s ] = value
10:    end for
11: until arbitrary condition is met
```

6.2 Second phase

After obtaining an estimate of the values from the first phase, the second phase is present to improve the values with guaranteed convergence. It does so by using value interation (Algorithm 2), where the initial values have been set to the output of the first phase. This phase continues until ϵ-convergence is reached, meaning none of the node values changed by more than $\epsilon > 0$ from the previous iteration.

6.3 Both phases combined

The resulting algorithm is running phase one and phase two in sequence, using values estimated by the first phase as the initial point for the second phase. See Algorithm 10. As the second phase has the same guarantees regarding convergence as value iteration, the node values approach the correct values as the number of iterations in second phase approaches infinity. However, this also means that the worst case complexity is the same - doubly exponential.

Algorithm 10 Resulting algorithm

```
1: game = Game to solve
2: depth = Depth of serialization
3: value estimates = First phase(game, depth)
4: values = Value iteration(game, value estimates)
```

Chapter 7

Experimental domain

Because there is no public library of stochastic games, we had to generate a set of games to experiment on. All experiments described in the next chapter have been carried out on a set of games with the following two properties: randomly generated and computationally interesting.

Every game in the set has the property of being *non-trivial*. This property has been tested by running value iteration for 3 seconds. If the algorithm did not ϵ-converge in that time (see Chapter 8 for specific values), the game is considered computationally interesting. Some of the generated games also have a predetermined number of given node types (e.g. chance nodes). This does not apply to all games in the set however, as some of the games have the numbers generated randomly (up to a fixed maximum).

All the games in the set have been generated by the algorithm described in Algorithm 11, which parametrizes every game by 8 values: a seed, number of terminal nodes, number of goal nodes, number of chance nodes, number of concurrent nodes, number of maximizer nodes, number of minimizer nodes and the number of actions available at each node. The algorithm first creates the nodes and then generates transitions between them in a pseudo-random fashion.

The experiments in Chapter 8 were done on four types of games:

- Small games - 5 chance nodes, 7 concurrent play nodes, 2 actions available at nodes, 1 single player node for each player.
- Medium games - 8 chance nodes, 15 concurrent play nodes, 2 actions available at nodes, 3 player nodes for the maximizer, 2 player nodes for the minimizer.
- Large games - 10 chance nodes, 20 concurrent play nodes, 3 actions available at nodes, 5 player nodes for each of the players.
- Random games with a maximum of 60 nodes in total and a maximum of 5 actions available at each node.

Even the large games are relatively small in number of nodes, but the difference is sufficient to demonstrate differences between the tested algorithms.

Algorithm 11 Game generator

```
goal = terminal node with utility 1
trap = terminal node with utility -1
chanceNodes = list of chance nodes
concurrentNodes = list of concurrent nodes
player1Nodes = list of player nodes
player2Nodes = list of player nodes
allNodes = goal ∪ trap ∪ chanceNodes ∪ concurrentNodes∪
                player1Nodes ∪ player2Nodes
for all chanceNode ∈ chanceNodes do
    successorCount = RandomNumber(low, high)
    distribution = RandomDistribution(successorCount)
    connections = SelectRandomNodes(sucessorCount, allNodes)
    chanceNode.probabilities = distribution
    chanceNode.successors = connections
end for
for all concurrentNode ∈ concurrentNodes do
    for i = 1 .. numActions do
        connections = SelectRandomNodes(numActions, allNodes)
        concurrentNodes.successors[i] = connections
    end for
end for
for all player1Node ∈ player1Nodes do
    player1Node.successors = SelectRandomNodes(numActions, allNodes)
end for
for all player2Node ∈ player2Nodes do
    player2Node.successors = SelectRandomNodes(numActions, allNodes)
end for
```

Chapter 8

Experimental results

Three algorithms were selected for comparison. From the standard algorithms, value iteration (VI) was chosen. Second algorithm was the one described in Chapter 6, which we will call value iteration with estimation (EST+VI) in this chapter. The last algorithm compared is serialization iteration, as described in Algorithm 8, also with value estimation (EST+SER).

The algorithms were allowed to run for up to 600 seconds before being stopped. The ϵ for double precision comparison used was set to 10^{-15}. All references to algorithm convergence in this chapter are with respect to this ϵ value. All the compared algorithms were implemented in Java using a single core. Experiments were done on a machine equipped with Intel i5-3210M @ 2.5 GHz CPU and 16 GB of RAM. Linear programs were solved using the IBM ILOG CPLEX v12.4 library.

Though the convergence of EST+SER is not proven or otherwise guaranteed, the values it produces have been experimentally used as the initial point in value iteration as a weak check of convergence. In all cases where EST+SER converged, using the values produced as the initial point for VI resulted in convergence after one iteration.

In total, 112 games were selected for experimental comparison. We selected 24 small games, 28 medium games, 32 large games and 28 random games. The configuration of EST and SER was set to depth = 2. In both EST+VI and EST+SER, the first phase was allowed to run for up to $\frac{1}{30}$ of the time - 20 seconds. The reason we chose this figure is twofold: On one hand, there is no guarantee that the EST phase is doing calculations that are actually helpful in determining the correct values, therefore letting the EST phase have a greater fraction of the total computing time is a risk (as it would take time away from the phase two algorithms). On the other hand, we did want to demonstrate that it can very powerful, and this figure seemed appropriate based on experimental data.

8.1 Measured data

Because the concept of an iteration is very different for the EST+SER and VI algorithms and therefore it would not make sense to compare them using the number of iterations it took for the algorithms converge, we compare their performance based on their runtime in seconds.

Out of the 112 games, VI converged in 15 cases. In those 15 cases, 3 instances were small games, 3 medium games, 6 large games and 3 random games. EST+VI

converged in 90 cases: 23 out of 24 small games, 24 out of 28 medium games, 21 out of 32 large games, 22 out of 28 random games. Lastly, EST+SER converged in 98 cases; 24 out of 24 small games, 26 out of 28 medium games, 23 out of 32 large games and 25 out of 28 random games.

Both EST+VI and EST+SER therefore show a substantial improvement in performance over VI on the experimental set of games, but the effect does seem to diminish as the games get larger. This fact could be caused by poor scaling of the EST algorithm, although more data would be necessary to confirm such hypothesis.

Tables 8.1, 8.2, 8.3 and 8.4 show the time in seconds needed for the algorithms to converge on given game instances. If an algorithm did not converge within the 600 seconds' time, the respective element in the table is left blank. The seeds listed uniquely identify a game of given type. They are the same seeds we used to generate those games.

It is interesting to note that EST+SER converged on several games where EST+VI did not (e.g. small game seed 77). Such phenomenon suggests that EST+SER exploits some properties of the games that EST+VI can not. Further research on this algorithm could explain the behavior.

Tables 8.5, 8.6, 8.7, 8.8 show the number of iteration until convergence for the VI and EST+VI algorithms. If the algorithm did not converge, the number of iterations shown is the number of iterations completed by the 600 seconds mark.

Even though iterations of the estimation algorithm and iterations of value iteration are hard to compare, the results show a clear trend: whenever VI solved a game, EST+VI also solved the game, while requiring strictly fewer VI iterations to do so, implying the estimation was closer to the solution than vector of zeros.

8.2 Detailed examples

Let us compare three example games in greater detail for the VI and EST+VI algorithms.

First of the selected examples is one of the large games, seed 155 and the value of its node 3 in number of iterations. See Figure 8.1. The correct value of the node is roughly 0.714. EST+VI algorithm is clearly faster, calculating the value correctly to 3 decimal places within 100 iterations. Similar precision can be reached only after around 850 iterations of VI.

The second example is one of the small games - seed 10, node 2. See Figure 8.2 for plot of precision against number of iteration. In this case, the correct value of the node is roughly equal to 0.478. EST+VI reaches precision of 3 decimal places after 1450 iterations, whereas VI needs more than 16000 iterations to reach similar precision.

In both examples, the trend of convergence is very similar (on a logarithmic scale) for both algorithms. It is obvious that the EST+VI algorithm is superior in these cases. The EST phase provides a speed-up factor of nearly an order of magnitude.

On the other hand, consider large game seed 141, node 3, in Figure 8.3. There, the trends appear to follow the previous examples until about 600 iterations (where the EST phase ends). From there, the performance of EST+VI is the same as that of VI.

Seed	VI	EST+SER	EST+VI
3	—	10.015	8.203
8	—	0.169	0.06
9	—	0.094	0.017
10	42.287	10.355	7.57
12	3.073	1.608	1.168
14	—	0.356	0.225
16	—	0.295	0.186
26	—	0.478	0.113
44	—	20.079	17.174
60	—	0.211	0.134
63	—	0.161	0.084
71	4.976	1.707	1.039
72	—	0.113	0.054
77	—	1.734	—
84	—	0.129	0.065
86	—	0.745	0.554
104	—	0.219	0.132
111	—	0.953	0.619
118	—	0.151	0.101
128	—	0.081	0.053
137	—	1.174	1.017
142	—	275.853	36.921
144	—	0.276	0.272
146	—	0.271	0.267

Table 8.1: Small games time comparison table

Seed	VI	EST+SER	EST+VI
3	—	1.731	2.778
6	—	2.201	1.808
7	—	0.814	0.576
14	—	3.180	2.766
16	—	0.531	0.268
27	—	1.414	0.998
28	—	2.731	2.139
36	13.461	11.998	9.029
37	—	2.976	2.423
46	6.051	0.795	0.545
56	—	0.595	0.338
57	—	7.155	—
59	—	0.266	0.142
64	—	3.829	3.107
65	—	6.385	4.99
68	—	1.058	0.742
75	—	0.665	0.404
76	—	1.048	0.709
85	—	—	—
91	—	0.862	0.687
96	—	0.515	0.357
115	5.481	0.650	0.421
120	—	1.134	0.881
123	—	3.080	2.685
156	—	3.026	—
159	—	16.011	8.598
165	—	—	—
167	—	1.783	1.206

Table 8.2: Medium games time comparison table

As previously stated, it would be misleading to compare the algorithms based only on the number of iterations. In Table 8.3 we can see that EST+VI and VI have similar run times on game 141, despite EST+VI performing fewer iterations in total. Although the EST phase only ran for 605 iterations, VI calculated over 6000 iterations in the same amount of time.

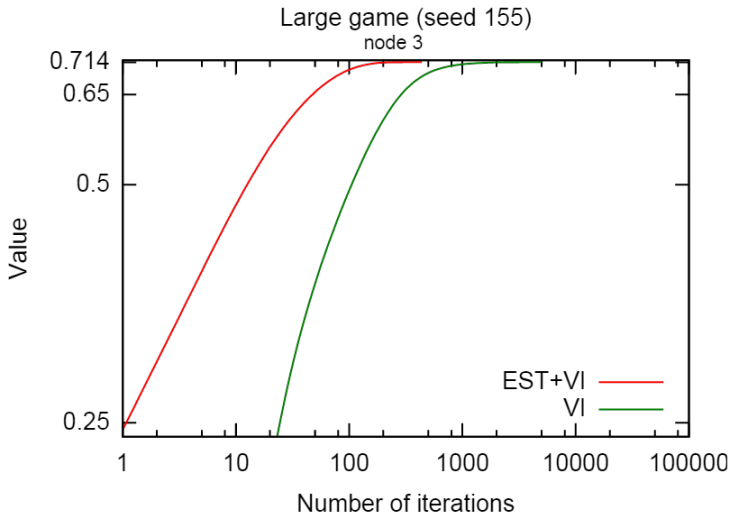


Figure 8.1: Large game (seed 155, node 3)

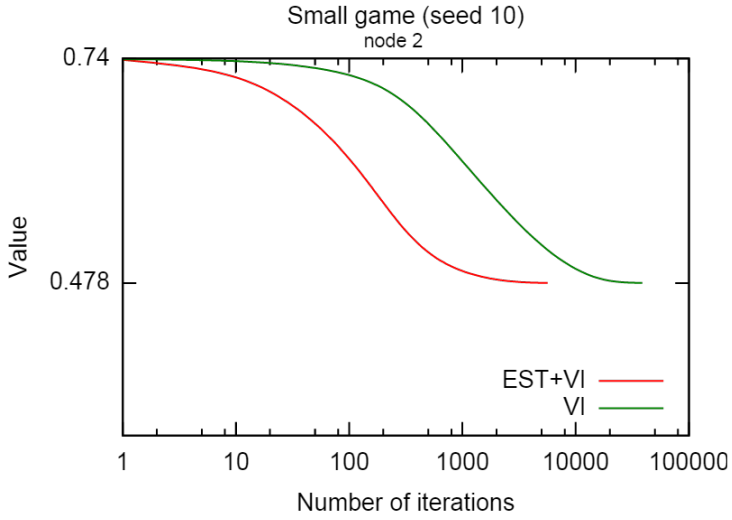


Figure 8.2: Small game (seed 10, node 2)

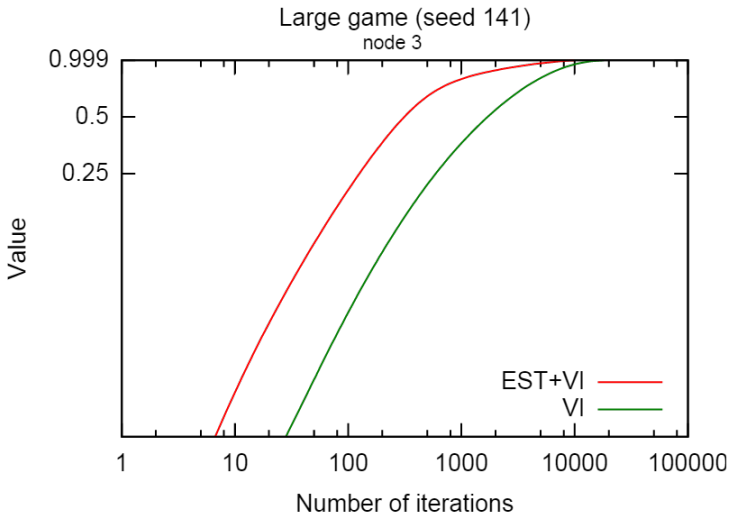


Figure 8.3: Small game (seed 10, node 2)

Seed	VI	EST+SER	EST+VI
127	—	5.531	3.945
130	185.797	—	187.759
131	—	23.389	20.121
133	—	—	—
135	—	—	—
137	—	7.647	—
139	—	—	20.221
141	54.903	414.061	59.519
142	—	71.425	—
143	—	—	—
146	—	6.662	5.322
147	7.385	20.365	20.083
148	—	2.817	2.393
152	—	75.175	—
155	330.073	12.111	11.069
159	30.651	150.821	31.923
161	86.958	1.732	1.256
166	—	5.386	4.205
169	—	20.645	19.68
173	—	—	—
175	—	4.529	4.072
179	—	24.158	20.361
182	—	—	—
184	—	6.214	5.173
192	—	—	—
198	—	15.034	15.120
199	—	24.080	—
205	—	6.865	6.073
208	—	88.337	20.91
215	—	—	—
216	—	21.324	20.072
217	—	10.559	8.677

Table 8.3: Large games time comparison table

Seed	VI	EST+SER	EST+VI
5	—	3.725	—
8	—	0.862	0.716
9	—	—	—
16	—	0.219	0.175
18	—	0.542	0.601
30	9.832	21.685	20.105
32	14.038	70.181	22.733
44	—	2.979	0.719
48	—	1.043	1.011
57	—	0.572	0.236
67	—	—	—
73	—	—	—
74	—	1.358	0.454
78	—	3.041	2.821
79	—	61.171	—
80	—	0.640	0.486
83	—	0.904	0.239
93	—	7.019	4.689
105	—	0.589	—
109	—	5.749	2.114
110	—	1.046	0.340
112	—	0.18	0.121
114	5.189	50.207	9.795
117	—	2.039	0.935
124	—	8.797	4.838
128	—	1.692	1.095
131	—	0.226	0.19
137	—	0.109	0.061

Table 8.4: Random games time comparison table

Seed	VI	EST+VI
3	> 650488	6803 + 77
8	> 644050	52 + 1
9	> 806261	14 + 1
10	51553	5607 + 1
12	4207	831 + 3
14	> 709121	362 + 8
16	> 678501	123 + 5
26	> 671679	50 + 51
44	> 644645	10243 + 1
60	> 568116	84 + 1
63	> 625912	97 + 3
71	6381	1118 + 20
72	> 648902	51 + 1
77	> 655767	> 49 + 605212
84	> 707445	34 + 1
86	> 787160	299 + 2
104	> 633750	103 + 1
111	> 665815	269 + 6
118	> 700366	50 + 1
128	> 876636	31 + 1
137	> 800542	389 + 11
142	> 707474	17271 + 21686
144	> 663424	124 + 3
146	> 650350	168 + 7

Table 8.5: Small games number of iterations comparison table

Seed	VI	EST+VI
3	> 316608	178 + 2
6	> 355229	207 + 2
7	> 248968	90 + 3
14	> 333406	315 + 9
16	> 366947	44 + 2
27	> 287119	136 + 10
28	> 366138	335 + 6
36	7958	1454 + 8
37	> 332865	362 + 3
46	3072	93 + 1
56	> 320462	52 + 2
57	> 298701	> 744 + 369018
59	> 342352	19 + 1
64	> 285442	411 + 2
65	> 308322	652 + 15
68	> 300028	111 + 5
75	> 334268	71 + 3
76	> 372349	117 + 3
85	> 323177	> 3527 + 284228
91	> 323142	78 + 1
96	> 321303	50 + 1
115	344	52 + 4
120	> 333939	112 + 2
123	> 355405	476 + 3
156	> 331153	> 48 + 306785
159	> 305674	1582 + 22
165	> 321810	> 71 + 304857
167	> 406402	176 + 11

Table 8.6: Medium games number of iterations comparison table

Seed	VI	EST+VI
127	> 229328	173 + 8
130	66203	810 + 59812
131	> 215719	880 + 43
133	> 225915	> 846 + 235765
135	> 219263	> 808 + 217487
137	> 207357	> 27 + 217575
139	> 216200	779 + 38
141	19653	817 + 14601
142	> 215502	> 885 + 200730
143	> 190637	> 810 + 182364
146	> 214156	244 + 8
147	2033	798 + 18
148	> 212380	75 + 1
152	> 212109	> 822 + 205265
155	109601	436 + 5
159	9954	745 + 3765
161	28800	54 + 2
166	> 201429	129 + 6
169	> 206229	733 + 13
173	> 187851	> 842 + 194402
175	> 191286	141 + 8
179	> 108629	729 + 128
182	> 219862	> 848 + 188404
184	> 195083	209 + 3
192	> 215529	> 749 + 197653
198	> 213067	614 + 6
199	> 214688	> 775 + 187726
205	> 225195	239 + 6
208	> 227773	866 + 222
215	> 209248	> 618 + 182049
216	> 214791	761 + 15
217	> 212956	321 + 12

Table 8.7: Large games number of iterations comparison table

Seed	VI	EST+VI
5	> 2044740	> 628 + 1906578
8	> 488387	50 + 3
9	> 347622	> 111 + 262020
16	> 558529	18 + 1
18	> 1535545	64 + 1
30	19430	2119 + 142
32	9796	692 + 1592
44	> 385162	23 + 69
48	> 340973	51 + 1
57	> 637780	19 + 1
67	> 256729	> 549 + 234069
73	> 330666	> 1281 + 379210
74	> 1974642	62 + 240
78	> 329530	114 + 1
79	> 337393	> 1031 + 277547
80	> 796677	113 + 1
83	> 1233892	26 + 95
93	> 539530	802 + 43
105	> 4406529	> 47 + 3686992
109	> 376768	70 + 41
110	> 1375031	57 + 112
112	> 486283	21 + 1
114	5058	923 + 3801
117	> 527152	391 + 2
124	> 353641	371 + 2
128	> 519090	173 + 5
131	> 712752	22 + 1
137	> 3823240	12 + 5

Table 8.8: Random games number of iterations comparison table

Chapter 9

Conclusion

Two-player zero-sum stochastic games with utilities in terminal nodes are an important type of games with many applications in other parts of computer science. The goal of this thesis was to design an algorithm for solving two-player zero-sum stochastic games using the ideas applied to solving sequential finite games and to experimentally compare its performance with estabilished algorithms on a set of games.

First, we described the basics of game theory, including solution concepts for normal-form and extensive-form games. Next, we described stochastic games and common algorithms for solving them. Then, we presented an algorithm for serializing stochastic games in order to make them finite and described the process of serializing concurrent play and computing lower and upper bound values of a matrix game.

Because there is no public library of stochastic games, we created an algorithm to generate a diverse set of stochastic games to experiment on. By combining serialization of stochastic games and serialization of concurrent play with value iteration, we designed a two-phase algorithm for computing the values of stochastic games. While the algorithm presented in Chapter 6 has the same worst case complexity as value iteration - doubly exponential, the value estimation phase can greatly improve the performance in practice, as seen on the experimental data in Chapter 8.

Three algorithms were experimentally compared on a set of 112 games. Measured data shows promising results of value estimation, in many cases significantly reducing the time needed by value iteration to reach ϵ-convergence. Even though the estimation has no theoretical guarantees, it seems to be very powerful. The experimental data also suggests that the serialization iteration algorithm could be a viable alternative to value iteration, but no proof of its convergence exists. However, more research regarding scaling of both algorithms on even larger games needs to be done before the algorithms could be used in applied research or real world applications.

Bibliography

[1] Flow control using the theory of zero sum markov games. *Queueing Systems*, 23, 1996.

[2] Branislav Bošanský. *Iterative Algorithms for Solving Finite Sequential Zero-Sum Games*. PhD thesis, Czech Technical University in Prague.

[3] Rosemary Emery-Montemerlo, Geoff Gordon, and Jeff Schneider. Game theoretic control for robot teams. *Robotics and Automation*, 2005.

[4] H. Everett. Recursive games. *Annals of Mathematics Studies*, 39, 1957.

[5] Kristoffer Arnsfelt Hansen, Rasmus Ibsen-Jensen, and Peter Bro Miltersen. The complexity of solving reachability games using value and strategy iteration. 2010.

[6] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems*. Cambridge University Press, 2009.

Appendix A

Contents of the attached CD

The attached CD contains source code files of our framework, including Java implementations of the algorithms described in this thesis. It also contains all files with experimental data, along with a runnable jar file which can be used to reproduce the experimental data. Running the jar requires IBM ILOG CPLEX optimization studio.

All solver algorithms reside inside the *solving* package, the set of games used for experiments can be found inside the *domain.GameSet* class. The *utility.GamePrinter* class can be used to construct a Graphviz file representing a game, it will also automatically produce a PNG image file if Graphviz is added to the PATH environment variable.

Example usage of the jar file: *java -Djava.library.path="<path to IBM CPLEX>" -jar algorithmComparison.jar*. Running this command will reproduce the values in the tables in Chapter 8 by running the three algorithms for a maximum of ten minutes on each of the games in the experimental game set. Output will be saved to a folder named 'results' in the current working directory.