Improving Agent Performance in an Alternating Offers Negotiation Game

A Thesis presented

by

Konstantin Pozin

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 1, 2010

Contents

1 Introduction **4**

2 Background **6**

2.1 Colored Trails 6

2.2 The Alternating Offers Domain 7

2.3 Related Work 10

3 Agent Design **12**

3.1 Baseline Design 12

3.2 A Learning Agent 15

3.2.1 Myopic Reliability Model 15

3.2.2 Deception at the End of the Game 17

3.2.3 Game Tree Search 17

3.2.4 Calculating Probabilities with Predictors 21

4 Methodology **25**

4.1 Peer-Designed Agents 25

4.2 Generating and Testing Classifiers 25

4.3 Testing Agent Performance 26

5 Results and Evaluation **27**

5.1 Transfer Strategy Based on Myopic Reliability 27

5.2 Myopic Reliability with End-of-Game Deception 28

5.3 Tree Search and Predictors 28

5.3.1 Predictor Quality 28
5.3.2 Proposal Strategy Based on Game Tree Search 30
5.4 Summary of Results 30

6 Conclusion **32**

Bibliography **34**

Acknowledgements

I would like to thank Dr. Kobi Gal for his guidance and support throughout the project, and Professor Stuart Shieber for his insights and advice. I am also very grateful to Yael Blumberg for providing training data, to Bart Kamphorst and Dimitrios Antos for their technical guidance in working with Colored Trails, and to Skye Isard and Swapna Reddy for their help in administering early experiments.

Chapter 1

Introduction

One area of artificial intelligence research that has enjoyed some attention in recent years is the use of computer agents as proxies or representatives for humans in interactions and transactions. Examples include auction bots that follow a human's directives in order to bid and make purchases in online auctions, or agents that might automate certain stock trades. Such agents interact in *open systems* — arenas in which agents represent, or are designed by, different entities. Those who design agents for such scenarios cannot control the behavior of opponents, and cannot know in advance how these opponents might act.

We refer to computer agents that are created by individuals to follow a particular strategy as *peer-designed agents* (PDAs). When we create agents to compete against PDAs in a particular domain, we face certain problems. A PDA may have been *intended* by its human designer to act as the designer himself would expect to act in similar scenarios. However, we cannot make this assumption. There has not been research indicating that techniques for modeling the social behaviors and personalities of humans have not been shown to be generalizable to modeling the behaviors and "personalities" of human-designed robots. Moreover, it has been shown that people can be ineffective at accurately describing their own strategies in competitive scenarios, and tend to behave differently from the agents they design (Chalamish et al., 2008).

The goal of our work is to design a learning computer agent that can compete against PDAs in a particular domain of interaction: the "Alternating Offers" scenario implemented using a research framework called Colored Trails (described in Sect. 2.1). More specifically,

we aim to create an agent that can compete successfully while treating every opponent as a "black box" — an agent whose behavior pattern and utility function are not initially known and cannot necessarily be modeled. In this paper we incrementally develop and test several behavior components for an agent:

1. A simple method for modeling the short-term reliability of another player
2. A simple deception strategy that improves the agent's performance against many PDAs
3. An Expectimax algorithm for generating and searching a partial game tree in order to make game play decisions
4. Statistical classifiers for modeling opponent behavior within the game tree

Chapter 2

Background

2.1 Colored Trails

Colored Trails (hereafter abbreviated as CT) is a framework for research in decision-making, designed by Barbara Grosz and Sarit Kraus (Grosz et al., 2004). It enables researchers to easily set up board games that can be used to model simple real-world tasks involving resources and limited interactions.

CT scenarios are played on an $n \times m$ board of colored squares. In a typical game, one square is marked as the "goal square." Two or more players are represented on the board with distinct icons, and are given the task of reaching the goal. In order to move to an adjacent square, a player must expend a resource called a "chip," the color of which corresponds to the square onto which he is moving. Depending on the game scenario, players may begin with different allocations of chips, and may be expected to exchange some chips with their opponents in order to complete a task. The experimenter is able to specify the rules of the interaction, the scoring function, the end conditions, and what game information is visible to each player. Scoring is usually based on whether a player reached the goal or how far he was from the goal at the end of the game, and on the number of chips remaining in his possession.

From a technical perspective, CT is implemented as a set of Java applications. A CT *server* hosts the games, to which *client* applications (either agents or GUIs for human users) connect using Java Messaging Service (JMS). The experimenter then launches a *controller*,

which tells the server to start a game or set of games. The server dynamically loads a *configuration class*, which is a Java class customized by the experimenter that provides the rules and specifications for the scenario.

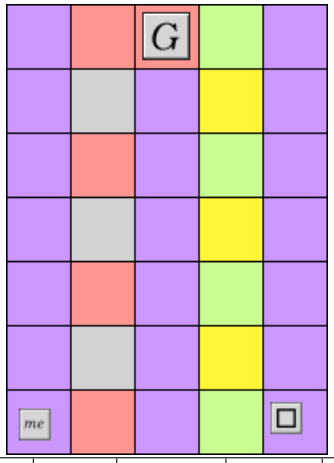
2.2 The Alternating Offers Domain

The Alternating Offers domain for CT (hereafter "AltOffers") was designed by researchers at Harvard and MIT, based on a game-theoretic scenario originally created by Ariel Rubinstein. In the AltOffers domain, two players are placed on the board and attempt to reach a single goal. The interaction consists of a series of rounds in which the players may negotiate offers to exchange chips, carry out exchanges, and move their icons on the board. AltOffers is a game of complete information; each player always knows where his opponent is located and what chips the opponent possesses.

From a game-theoretic perspective, AltOffers presents an interesting challenge through its exchange mechanism. Exchange agreements are not binding — a player is free to promise to send his opponent some set of chips, and then to renege on his promise by a sending only a subset of the agreed-upon chips. (A player is also free to send chips that he did not promise, or to send chips even if no agreement was reached.) Therefore, a player must take into account not only the exchange values that is decided upon, but also how reliable or generous the opponent will be, and how reliably the player himself should behave. Because interactions take place over several rounds, AltOffers is not a one-shot game; players must be wary of their reputations, and reason about how their opponents will retaliate or reward them for their actions.

AltOffers is played on a 7×5 board. At Round 0, the two players start out in the lower corners of the board: Player 0, on the left, begins at position $(6, 0)$, while Player 1 begins at $(6, 4)$. The goal icon is fixed in the top center at $(0, 2)$.

A player is considered *task-independent* (TI) if he possesses all of the chips necessary to reach the goal; a player who lacks some of the needed chips is called *task-dependent* (TD) because he is dependent on the other player for the chips he needs to reach the goal. We use two general conditions for the initial allotment of chips in a game:



	Red	Green	Gray	Yellow
Player 0	5	10	0	9
Player 1	1	13	10	0

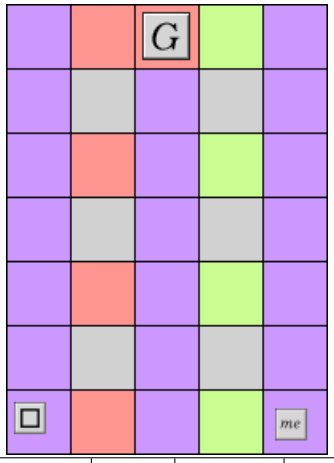
Figure 2.1: DD board (Player 0 on the left, Player 1 on the right)

Dependent-dependent (DD). At the beginning of the game, each player is dependent on the other. In Figure 2.1, Player 0 lacks three gray chips, while Player 1 lacks three yellow chips. Each would be able to provide the other with the missing chips in an exchange.

Dependent-independent (DI or ID). At the beginning of the game, one player is dependent on the other. In the DI scenario in Figure 2.2, Player 0 on the left lacks three gray chips he needs to reach the goal, while Player 1 on the right has all the chips he needs and would be able to supply Player 0 with the needed chips in an exchange. In an ID scenario, the board and chip allocations are mirrored. In the DD as well as the DI/ID scenarios, both players possess a number of extra chips that they do not require to reach the goal; a dependent player is able to use these extra chips to bargain for chips of the color that he actually needs.

Every game consists of a variable number of rounds. A round consists of the following sequence of phases:

1. **Communication Phase.** The players exchange proposals. One player is initially the *proposer* (on Round 0, the proposer is always Player 0, who is on the left player). He can make an offer to send some subset of the chips he possesses in exchange for some subset of the *responder*'s chips. The responder can accept or reject. If he rejects, he becomes the proposer and can make a counter-offer. There is a maximum of two such



	Red	Green	Gray
Player 0	5	10	0
Player 1	1	13	10

Figure 2.2: DI board (Player 0 on the left, Player 1 on the right)

proposals in a Communication Phase.

2. **Chip Exchange Phase.** Each player is able to transfer *any subset* of his available chips to the other player. In other words, any agreement made in the preceding Communication Phase is *non-binding*. Neither player is aware how many chips the other sent until the phase is over, so a player cannot wait and see what his opponent sends before making his own decision.

3. **Movement Phase.** Each player can move a single square if he has the available chips, expending, for example, one red chip if he moves to a red square.

In games with at least one human player, Round 0 begins with an extra *Strategy Preparation* phase in which human players have some time to plan a path to the goal. Computer players do not do anything during this phase.

The game can end under two possible conditions:

- Both players acquire enough chips to reach the goal. In this case, the system automatically advances each player to the goal (expending the appropriate chips) and the game ends.

- At least one of the two players remains *dormant* (does not move) for three consecutive rounds. In this case, the system moves each player towards the goal in a way that

would maximize his score to match his *BestUse* score (see below) for the round, and the game ends.

The scoring function for each player is as follows:

- 100 points for reaching the goal
- 5 points for each chip that the player possesses at the end of the game
- -10 points for each square in the path from the player's final position to the goal

A player's *BestUse* score is calculated as follows. For a player i, given a current position X and a current chipset C, we find $BestUse(X, C)$ by iterating over every possible *path* (sequence of moves) the player can take from X, and maximizing the resulting score.

2.3 Related Work

A number of papers have described work involving evaluation of and competition against peer-designed agents. Several studies have demonstrated the potential incongruity between people's intended agent strategies and the actual behavior and performance of the agents they design. In a study by Manisterski et al. (2008), a group of graduate students attempted to progressively improve the strategy of agents that they had designed for a trading game. Instead, it was found that the modified agents performed worse with each revision. Chalamish et al. (2008) had a large group of computer science undergraduates play one of four different games numerous times in order to achieve some proficiency, and then asked them to design agents that would follow their own strategies; it was found that these agents' actual behavior were dramatically dissimilar to what was intended by their designers.

Several related papers have used Colored Trails scenarios in their work. A paper by Talman et al. (2005) presented research very similar to our paper. They designed agents to compete in a negotiation game almost identical to AltOffers, in which two agents negotiate chip exchanges in order to obtain the chips necessary to reach the goal, and are able to renege on agreements. The primary difference in their scenario was that players did not not know which chips their opponents actually possessed.

Instead of attempting to model opponents' utility functions as in some other studies, Talman et al. programmed their agents to categorize opponents into different personalities using two variables (cooperation and reliability) at three levels (low, medium, and high), based on the opponents' behavior. All tested agents were further categorized as *helpful* or *unhelpful*. The researchers' agents could be either single-personality (having a fixed personality for all games) or multiple-personality (adopting different personalities depending on the current opponent). The various classes of agents were tested against each other and against a set of PDAs created by graduate students. The researchers found that *helpful* agents tended to fare better when they were *dependent* and to cooperate with their opponents, but were exploited when they were *independent*. It was found that in general, multiple-personality agents outperformed all other agents.

Carr et al. (2009) propose a method for reducing game tree branching factors in CT games by simplifying exchange proposals into *meta-proposals*. Rather than considering the individual counts for every chip color sent by both players, the researchers group all chip counts into four equivalence classes depending on who needs them to reach the goal: (1) chips needed by both players, (2) chips needed by the agent but not the opponent, (3) chips needed by the opponent but not the agent, and (4) extra chips not needed by either player. Every proposal is then represented by a single 4-tuple. Carr et al. ran a tournament of 525 matches amongst 21 PDAs, and used several features including the meta-proposals in order to train four behavior predictors: (a) whether a proposal would be accepted, (b) whether a player would renege on an agreement, (c) whether a player would make a proposal, and (d) what the proposal would be. The researchers designed algorithms that would use these predictors to narrow the search tree, but were unsuccessful in implementing working versions.

Chapter 3

Agent Design

3.1 Baseline Design

The baseline agent whose performance we aim to improve was designed by Sarit Kraus of Bar-Ilan University. It is called a Personality Rule-Based (PERB) agent, and is intended to have a medium level of reliability in terms of how well it keeps its side of agreements. The following is a summary of its behavior (Kraus, 2009).

Personality Categorization

The agent categorizes its opponents into low, medium, and high cooperation and reliability, as in Talman et al. (2005).

Utility Function for Exchanges

The agent's utility function for an exchange, U, is based on an estimation of the agreement's benefit to the agent (U_{agent}) as well as to the opponent (U_{other}). For a given player i, U_i is function based on player i's expected score if the promised exchange takes place (including an approximate probability of eventually reaching the goal), and on a simple model of the other player's "opinion" of player i after the exchange.

The value of U is $(a \cdot U_{agent} + b \cdot U_{other})$, where a and b are constants and $a : b$ is determined by the two players' dependency statuses (who is task-dependent and who is task-independent), as well as which player is making the offer.

Generating an Offer

The agent makes an offer by

1. generating a set of possible offers S
2. evaluating each offer with the utility function U, where the highest utility over S is U_{max}.
3. keeping a subset offers $R \subseteq S$, for each of which $U \geq U_{max} - \epsilon$ (where ϵ is a constant)
4. choosing a random offer from R

The initial offers that are included in S depend on the players' respective dependency statuses and on their personalities. In general, if both players are task-dependent, most possible offers will be 1:1. If one player is "stronger" than the other, meaning that he needs fewer chips to reach the goal (or is already task-independent), then most offers will be made at a ratio of 2:1 in favor of the stronger player. For example, if the agent is dependent and the opponent is independent, the agent will offer to send twice as many chips as it would receive.

Responding to an Offer

The following description of the PERB agent's algorithm for responding to an offer is adapted directly from Sarit Kraus's specification:

1. If the offer requires the agent to send a chip it needs in order to reach the goal, it rejects the offer.
2. When responding to a counter-offer (the second offer in one Communication Phase): if the agent must move to avoid ending the game, and the offer will enable it to move, the agent accepts.
3. When responding to a counter-offer: if the agent is TI, and the other player needs a chip to move to avoid ending the game, then the agent will compare the utilities of (a) accepting the offer and (b) sending just the required chip to the other player. If the utility of the offer is higher, the agent accepts.

4. If the opponent is TI and the agent is TD, and the offer (a) has a chip ratio of at most 1 : 2 for *chipsSentByOpponent* : *chipsSentByAgent*, and (b) provides the agent with required chips, then the agent accepts.

5. Otherwise, the agent computes the utility of

   - the proposed agreement (U_a)
   - doing nothing (U_n)
   - the offer that the agent itself would have offered in this setting (U_o)

   Then:

   (a) If $U_n > I_a$, the agent rejects.

   (b) If there have been no earlier agreements in the game (and the agent therefore has no information on the opponent's reliability), and the agreement would shift the opponent from TD to TI, then the agent rejects.

   (c) If $U_a + \epsilon > U_o$, then the agent accepts.

   (d) Otherwise, the agent rejects.

Sending Chips After an Agreement

1. On the first agreement, the agent always sends all promised chips.

2. If the opponent has Low reliability, the agent sends no chips.

3. If the opponent has High reliability, the agent sends all chips.

4. If the opponent has Medium reliability, the agent follows a rather complicated set of rules that can be read in the PERB agent specification (Kraus, 2009). In general, the agent (a) tries to make sure to send a chip if *not* sending it would cause the game to end, and (b) sends either all or "part" of the chips with some probability that is determined by the opponent's reliability level.

5. If the agent is TI, the opponent needs a chip to move, and sending this chip would not make the opponent TI, the agent sends the chip.

Moving

If the agent is task-independent, it will advance one step on each movement phase. If the agent is task-dependent, it will move only when not moving would cause the game to end (if the agent has already been dormant for two rounds).

3.2 A Learning Agent

The PERB agent described above suffers from a number of design flaws. Its behavior is determined by a large set of *ad hoc* rules that are not based on any clear game-theoretic principle. Although it does does create and update a basic model of its opponent (through measures of cooperativeness and reliability), this model's features are only used to choose arbitrary parameters for the PERB agent's actions. No reasoning about the opponent's expected behavior takes place. Moreover, proposals are generated and considered based on ratios of unweighted chip counts, rather than ratios of utility values.

Our goal is to design and implement an agent that addresses some of these issues. We intend to create an agent that will:

- model its opponents using continuous distributions, rather than divisions into levels
- use a game tree to determine at least part of the agent's behavior, using the opponent model to reason about expected utilities

In this paper, we incrementally replace parts of the PERB agent's behavior with our own, as described below. As a first step, we decided to replace the PERB agent's chip sending behavior with an algorithm based on our myopic reliability model.

3.2.1 Myopic Reliability Model

We present a simple reliability model that can be used as a starting point for creating a learning agent:

Let the agent be Player 0 and the opponent Player 1. At the beginning of a round r, Player i's position is given by $Pos_i(r)$ and his chipset is given by $Chips_i(r)$. Then

let the two players' initial *BestUse* scores for Round 0 be respectively defined as $B_0(r) = BestUse\left(Pos_0\left(r\right), Chips_0\left(r\right)\right)$ and $B_1(r) = BestUse\left(Pos_1\left(r\right), Chips_1\left(r\right)\right)$.

Let the final accepted offers be $PromisedToSend_0(r)$ and $PromisedToSend_1(r)$, representing the chipsets that the respective players agree to send at the end of the Communication Phase for round r.

Let the chips that each player *actually* sent by given by $ActuallySent_0(r)$ and $ActuallySent_1(r)$. Then we define the *reliability* of Player 1 for round r as follows:

$$Reliab_1(r) = \begin{cases} \frac{BestUse(Pos_0(r), Chips_0(r) + ActuallySent_1(r)) - B_0(r)}{BestUse(Pos_0(r), Chips_0(r) + PromisedToSend_1(r)) - B_0(r)} \\ DefaultReliability \text{ if the fraction is } \frac{0}{0}\text{, implying no agreement} \\ 1 \text{ if just the denominator is 0} \end{cases}$$

In other words, a player's reliability for a given round is the ratio of the *actual* improvement in his *BestUse* score to the *promised* improvement, based on the chips sent by his opponent and the chips promised by his opponent, respectively. When the opponent sends all the chips it promised on a given round, his reliability for that round will be 1. Understandably, if he sends fewer chips than promised, his reliability will be between 0 and 1, and if he sends more chips, the reliability will be greater than 1.

Sending Strategy Based on Myopic Reliability Model

In the current version of our agent, we use the reliability model described above to calculate the chips that the agent should send to its opponent during the exchange phase. Assume that the agent is Player 0. The agent's *promised* chipset during round r, $PromisedToSend_0(r)$, is determined by the proposal strategy described in Section 3.1. We use the reliability model to calculate the subset of the promised chips that would bring the agent's behavior as close as possible to "tit-for-tat" — in other words, we attempt to make the agent as reliable on the current round as its opponent was on the previous round. In practice, considering *only* the previous round is overly myopic. For this reason, we take into account the last *two* rounds when possible, using the weight w for Round $(r-1)$ and $1-w$ for Round $(r-2)$.

Let $TargetReliab_0(r+1) = w \cdot Reliab_1(r-1) + (1-w) \cdot Reliab_1(r-2)$. We define each

player's reliability for the zeroth round, $Reliab_i(0)$, to be some constant $DefaultReliability$.

Let the chipset $X = ActuallySent_0(r+1) \subseteq PromisedToSend_0(r+1)$. Then our goal is to choose an X such that

$$X = \arg\min_{X^*} |TargetReliab_0(r+1) - Reliab_0(r+1)|$$

More specifically, the chipset X is calculated as follows:

$$X = \arg\min_{X^*} \left| TargetReliab_0(r+1) - \frac{BestUse(Pos_1(r+1), Chips_1(r+1)+X^*) - B_1(r+1)}{BestUse(Pos_r(r+1), Chips_1(r+1)+PromisedToSend_0(r+1) - B_1(r+1))} \right|$$

3.2.2 Deception at the End of the Game

In preliminary trials using the above sending strategy (see Section 5.1), we found that AltOffersLearning fared poorly in most games because of its high level of "trust." As long as the opponent PDA keeps all its promises early in the game, the reliability model above dictates that AltOffersLearning itself should keep its promise on subsequent turns. However, the PDAs often take advantage of this naive trust by reneging on promises as soon they become task-independent. To counter this deception, we introduced a deceptive behavior into AltOffersLearning's sending strategy: the agent never completes a transfer that would make the opponent task-independent (if the opponent is not already independent). This provides a further advantage against those PDAs that naively keep negotiating and sending the agent additional chips in hope of becoming task-independent, even when the agent's behavior history would imply that it is too unreliable.

Algorithm 3.1 Never make the opponent independent

repeat
    chipsToSend ← removeRandomChips(*chipsToSend*, 2)
until receiving *chipsToSend* would not make opponent independent

3.2.3 Game Tree Search

We developed a partial game tree search to replace or augment some of the rule-based behavior in the baseline agent. A simple Expectimax algorithm is used for the tree search, which is initiated when it is the agent's turn to make an offer to the opponent. The tree is

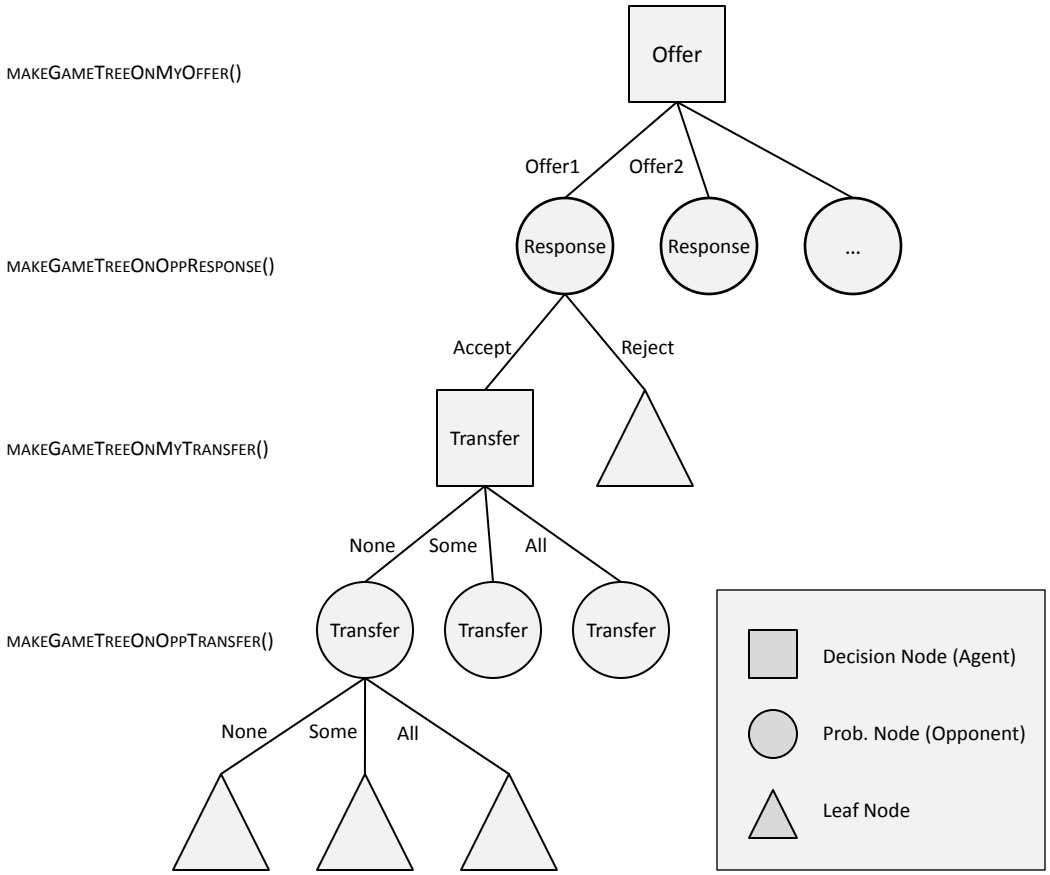


Figure 3.1: Game tree from agent's offer generation

only generated when the agent needs to make an offer, and is limited in depth to a single round of the game. Figure 3.1 shows the structure of the tree, consisting of *nodes* linked by *action* edges.

Nodes

Every node has a game state reflecting the positions and chips of the two players, as well as the last offer made, if any. Each node also has some calculated *utility* with respect to the agent; this value is directly related to the final *BestUse* score that an agent might earn as a consequence of the game state. There are three types of nodes:

Decision nodes (squares in the diagram) are points at which the agent chooses an action by picking the child node that has the best utility. The utility of a decision node

is equal to *the utility of its best child*. This is the MAX part of Expectimax; the search algorithm tries to maximize the utility at decision nodes.

Probability nodes (circles in the diagram) are points at which the opponent chooses an action. Because the opponent's action is uncertain, we generate a list of possible actions with a corresponding set of probabilities. The utility of a probability node is equal to the *average utility of its children*, weighted using the probability of each child. This is the EXPECTI part of Expectimax; the algorithm calculates the expected utility at probability nodes.

Leaf nodes (triangles in the diagram) are points at which we stop searching the tree further — either because the game ends at that point, or, as in our case, because we are limiting the size of the game tree by stopping at this point. The utility of a leaf node is *the* BestUse *score for the agent*, given the game state contained in the leaf node.

Actions

There are three types of actions used in our tree search.

Offer actions describe a proposal for an exchange, and consist of the two respective chipsets to be sent by the proposer and the responder.

Response actions describe whether a proposal is accepted or rejected.

Transfer actions describe a transfer of chips from one player to another. As can be seen in Figure 3.1, we split the exchange phase into *two* actions for the purposes of the Expectimax algorithm, even though the transfers actually take place simultaneously. In the tree search, the agent first chooses a chipset to transfer, and then we make a guess about the chipset that the opponent transfers. This does not mean, however, that the opponent has any knowledge of the agent's transfer in advance.

The Search Algorithm

When the agent has to make an offer during the Communication Phase, the method MAKE-GAMETREEONMYOFFER is called. This generates a copy of the current game state and then places the state data into a new root node. The method then retrieves a set of potential offers generated by PERB's rules (see Section 3.1). For each possible offer, a new

node is created as a ProbabilityNode and added to the root's list of child nodes, linked by a corresponding TransferAction edge.

Algorithm 3.2

```
1: procedure makeGameTreeOnMyOffer
2:     root ← new TreeNode(current game state)
3:     S ← generateOffers                          ▷ use the PERB rules to generate offers
4:     for all s ∈ S do
5:         action ← new OfferAction(offer)         ▷ make a new edge representing the offer
6:         child ← new ProbabilityNode(root state with offer s)
7:         makeGameTreeOnOppResponse(child)
8:         root.addChild(action, child)
9:     end for
10:    return node.chooseBestChild()
11: end procedure
```

Before completing makeGameTreeOnMyOffer, we recurse down and call makeGameTreeOnOppResponse on each child node. This method generates new actions and nodes in which the offer has either been accepted or rejected, assigns probabilities to both actions, and adds them as children of the ProbabilityNodes. In this paper, we are not considering cases where the opponent rejects the agent's offer and makes a counter-offer, so a *reject* action links to a LeafNode. For the *accept* action, we create a new DecisionNode, on which we call makeGameTreeOnMyTransfer.

Algorithm 3.3

```
1: procedure makeGameTreeOnOppResponse(ProbabilityNode node)
2:     rejectAction ← new ResponseAction(REJECT)
3:     rejectNode ← new LeafNode(REJECT)
4:     acceptAction ← new ResponseAction(ACCEPT)
5:     acceptNode ← new DecisionNode(ACCEPT)
6:     makeGameTreeOnMyTransfer(acceptNode)
7:     P ← getProbabilities({acceptAction, rejectAction})
8:     node.addChild(acceptAction, acceptNode, P[acceptAction])
9:     node.addChild(rejectAction, rejectNode, P[rejectAction])
10: end procedure
```

Using the offer information that has been passed down through the tree, makeGameTreeOnMyTransfer generates a set of TransferActions and ProbabilityNodes representing the possible transfers that the agent can make, where the game state in each new node reflects the subtraction of chips from the agent and the addition of those chips to the op-

ponent. To avoid generating branches for the entire powerset of the promised chipset, we only generate three possible transfers, representing *none*, *some*, or *all* of the promised chips. The *some* transfer is chosen by taking a random subset of the promised chips. For each of these child nodes, we call makeGameTreeOnOppTransfer.

Algorithm 3.4

```
1: procedure makeGameTreeOnMyTransfer(DecisionNode node)
2:     allChips ← chips for agent to send from node.lastOffer.chipSet
3:     noChips ← new ChipSet()
4:     someChips ← random subset of allChips
5:     transferAll ← new TransferAction(allChips)
6:     transferNone ← new TransferAction(noChips)
7:     transferSome ← new TransferAction(someChips)
8:     for all action ∈ {transferAll, transferNone, transferSome} do
9:         child ← new ProbabilityNode based on node state, with the transfer action
   applied
10:        node.addChild(action, child)
11:        makeGameTreeOnOppTransfer(child)
12:    end for
13:    node.chooseBestChild()
14: end procedure
```

makeGameTreeOnOppTransfer creates another set of possible transfers of *none*, *some*, or *all* of the opponent's promised chips. The *some* transfer is again determined by choosing a random subset of the promised chips. We assign probabilities to each of these transfers, create the appropriate TransferActions and new nodes, and add them as child nodes. At this point, all the child nodes are LeafNodes; the game tree is thus confined to a single round of the game.

Once the entire game tree has been recursively generated, the top-level method, makeGameTreeOnMyOffer, initiates the search. The root node's instance method chooseBestChild is called; this chooses the child node with the best Expectimax utility by recursively calling getUtility and/or chooseBestChild from all deeper nodes in the tree.

3.2.4 Calculating Probabilities with Predictors

The game tree search requires a means of calculating probabilities at the ProbabilityNodes in the tree, specifically to determine (a) whether the opponent will accept the agent's proposal, and (b) how well the opponent will keep its promise to transfer chips. This can

Algorithm 3.5

```
procedure makeGameTreeOnOppTransfer(ProbabilityNode node)
    allChips ← chips for opponent to send from node.lastOffer.chipSet
    noChips ← new ChipSet()
    someChips ← random subset of allChips
    transferAll ← new TransferAction(allChips)
    transferNone ← new TransferAction(noChips)
    transferSome ← new TransferAction(someChips)
    P ← getProbabilities({transferAll, transferNone, transferSome})
    for all action ∈ {transferAll, transferNone, transferSome} do
        child ← new LeafNode based on node state, with the transfer action applied
        node.addChild(action, child, P[action])
        makeGameTreeOnOppTransfer(child)
    end for
end procedure
```

be accomplished through the use of a statistical classifier.

A classifier is first trained on a set of training instances. An *instance* is a single data point consisting of multiple features. Features can be *numeric*, where the values are all real numbers in some arbitrary range, or *nominal*, where every value is chosen from a finite, discrete list of possible values. In our case, each instance represents information about a single proposal made by a player. Numeric features may include players' reliability and current and potential scores based on that proposal. Nominal features include whether the proposal was accepted (*true* or *false*), and transfer classes (whether a player transferred *none*, *some*, or *all* of the chips he promised). For an instance I with a given set of features (i_1, i_2, ..., i_n), a classifier can be trained, for example, to use features (i_1, i_2, ..., i_{n-1}) to make guesses about feature i_n (the output). Training the classifier results in the creation of a model — an algorithm with a set of parameters that can be used to classify new, previously unseen instances.

If the output feature is nominal, the model should return a discrete probability distribution for the sample space (list of possible values). Often, the "correct" value is assumed to be the one with the highest probability; in our case, however, we are interested not in the most likely value, but in the entire probability distribution, which we will use within the Expectimax algorithm to calculate expected utility at ProbabilityNodes. Table 3.1 shows the two predictors that we integrated into our tree search. OpponentResponsePredictor

	OpponentResponsePredictor	OpponentTransferPredictor
Features	prevPlayerReliability prevOpponentReliability playerCurrentBestUseScore playerPromisedBenefit opponentCurrentBestUseScore opponentPromisedBenefit prevPlayerTransferClass prevOpponentTransferClass	prevPlayerReliability prevOpponentReliability playerCurrentBestUseScore playerPromisedBenefit opponentCurrentBestUseScore opponentPromisedBenefit prevPlayerTransferClass prevOpponentTransferClass
Output	**opponentResponse** {*accept*, *reject*}	**opponentTransferClass** {*none*, *some*, *all*}

Table 3.1: List of features in the two predictors.

predicts whether the opponent will accept an offer and is used in makeGameTreeOnOppResponse; OpponentTransferPredictor predicts the opponent's transfer class and is used in makeGameTreeOnOppTransfer.

The following is an explanation of the features:

- *prevPlayerReliability, prevOpponentReliability* — This is each player's myopic reliability value from the *last accepted offer* (and corresponding transfer). If there are no accepted offers before the current one, then this value is missing.

- *playerCurrentBestUseScore, opponentCurrentBestUseScore* — This is each player's current best-use score for each player, using the chips possessed at the beginning of the current round.

- *playerPromisedBenefit, opponentPromisedBenefit* — This is the difference between *CurrentBestUseScore* and *PromisedBestScore* (the latter not included as a feature). In other words, this is the number of points that the player has been promised he would be "given" by his opponent.

- *prevPlayerTransferClass, prevOpponentTransfer* — This value summarizes whether each player transferred none, some, or all of the chips he promised on the *last accepted offer*. This value is directly calculated from reliability values using the table below. If there is no earlier accepted offer, then these attribute values are missing.

Reliability	$r = 0$	$0 < r < 1$	$r \geq 1$
TransferClass	none	some	all

- *opponentResponse* — This describes whether the opponent accepts or rejects the current offer.

- *opponentTransferClass* — If the current offer is an accepted offer, this describes whether the opponent will transfer none, some, or all of the promised chips.

Weka

Rather than implement classifiers from scratch, we used Weka, a free, open-source Java-based application for data mining and machine learning (Hall et al., 2009). Weka includes a large set of classifiers that can be trained and tested using a graphical interface, the command line, or integration into a Java application. After testing several available classifiers on our training data for proposals, we chose to use Weka's J48 classifier, which had one of the highest degrees of accuracy for this dataset (see Section 5.3.1). J48 is an implementation of the *C4.5* algorithm for generating decision trees; the algorithm is described by Quinlan (1993). For each predictor, we used the Weka GUI to train the classifier offline and to export a serialized model. We then created wrapper functions around Weka's Java library to integrate the classifier into the agent's code by loading the model dynamically at runtime, creating new instances from the current game state, and classifying them during the game tree search.

Chapter 4

Methodology

4.1 Peer-Designed Agents

We had available to us a set of 19 PDAs for AltOffers that were created by undergraduate computer science students at Bar-Ilan University in 2009. Each of these PDAs was placed into the Java source tree for our project and compiled together with rest of our Colored Trails codebase. This ensured that all agents would use the same core CT codebase, thus avoiding class version problems that would otherwise have arisen in JMS communications.

4.2 Generating and Testing Classifiers

We used a set of structured logs from a tournament that was run among the 19 PDAs by Yael Blumberg at Bar-Ilan University. For the DD board, we had logs of offers and transfers for a total of 337 games. Using a Python script, we extracted values for the features listed in Table 3.1 into CSV files that we could import directly into Weka. In order to use all possible training instances, each offer was used to create two training instances for the classifier — one from the perspective of the proposer, and one from the perspective of the responder.

For the OpponentResponsePredictor, we had a total of 3976 training instances. For the OpponentTransferPredictor, we were limited to accepted offers, since a transferred chipset can only be classified in relation to the promised chipset; we had 1860 instances for this predictor.

We used the Weka GUI to load the training sets and generate our classifiers. We then tested the classifiers directly in Weka using its built-in cross-validation mechanism, with 10 folds.

4.3 Testing Agent Performance

To test the various versions of our agent (called *AltOffersLearning*), we compared its performance to the baseline performance of Kraus's agent (PERB). Incorporating some utility code for launching Windows processes (written by Yael Ejgenberg), we created a partially automated tournament system for Colored Trails. This system loads tournament settings from two external text files:

- an *agent definition* file that lists identifiers for the various CT agents we wish to run, and the the Java runtime arguments needed to execute them
- a *tournament definition* file, which includes the CT runtime commands needed to start the server and to start the controller, followed by a list of agent pairs that should play against each other in the tournament

A tournament is run by providing the file names of the two definition files to a particular Java executable; this executable starts the CT server, starts the agents, and then starts a controller specially programmed for the tournament system. A single server then runs all the games in the tournament simultaneously (about 10-20 games per tournament can be run on a relatively recent PC).

We evaluated the performance of AltOffersLearning and the various configurations of PERB by having them play against the set of 19 PDAs described above.

Due to an undetermined defect in our agent code that emerged when running multiple instances of the agent simultaneously on the the imbalanced boards (ID and DI), we were unable to gather valid data for those two boards. Therefore, all of our experimental data is based on the DD board.

Chapter 5

Results and Evaluation

5.1 Transfer Strategy Based on Myopic Reliability

Our first test compares the performance of PERB against a version of AltOffersLearning which differs from PERB only in that it uses a sending strategy based on the myopic reliability model (Section 3.2.1). Table 5.1 shows, for each of the two agents, average scores for two tournaments of 19 games against the PDAs. (The other tables below are also based on two tournaments of 19 games.) Wins, ties, and losses reflect the number of times that the agent's score was greater than, equal to, or less than the score of the competing PDA.

PERB does poorly compared to the set of PDAs, losing by an average of 25.4 points. Our agent's absolute performance is slightly worse than, but still very similar to, that of PERB. (In fact, PERB's average score in one of the two tournaments is identical to AltOffersLearning's average score in a tournament). However, the *opponent PDAs' average score* is higher for AltOffersLearning, suggesting that transferring based on our myopic reliability model alone causes AltOffersLearning to be more generous than necessary.

	Agent's avg	PDAs' avg	Wins	Ties	Losses
PERB	99.2	124.6	2	11	6
Learning - Reliability	**96.7**	**135.7**	**3.5**	**9.5**	**6**

Table 5.1: PERB vs. AltOffersLearning with Myopic Reliability

	Agent's avg	PDAs' avg	Wins	Ties	Losses
PERB - Deception	124.5	41.8	10	7	2
Learning - Reliability & Deception	**112.0**	**41.6**	**10.5**	**6.5**	**2**

Table 5.2: PERB with Deception vs. AltOffersLearning with Myopic Reliability and Deception

5.2 Myopic Reliability with End-of-Game Deception

Based on these preliminary results, we decided to introduce end-of-game deception, overriding AltOffersLearning's sending strategy in cases where the reliability model strategy would cause it to make the opponent task-independent (see Section 3.2.2). As shown in the bottom row of Table 5.2, this resulted in a noticeable improvement in the agent's performance, and a drastic falloff in the scores of the PDA opponents.

For the sake of completeness, we also ported the deception strategy to PERB in order to enable a more direct comparison. The addition of a deceptive sending behavior had a much more pronounced effect on performance in PERB than in our agent; PERB had an average increase of **25.3** points, compared to AltOffersLearning's **16.3**. This has several implications. First, PERB is much more generous and reliable than necessary; PERB's performance can clearly be improved by reneging on agreements that would make the opponent independent. Second, although PERB's performance is not very good to begin with, our myopic reliability model is particularly sub-optimal in comparison; it severely dampens the improvement that can be achieved by introducing deception. A myopic tit-for-tat sending strategy is clearly not an effective one, at least against the set of PDAs that we have been testing.

5.3 Tree Search and Predictors

5.3.1 Predictor Quality

Weka includes a collection of several dozen classifiers, with a variety of classification methods and memory/time/accuracy trade-offs. For our two classification tasks, we found the C4.5 classifier (called "J48" in the Weka collection), to be the most accurate. Accuracy is measured based on the percentage of correctly classified instances; "correct" clas-

		Predicted	
		accept	*reject*
Actual	*accept*	1405	729
	reject	697	1145

Table 5.3: Confusion matrix for OpponentResponsePredictor (C4.5)

		Predicted		
		none	*some*	*all*
Actual	*none*	759	38	279
	some	16	38	14
	all	84	60	572

Table 5.4: Confusion matrix for OpponentTransferPredictor (C4.5)

sification, in turn, occurs when the distribution's highest probability value corresponds to the correct class. For example, if we have an opponent transfer instance with a result of *some*, and the OpponentResponsePredictor returns a probability distribution of ($P(none)$, $P(some)$, $P(all)$) = (0.15, 0.6, 0.25), then the predictor's classification is correct because $P(some) = 0.6$ is the largest value in the distribution.

For the OpponentResponsePredictor, C4.5 yielded an accuracy of **72.16%** in 10-fold cross-validation. (By comparison, a Bayes net classifier had an accuracy of only **64.13%**). Table 5.3 shows the confusion matrix for OpponentResponsePredictor.

For the OpponentTransferPredictor, C4.5 was even more impressive, with an accuracy of **81.45%** (compared to **73.60%** for Bayes net). Table 5.4 shows the confusion matrix for OpponentResponsePredictor.

In both cases, the classifiers performed significantly better than chance, indicating that the PDAs' behavior was to a large degree predictable. Our classifiers' accuracy might be further improved by incorporating additional attributes from the game state; it is almost certain that the behavior of many PDAs relies on additional parameters (such as number of dormant turns, distance to goal, cumulative reliability, etc.) that we did not implement in this version of the predictors.

	Agent's avg	PDAs' avg	Wins	Ties	Losses
PERB - Deception	124.5	41.8	10	7	2
Learning - Reliability & Deception	112.0	41.6	10.5	6.5	2
Learning - Reliability, Deception, Tree	**105.1**	**42.1**	**11**	**7**	**1**

Table 5.5: AltOffersLearning with Myopic Reliability, Deception, and Tree Search

5.3.2 Proposal Strategy Based on Game Tree Search

As detailed in Sections 3.2.3 and 3.2.4, we incorporated the above predictors into the EXPECTI part of an Expectimax game tree search, and used the search algorithm to choose chipsets to offer to the opponent. The bottom row of Table 5.5 shows the effect of adding this offer strategy to our agent's behavior. When we replace the offer selection strategy from PERB with one based on our implementation of Expectimax, AltOffersLearning unfortunately suffers a further loss in performance.

5.4 Summary of Results

Figure 5.1 shows a comparison of average scores for the two versions of PERB and the three versions of AltOffersLearning that we tested. Note that in order to show correct proportions, the lower bound is at -80, which is the minimum possible score on the DD board.

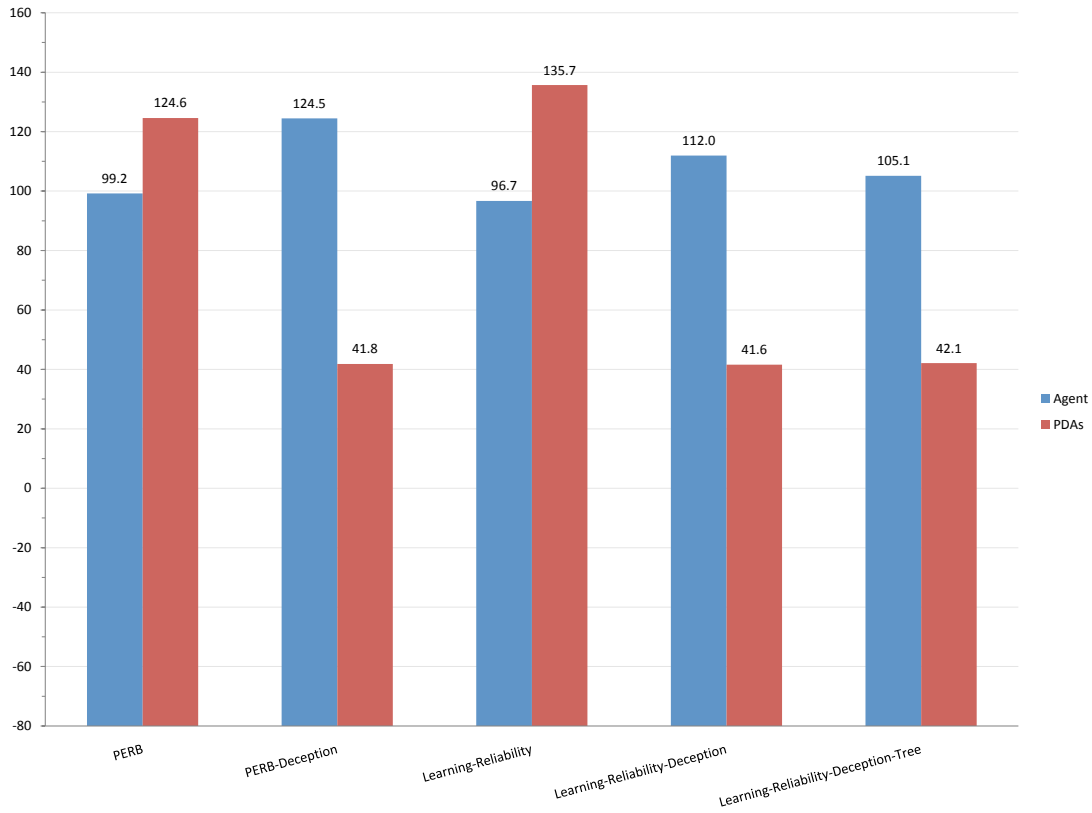


Figure 5.1: Summary of average scores for agent versions

Chapter 6

Conclusion

As shown in Figure 5.1, we actually achieve a small net improvement over the performance of the baseline PERB agent. However, this improvement is entirely attributable to a single modification — the addition of guaranteed deception to the agent's transfer behavior. The more theoretically interesting modifications — the reliability model and the game tree search — actually caused declines in agent performance.

This does not, however, imply that these components cannot be useful in an agent implementation. Our reliability model is a very reasonable way of expressing an agent's reliability for any *single* round, since it considers ratios of changes in *utility* (as measured by score), rather than simply ratios of changes in chip counts. However, as our results show, the use of the reliability model as the sole determining factor for chip transfers is not effective; choosing transfers reactively, in a "tit-for-tat" manner, is inadequate, as this involves no reasoning about the future.

From the results in Section 5.3.1, as well as from the results of Carr et al., we find that statistical classifiers can be surprisingly effective in predicting certain aspects of opponents' behavior. It is likely possible to improve accuracy further by including additional attributes from the game state. In order for predictors to become more useful in an agent strategy, their role and number should be expanded.

The ineffectiveness of our Expectimax search most likely stemmed from its very limited scope. Although it is necessary to constrain tree depth in a scenario with as large a branching factor as AltOffers, our restriction of the search to a single entry point (makeGameTree-

OnMyOffer) and a single game round meant that Expectimax's utility calculations were based on very uneducated guesses. At a minimum, the tree search should be expanded to include rounds where the opponent is the first proposer or rejects the player's proposal and makes a counter-offer, and should recurse down to one or two subsequent rounds. At cutoff points, it may be fruitful to try to create and use new predictors to estimate utility values, rather than simply using the default best-use scores at those leaf nodes.

Bibliography

R. Carr, P. Roos, and B. Wilson. Opponent modeling in Colored Trails: Meta-proposals and behavior predictors in a feasible game tree search. Unpublished., 2009.

M. Chalamish, D. Sarne, and S. Kraus. Programming agents as a means of capturing self-strategy. *AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1161–1168, 2008.

B. J. Grosz, S. Kraus, S. Talman, B. Stossel, and M. Havlin. The influence of social dependencies on decision-making: Initial investigations with a new game. *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 782–789, 2004.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

S. Kraus. General specification for agent that is medium reliability and medium cooperativeness. Email correspondence, September 2009.

E. Manisterski, R. Lin, and S. Kraus. Understanding how people design trading agents over time. *AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1593–1596, 2008.

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

S. Talman, M. Hadad, Y. Gal, and S. Kraus. Adapting to agents' personalities in negotiation. *AAMAS '05: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 383–389, 2005.