University of Alberta

Symmetries and Search in Trick-Taking Card Games

by

Timothy Michael Furtak

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Fall 2013
Edmonton, Alberta

Abstract

In this dissertation we consider the problem of cardplay move evaluation in the context of trick-taking card games, specifically the German card game Skat, and to a lesser extent Contract Bridge. To this end we construct symmetry-based search extensions for efficiently solving perfect information game states, for use in a Monte Carlo-based high-performance computer Skat player. We extend these symmetries to construct precomputed endgame lookup tables, reducing search time substantially. Finally, we show that recursive rollout-based move selection techniques can achieve superior performance in Skat and a broad class of parameterized games, with respect to both tournament performance and game-theoretic exploitability.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"There's a Bene Gesserit saying," she said.
"You have sayings for everything!" he protested.

Dune
FRANK HERBERT

1.1 Motivation

There is a rich history of computing science research being motivated by games and puzzles. These domains provide well-defined problems where success is easy to quantify. Domain-specific search and optimization techniques have gone on to spur improvements in far-removed fields and applications. Yet even ignoring these spin-off successes, games themselves continue to capture the interest of society. Classic games have undergone a modern resurgence in both quality and popularity. Computer games are a billion dollar industry, and are even a popular spectator activity — both competitive video gaming and self-published casual gameplay.

Whereas classical computing science testbeds have tended to be perfect information games, there has been growing interest in imperfect information domains. These domains contain the types of problems that are frequently encountered in the real world, as rarely do agents have access to all of the information they might wish.

The domain that we have chosen to study is the trick-taking German card game of Skat. It is rather ironic that an imperfect information domain has provided such a rich source of ideas for perfect information search techniques. We will restrict ourselves to discussing perfect information (open-handed) cardplay positions until Chapter 9, and to a lesser extent Chapter 6.

1.2 Organization and Contributions

This dissertation is a combination of previously published work, interspersed with additional details and new results. Although sections have been rewritten to form a more coherent narrative, it is inevitable that large blocks of text have survived intact.

The first portion of this work is concerned with improving the speed of open-handed cardplay evaluations. These evaluations are central to current state of the art methods for constructing computer trick taking card game agents. Taken in combination, the improvements given in this work resulted in search speeds of 45-, 58-, and 22-times[1] the previous state of the art for Skat suit games, grand games, and null games respectively. These speed increases made possible the technique used in Chapter 9 — using a full, high-performance, Skat-playing program as a recursive playout module for Monte Carlo move selection. The resulting agent substantially outperforms the previous state of the art in head-to-head play.

Chapter 2 describes related work and provides an introduction to our Skat-playing program, Kermit. Chapter 3 discusses the problem of constructing an adversarial search minimum proof tree. This includes a theoretical analysis, and extends into a general move-ordering heuristic for α-β search. Chapter 4 provides a novel and meaningful analysis of Ginsberg's partition search applied to trick-taking card games. We present an intuitive description of how sets of equivalent states can be constructed over the course of an α-β search.

Chapter 5 discusses static analysis methods for quickly bounding the value of an open-handed cardplay position in Skat and Bridge. Chapter 6 describes "paranoid" α-β search, with applications for Skat bidding, and related algorithms for move selection. Chapter 7 introduces a method for bounding the difference in α-β score between two cardplay states, resulting in significant performance gains in solving open-handed Skat positions. Chapter 8 extends this state-similarity method to constructing precomputed lookup tables. These lookup tables provide further substantial gains for Skat trump games, and virtually eliminate search in Skat null games.

In Chapter 9 we discuss new move-selection algorithms that better consider the imperfect information aspects of Skat games. We provide a detailed analysis of this approach using synthetic game trees. This approach leverages our work on fast open-handed cardplay, and the resulting agent is the new state of the art for Skat cardplay, decisively beating competing approaches.

Chapter 10 concludes the thesis and suggests lines of future work.

1.3 Publications

Chapter 3 is based on the paper "Minimum Proof Graphs and Fastest-Cut-First Search Heuristics" [14], presented at IJCAI 2009, in Pasadena, California, USA.

[1] Measured using random, 30-card start of game positions.

Chapters 7 and 8 are based on the paper "Using Payoff-Similarity to Speed Up Search" [15], presented at IJCAI 2011, in Barcelona, Spain.

Chapter 9 is based on the paper "Recursive Monte Carlo Search for Imperfect Information Games", which has been accepted to the IEEE Conference on Computational Intelligence in Games, in Niagara Falls, Canada.

Contributions from two other papers are distributed throughout this thesis. The first paper is "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search", [29] published in AAAI 2010, in Atlanta, Georgia, USA. The second is "Improving State Evaluation, Inference, and Search in Trick-Based Card Games", [4], published in IJCAI 2009, in Pasadena, California, USA.

Chapter 2

Background

> "When strangers meet, great allowances should be made for differences of custom and training."
>
> *Dune*
> FRANK HERBERT

2.1 Preliminaries

A large portion of this thesis is devoted towards efficiently computing the utility (a.k.a. the *value*) of a given game state. This value is defined as the best score achievable by all players, assuming optimal play by all parties. We tend to restrict ourselves to the simplified case of finite, two-player, perfect information, zero-sum games. A game is finite if it does not support arbitrarily long move sequences (i.e., no cycles). A game is perfect information if all players know the true state of the world at all times (no player has private information), and the outcomes of all possible future chance events are known ahead of time. A game is zero-sum if the utilities for each player always sum to zero (or more generally a constant), such that a player can only gain at the expense of his or her opponent. In two-player games the score is usually only stated with respect to one player (since the other player receives its negation). This player (usually the player currently to move) is then termed the *maximizer*, as he tries to maximize the value of the game, whereas his opponent (the *minimizer*) attempts to minimize it.

A game state may also be referred to as a *position*, a *node* (say, in the case of alpha-beta search), or simply a *state*. The *game tree* is the graph formed by equating game states with nodes, and actions with edges. A directed edge joins two nodes if there is a corresponding legal action such that the *parent* state transitions to the *child* state. Note that although the term "game tree" is traditionally used, the actual structure is that of a directed acyclic graph, a DAG. In particular, this means that a state may be reached via multiple distinct paths (sequences of actions). In game tree search nomenclature this is a *transposition*. The *minimax* value of a state is equal to its utility for the player to move. This value may be simply (but inefficiently) computed by considering, at each state, which

move results in the best score for the player to move. The value of a terminal node is defined by the game itself.

Throughout this work we will relate our results to Skat, the popular trick-taking German card game. Readers who do not have at least a passing familiarity with the rules of Skat are encouraged to consult Appendix A, or a similar resource. For the casual reader it may be sufficient to consider Skat as a standard 3-player trick-taking card game, except that the cards have point values. Note that while the card points obtained by each team affect the final score, they do not directly correspond to the game value itself. The cardplay phase of Skat is a team game, in which a soloist (game declarer) competes against two defenders. In addition to Skat, but to a lesser extent, we will also refer to the game of Contract Bridge, although we will only consider the cardplay phase — the bidding process is not addressed.

In order to discuss the algorithms and challenges relating to imperfect information search we will need to make reference to *information sets*. An information set contains one or more game states which are indistinguishable to the player that is to act in that information set. This is due to the player not having access to a particular piece of information — for example we may know which cards we are holding, but not those of the other players. Because the states are indistinguishable, the player's policy in all of those states must be the same. We need not, however, assume a uniform distribution over the states in an information set. We may be able to bias these distributions based on knowledge of the underlying game structure (say, dice rolls or card shuffling), or based on our beliefs of the other players' strategies. A player's *strategy* is a policy which they follow in every position where they are to act. A *pure* strategy involves taking a deterministic action in every information set, whereas a *mixed* strategy is a distribution over pure strategies, with probabilities summing to 1.

2.2 Perfect Information Monte Carlo

One of the best-known and widely used methods for dealing with imperfect information games, especially card games, is the Perfect Information Monte Carlo search algorithm. PIMC is a sampling algorithm that constructs a number of hypothetical worlds which are consistent with previous observations. In each of these hypothetical worlds, the value of taking each action is computed under the assumption that all players have perfect information. In the end, the move that has the highest value, averaged over all hypothetical worlds, is taken. These hypothetical worlds may be weighted according to some likelihood function (i.e., inference) — say, based on an a priori belief over the hidden state variables, or over the other players' policies/strategies. This approach has also been de-

scribed [36] as "averaging over clairvoyancy". PIMC search forms the core of Kermit, our computer Skat player.

This process tends to overcome the theoretical limitations of treating a game as its perfect information variant in two ways. The first is by exploiting the ability of perfect information search to find highly tactical lines of play. If there is only one way to win in a position, PIMC will find it, given enough time. The second is by bludgeoning the problem into submission by examining many worlds — decades of research has made perfect information search rather fast. If, in the real game, information is revealed at a sufficient rate, then the simplification of assuming all players know everything may not be as prohibitive as it first appears. Indeed we have demonstrated certain empirical game tree properties which correspond to PIMC being effective [29], for both synthetic and real games (Skat, Hearts).

Applying PIMC to card games (specifically contract Bridge) was first proposed by Levy [27], and was successfully implemented by Ginsberg [16]. PIMC has also been applied to Skat by Kupferschmid and Helmert [26], as well as to the games of Hearts and Spades by Sturtevant and White [42].

Drawbacks of PIMC

Despite the successes of PIMC in high-performance game-playing systems, it suffers from some theoretical drawbacks which, as formalized by Frank and Basin [11], may result in two different types of errors. The first of these errors is termed *strategy fusion*, and arises from PIMC incorrectly believing that it can select different (and incompatible) strategies across the hypothetical worlds. That is, PIMC breaks the assumption that we must act identically over all states in an information set. This may result in overly optimistic and/or pessimistic actions for one or more players. Thus, a perfect information evaluation may be arbitrarily inaccurate, either higher or lower, compared to the actual game theoretic value. The resulting PIMC-produced strategy may then be arbitrarily bad in general.

The second type of error is *non-locality*. In perfect information games the value of a position is determined entirely by its subtree (the minimax value), but this is not the case for imperfect information games. In imperfect information games the actions of our opponent will, in general, be based on private information — information our opponent possesses but we do not. By observing the results[1] of these opponent actions we may make inferences about the information that led to them.

[1] If we cannot observe opponent actions directly, we may be able to observe some consequences of those actions.

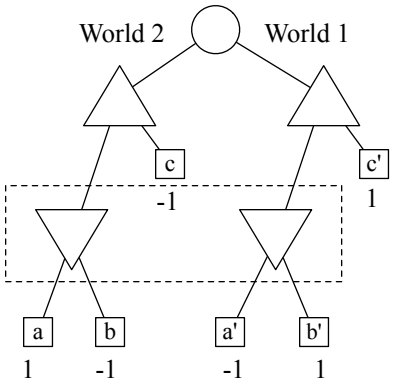

Figure 2.1: Example of non-locality. Player nodes are depicted as triangles, with maximizer nodes pointing up and minimizer nodes pointing down. Terminal nodes are depicted as squares.

Consider the example shown in Figure 2.1. At the root of the tree a chance node generates one of two worlds. The maximizing player is able to distinguish between these two events, but the minimizing player cannot (the minimizer nodes are in the same information set, represented by a dotted rectangle). In world 1 the maximizer can win immediately by moving to the right. Thus if the minimizer is ever to act (i.e. the maximizer moved to the left), the minimizer may infer that he is in world 2, and so always select the correct action. Compare this to a PIMC agent which uniformly samples consistent worlds. Such a PIMC agent would consider both worlds equally likely, causing the minimizer to select each action with equal probability. The non-locality phenomenon can thus create dependencies between potentially distant nodes in the tree. In this case, the correct choice of action depends on the value of c'. If c' had value -1 then no meaningful inference could be made.

There are a number of approaches which deal with imperfect information in a somewhat more principled manner, compared to PIMC, although we postpone their discussion until Section 2.5.

2.3 Computer Bridge

One of the most well-known success stories for computer play in trick-taking card games is that of Ginsberg's GIB player for contract Bridge [16]. GIB was introduced in 1998, and was the first full-fledged Bridge program to make use of PIMC. By 2001, GIB was claimed to be the strongest computer Bridge player in the world, with playing strength roughly equivalent to that of expert humans.

The core of GIB's strength is its ability to solve the perfect information (so-called "double dummy") version of a Bridge deal. Open-handed position analysis in Bridge is sometimes called

double-dummy search (DDS). With start-of-game Bridge positions consisting of 52-cards, existing search techniques (α-β search) were unable to solve sufficiently many positions (under reasonable time constraints) for PIMC to be effective. Consequently, Ginsberg developed *partition search*, which reduced the number of nodes expanded by "an order of magnitude or more", as well as developing *lattice search*, in collaboration with Jaffray, for declarer card play under uncertainty. A detailed discussion of partition search is presented in Chapter 4.

Lattice Search

In [17] Ginsberg proves necessary and sufficient conditions for applying α-β pruning in the case where terminal values belong to a distributive lattice, rather than a total ordering (as is usually the case, e.g., integers or real numbers). That is, the values satisfy only a partial ordering. In the context of Bridge, this technique is applied to compute imperfect information declarer strategies, assuming the defenders have perfect information — the so-called "best defense" model. The partial ordering used corresponds to identifying consistent worlds in which the declarer can make her contract. This allows Ginsberg's program to essentially search over the space of possible imperfect information strategies, explicitly considering that it is constrained to act consistently within an information set. This method is only used by GIB when the declarer is to act.

Bidding in GIB

GIB's bidding is based on a database of hand-crafted rules capable of suggesting potential bids for a given situation. Where the suggestions are "close", a Monte Carlo simulation is used to generate consistent worlds and a hypothetical bidding sequence(s) for each option. Such a bidding sequence is known as a *Borel simulation* within the Bridge community. The resulting contract is then evaluated using a perfect information search, and the best bidding option (from a PIMC perspective) is taken.

Bidding in Bridge is significantly more constrained than in Skat, however. In Bridge, all bids have a well-defined meaning, which partners use to communicate information about their hands. The meaning of these bids must also be furnished to the opponents upon request (and must thus be describable in human terms).

2.4 Computer Skat

Skat Bidding

Unlike Bridge, the bidding in Skat has essentially no restrictions. Players will often stop at a multiplier of the suit they would have liked to play (say, at 33 to indicate that they have many spades), but this tends to be a consequence of simply wanting to win the bid and not being able to go any higher. In certain situations, e.g., where one player believes that she cannot possibly win the bid (based on the bidding of others, and her own hand) she may bid a multiple of a suit in which she holds an ace, to indicate this to her future partner.

In Skat the choice of which game type to declare as soloist is an important one. Choosing the wrong trump suit or the wrong discard can lead to losing a game with 60 points (or worse). But there is more to consider than simply which suit the soloist has the most of. Some things are obvious: having a singleton ten is bad — all it takes is one defender playing an ace for the soloist to lose 21 or more points. Putting twenty points in the skat means that the soloist might only need to win 2 or 3 tricks to win the game — a good idea if the defenders are strong in trump. Having aces in the non-trump suits is good, because it lets the soloist get back into the lead without spending trump, but long suits are also likely to get trumped by a defender. Moreover, hearing the bidding of the other players all plays a role in what one believes is in their hand, and what might be in the skat.

Keller and Kupferschmid [22] describe a Skat bidding algorithm that uses k-nearest neighbour classification to estimate the number of card points that a particular hand configuration will result in. Specifically, they construct a knowledge base of 10,000 entries labeled with the open-handed position value. They define a similarity metric based on hand features such as the number of jacks, the number of card points, etc. They empirically optimize the value of k to best perform on a training set, where the error is the difference between the predicted value and the perfect information value. They suggest how this method can be employed to optimize over discards, by considering all $\binom{12}{2} = 66$ possible choices, and taking the best as suggested by their predictor. Notably, their results on a test set are reported in terms of the difference between their predictions and the open-handed values, rather than by playing out actual games. While their stated results seem reasonable, it is unclear exactly how well they translate to real games.

The bidding engine of our program, Kermit, as described by Buro et al. [4], uses a so-called "10+2" evaluation. This evaluations considers only the 10 cards in hand and the 2 in the skat, and is trained from millions of human games. The evaluation is a table-based lookup using features

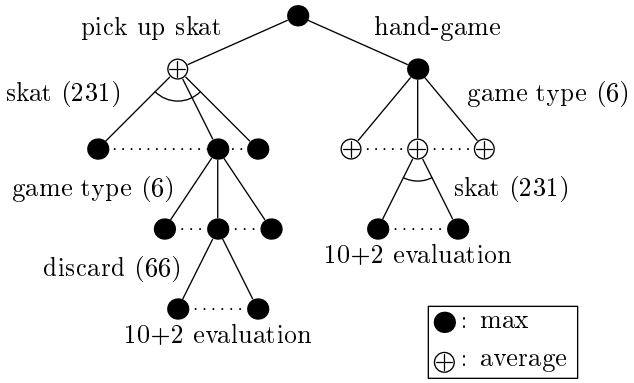

Figure 2.2: Diagram of Kermit's bidding evaluation process.

which are similar to those of Keller and Kupferschmid, but its output is not the predicted number of card points, rather the winning probability. Specifically it is the winning probability based on the results of actual human cardplay. Exhaustive sampling over the possible skat cards ($\binom{22}{2} = 231$ possibilities) is used to estimate the expected winning probability at each bid level. After picking up the skat, the 10+2 evaluation is then used for discarding and selecting a game type. This process is parameterized by a single winning probability threshold — if the expected winning probability at the root, after maximizing over game declarations and discards, is greater than the given threshold, then we bid/accept. A high threshold thus results in a conservative player that only bids on very solid games, whereas a low threshold results in riskier bidding. Empirically a highly robust threshold of around 60% was chosen. A diagram of the bidding evaluation process is shown in Figure 2.2.

Skat Search

Although there is not a large body of research on Skat, some Skat-specific enhancements to α-β (with broader generalizations) have been noted outside of our own work. Notably, Kupferschmid and Helmert [26] introduce *adversarial heuristics* and *quasi-symmetry*, which is related to our work in Chapters 5 and 7.

2.5 Non-PIMC Approaches

We would be remiss not to address several non-PIMC-based algorithms for playing imperfect information games. The following algorithms have varying degrees of justification and their own set of pros and cons which we will discuss in relation to our chosen problem domain.

It is worth noting that zero-sum 1 vs. 2 team games with imperfect information, as shown by von Stengel and Koller [44], are solvable by a quadratic programming approach.[2] The resulting solution is termed a *team-maxmin equilibrium*. Such an equilibrium is, in general, distinct from the equilibria arising from the 2-player version of the same game (where the team players are allowed to coordinate their actions). The resulting gap in payoff (the team-maxmin value is generally less than the 2-player value) arises from the team being unable to share information perfectly.

The "Best Defense" Model

As part of Frank and Basin's critique [11] of PIMC search (referred to as repeated minimaxing — α-β search repeated over multiple worlds) they introduce the best defense model in the context of Bridge. As the name suggests, the defenders are able to mount the best-possible defense — they play with perfect information about the true state of the world and know the declarer's (pure) strategy. Moreover, the declarer has only imperfect information about the true world. This is a conservative model that corresponds to the expert analyses given for Bridge hands. Termed *exhaustive strategy minimisation*, the algorithm given by Frank and Basin for solving the best defense variant is doubly exponential in the number of maximizer (declarer) levels in a search tree, and is only intended for "the smallest of game trees".

In 2000, Frank, Basin, and Bundy [12] produced Finesse, a program capable of solving single-suit Bridge problems under the best defense model. Notably, Finesse searches on the level of *tactics*, which enumerate the possible ways for the declarer to play a trick. This search space is significantly smaller than the full game, which allowed for the stated results, although the ability to solve larger Bridge problems has not yet been shown.

Counterfactual Regret

Unlike trick-taking card games, poker (specifically Texas Hold'em) hinges almost entirely on the assumption that neither player knows the true state of the world. Applying PIMC in such a scenario (absent any opponent modelling) would result in bidding the expected value of one's hand — which would quickly be exploited by any observant opponents. Recently computer poker has achieved much success due to the use of *counterfactual regret minimization*, or CFR, as described by Zinkevich et al. [46]. CFR is an iterative algorithm that updates the *regret* of taking each action in a given

[2] Naturally this assumes that the size of the game is sufficiently small, which is not so in our case.

information set, relative to the best move, given the current policies of all players. Given sufficient time this algorithm will converge to an ϵ-Nash equilibrium.

CFR is an offline algorithm requires keeping track of information for each information set. For a large game such as poker, abstraction is necessary to make the problem tractable. The resulting equilibrium strategy is simply a giant lookup table containing the move probabilities for each information set, and is thus move selection is very fast.

Unlike Skat, the actions in poker are independent of the cards that one is holding. Poker abstractions thus tend to have some measure of *hand strength*, but they do not need to capture any tactical cardplay. While it may be possible to define an abstract Skat game which abstracts states and actions in a meaningful way, it is not clear how this would be done.

UCT: Upper Confidence Bounds applied to Trees

The UCT algorithm, Upper Confidence Bounds applied to Trees, by Kocsis and Szepesvari [24] is a search algorithm which grows the search tree in-memory in a manner which seeks to exploit the best-known line of play for both players, while exploring infinitely often in the limit. UCT operates in the manner of a Monte Carlo rollout: each iteration starts at the root and proceeds by following the UCT exploration policy while within the tree. When a leaf node is reached, a rollout module is used to play the remainder of the game — the terminal value of the rollout is then used as an estimate for the value of the leaf. The search tree is also grown by one node in the direction of the rollout.

Since the values backed up by the rollouts are used as estimates within the tree, the rollout policy itself is important in determining the effectiveness of UCT. Depending on the domain, the rollout module could be a rule-based system or even a random player. Note that stronger policies do not always result in stronger move choices at the root.

UCT has been instrumental in improving the strength of computer Go programs, and was used in the winning entry of the 2007 General Game Playing competition [10]. Schäfer [38] applied the UCT algorithm to Skat, such that nodes in the UCT tree are not states, but information sets. The rollout policy used is the rule-based open source Skat program XSkat [45]. XSkat is a rather weak program, but the resulting UCT player was roughly equivalent to a PIMC-based player (although not *our* PIMC player). This is discussed further in Chapter 9.

Chapter 3

Minimum Proof Graphs and Fastest-Cut-First Search Heuristics

"And remember that growth itself can
produce unfavorable conditions
unless treated with extreme care."

Dune
FRANK HERBERT

3.1 Introduction

Alpha-beta (α-β) [23] search is the classic heuristic search algorithm for two-player games with perfect information. In past decades AI researchers have found numerous enhancements that, for instance, enable today's chess programs, running on ordinary desktop computers, to defeat human world champions. Algorithmic improvements to α-β generally take one of two complementary forms. The first is to improve the heuristic evaluation of interior nodes — if search cannot efficiently compute a state's minimax value by propagating (exact) terminal node values then an estimate must be used. Better node evaluations can lead to more accurate values at the root of the tree, as well as potentially causing more α-β cutoffs, due to computing exact and/or heuristic bounds for a node's value. The second class of improvements involves decreasing the number of search nodes that need to be expanded. This can be accomplished, for example, through better move ordering, or by more efficiently reusing information from previously searched portions of the game tree.

Increasing the speed of α-β search is important because it allows search to explore deeper in the game tree, where leaf node evaluations tend to be more correct. In this chapter we focus on the *Minimum Proof Graph* problem that asks, given

- a directed acyclic graph (DAG) G corresponding to the states in a 2-player zero-sum game,
- a player labeling for each state denoting which player is to act, and
- a score function for each terminal state,

what is the minimum number of vertices needed in a subgraph $H \subseteq G$ in order to prove one of the following properties:

1. Can the first player achieve a score of at least t, for some integer t? This assumes that the first player only seeks to achieve a score $\geq t$, and the second player tries to prevent this.

2. What is the minimax value of the game? This assumes both players seek to maximize their numerical score.

Knuth [23] answered question 2 in the context of homogeneous trees with constant branching factor b and depth d: searching $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ leaves is necessary and sufficient for establishing the minimax value of such trees. However, as Plaat [34] notes, trees are rarely homogeneous in practice, and in fact, inputs usually are not even trees, but DAGs. For the purpose of judging how close programs for popular games — such as chess, checkers, and Othello — come to searching minimum proof graphs, they introduce the concepts of Left-First-, Real-, and Approximate-Real-Minimal-Graphs, and estimate upper bounds on minimal proof graphs from game data.

In this chapter we approach the problem more formally by first proving some fundamental properties of proof graphs. We then proceed to show that computing minimum proof graphs for DAGs is NP-hard, and that minimum proof trees for trees can be constructed in linear time. Finally we describe fastest-cut-first search heuristics, which we evaluate using a novel synthetic game tree model.

3.2 Proof Graph Definitions

We shall assume throughout this chapter that $G = (V, E)$ is a DAG with vertex set $V = \{v_0, v_1, \ldots, v_{n-1}\}$ corresponding to states in the original game, and directed edge set E corresponding to legal moves from each state. Let p define the player function which maps vertices in V to $\{\text{white,black}\}$, and let f be the score function which maps terminal vertices to an integer.

Corresponding to the two properties being proven, we define two classes of decision problems. The first class is a tuple $\langle G, p, f, s, t \rangle$ which asks whether a proof graph exists using at most s nodes to prove that the first player can achieve a score of at least t. The second class is a tuple $\langle G, p, f, s \rangle$ which asks whether a proof graph exists using at most s nodes to prove the minimax value of the game. To clarify which type of proof graph we are referring to, we shall use the terms *target proof graph* and *minimax proof graph* respectively.

In both cases we require that accepted instances of the decision problems satisfy the condition that only one vertex of G, which we shall call the root, has in-degree 0. Without loss of generality we

shall assume that v_0 is the root and that $p(v_0) =$ white, such that white is the first player, and that the player to move alternates. We denote the children of a vertex using $c(v_i) := \{v_j : (v_i, v_j) \in E\}$.

Note that the minimax value may be computed recursively in time linear in the size of G. As such, we extend f to be defined over all vertices, rather than only the leaves, such that f is now the negamax value of a vertex — i.e., the score is given with respect to the player to move at that vertex. Specifically, if $c(v_i) \neq \emptyset$ then $f(v_i) = \max_{v_j \in c(v_i)} -f(v_j)$.

Target Proof Graphs

A **target proof graph** is a proof graph that proves a lower bound on the minimax value at the root. A subgraph $H \subseteq G$ is a valid target proof graph if and only if all the following hold:

- $v_0 \in H$
- $\big(v_i \in H \text{ and } p(v_i) = p(v_0) \text{ and } c(v_i) \neq \emptyset\big) \Rightarrow \exists v_j \in c(v_i) \text{ such that } v_j \in H$
- $\big(v_i \in H \text{ and } p(v_i) \neq p(v_0)\big) \Rightarrow c(v_i) \subset H$

These conditions correspond to H being an AND/OR tree, where the root is an OR node. For a given target proof graph H, and $v_i \in H$, let

$$f^{(H)}(v_i) = \begin{cases} f(v_i) & : \text{if } v_i \text{ is a leaf,} \\ \max\limits_{v_j \in c(v_i) \cap H} -f^{(H)}(v_j) & : \text{otherwise.} \end{cases}$$

This function specifies the proven lower bound on the negamax value of each vertex in H.

Definition 3.2.1. **MinProofGraph-1** is a set of succinct encodings of $\langle G, p, f, s, t \rangle$ where there exists a subgraph $H \subseteq G$ corresponding to a valid target proof graph such that $f^{(H)}(v_0) \geq t$, and such that $|H| \leq s$.

Minimax Proof Graphs

The evaluation of a minimax proof graph (in the α-β sense) is slightly more involved than for a target proof graph as it depends on the concept of α-β pruning. Specifically, the order in which a vertex's children are evaluated can affect the size of their proof graphs. This is due to more effective search cutoffs as the range of potential values for a node (and thus its ancestors) is reduced. It should be noted that verifying a minimax proof graph can be done in polynomial time even without explicitly giving the order in which to evaluate a node's children. This follows from results which will be

presented in Section 3.3, that an optimal traversal will only use at most three possible α-β pairs when examining vertices.

For concreteness we will now define an indicator function to specify whether, for a vertex in a given minimax proof graph, the proof is accepted. First, for a sequence, σ, of child vertices, define α_j to be the highest possible alpha bound resulting from examining the first j nodes in σ, given the (potential) proof graph $H \subseteq G$. Now let $\mu^{(H)}_{(\alpha,\beta)}(v_i) = \text{true}$, if and only if:

1. $v_i \in H$ is a leaf, or
2. $\exists \sigma$, a permutation of $c(v_i)$, such that $\underset{1 \le j \le |c(v_i)|}{\forall}$ either:
   a) $\sigma_j \in H, \quad \mu^{(H)}_{(-\beta,-\alpha_j)}(\sigma_j) = \text{true}$ or
   b) $\exists k < j$ such that $\beta \le \alpha_k$.

These conditions correspond to having an ordering to explore a node's children such that the α-β bounds at any point are sufficient to prove the next child's α-β value using the vertices in H, or for the bounds to short-circuit further child evaluations by causing a cutoff, which occurs when $\beta \le \alpha$, indicating that the values of a parent's remaining children will provably not affect the game's minimax value. A subgraph $H \subseteq G$ is a valid minimax proof graph if and only if $v_0 \in H$ and $\mu^{(H)}_{(-\infty,\infty)}(v_0) = \text{true}$.

Note that our indicator function is not concerned with whether H is sufficient to correctly compute the exact minimax value at each node, but whether an α-β search with the given incoming bounds will return a correct result, namely fail-low ($f(v_i) \le \alpha$), fail-high ($f(v_i) \ge \beta$), or the exact value ($\alpha < f(v_i) < \beta$). The correctness of the indicator function may be more easily seen by considering that α-β search will evaluate all children unless a beta cut is triggered. Proceeding by induction on the height of the search tree gives the desired result.

Definition 3.2.2. **MinProofGraph-2** is a set of succinct encodings of $\langle G, p, f, s \rangle$ where there exists a subgraph $H \subseteq G$ corresponding to a valid minimax proof graph such that $|H| \le s$.

3.3 Minimum Minimax Proof Graph Properties

We will now proceed to make some observations which will simplify the computation of a minimax proof graph, assuming that we know the true values of each vertex. Recall that it is efficient to

compute these true values. For convenience we introduce the indicator function

$$I(v_i, v_j) := \begin{cases} +1 & : \text{if } p(v_i) = p(v_j), \\ -1 & : \text{otherwise.} \end{cases}$$

We will use $f_u(v) := I(u,v) \cdot f(v)$ to refer to the value of a vertex v with respect to some other vertex u, usually v's parent, although sometimes the root node.

Theorem 3.3.1. *For an alpha-beta examination of a vertex v with $\alpha < \beta \leq f(v)$ or $f(v) \leq \alpha' < \beta'$, the size of a minimum minimax proof tree rooted at v is independent of α and β'.*

Proof. We proceed by induction on the height of v, the maximum path length starting from v. Clearly the observation holds for a height of 0, where v is a leaf and has no children. Now assume the observation holds for all heights less than or equal to some n, and that v has height $n+1$ (clearly all of v's children have height $\leq n$).

First consider the case where $\alpha < \beta \leq f(v)$. The examination of v must end with a cutoff where a child is proven to have a value of at least β. Call such a child *cutting*. Note that examining a child with value $< \beta$ (so as to improve the alpha bound) cannot reduce the effort required to prove a cutting child w. To see this, observe that $\alpha < \beta \leq f_v(w) \leq f(v)$ and so w will be examined with bounds $\langle -\beta, -\tilde{\alpha} \rangle$ satisfying $f(w) \leq -\beta < -\tilde{\alpha}$, where $\tilde{\alpha}$ is some (potentially) improved alpha bound. By the induction hypothesis, the minimum proof graph size for w does not depend on $\tilde{\alpha}$. Therefore, because improving the alpha bound cannot help, the minimum proof graph for v will consist of examining only one cutting node, and so also does not depend on α.

Now consider the case where $f(v) \leq \alpha' < \beta'$. Since no child can have a value greater than $f(v)$, the alpha bound can never be improved, and all children must be examined. The argument that the induction property holds for $n+1$ is the same as in the first case, except now $\tilde{\alpha} = \alpha$.

Thus, by induction, the observation holds for all n.

□

Theorem 3.3.2. *For an alpha-beta examination of a vertex v the following properties hold:*

(i) $\alpha \leq f(v) < \beta \Rightarrow$ *the minimum proof graph size does not depend on β.*

(ii) $\alpha < f(v) \leq \beta \Rightarrow$ *the minimum proof graph size does not depend on α.*

(iii) $\alpha \leq f(v) \leq \beta$ *and* $\alpha < \beta \Rightarrow \exists\, w \in c(v)$ *to examine first which minimizes the proof graph size and* $f_v(w) = f(v)$.

Proof. We again proceed by induction on the height of v, for which the listed properties clearly hold for a height of 0 — a leaf. Now assume the properties holds for all heights less than or equal to some n, and that v has height $n+1$.

Property (i): Since $f(v) < \beta$ all children of v must be examined. Note that α can never become larger than $f(v)$ so the initial constraint that $\alpha \leq f(v) < \beta$ holds at each step. For any child $w \in c(v)$ we have that $f_v(w) \leq f(v)$. Now, either $f_v(w) \leq \alpha < \beta$ or $\alpha < f_v(w) < \beta$. In the first case the minimum proof graph (MPG) size is independent of β by Theorem 3.3.1. In the second case, part (ii) of the induction hypothesis gives the desired result.

Property (ii): If we can show that the MPG size to examine any child w with $f_v(w) = f(v)$ does not depend on the initial value of α (as long as $\alpha < f(v)$) then we are done, since all other nodes can be examined afterwards, and the optimal α bound from searching w can only decrease the size of those nodes' MPG. Let $w \in c(v)$ be such that $f_v(w) = f(v)$ and $\alpha < f(v)$, which implies $\alpha < f_v(w) \leq \beta$. By part (i) of the induction hypothesis, the MPG size of w does not depend on α.

Property (iii): The preconditions of property (iii) match those of (i) or (ii) (or both). In the case of (i), as noted in its proof, the child ordering does not affect the MPG size, so simply take w to have $f_v(w) = f(v)$, as at least one such w exists. In the case of (ii) it is never suboptimal to examine such a w first, and as at least one such child must minimize the proof graph, take w to be that one. Hence (iii) holds. Thus, by induction, the properties hold for all n.

□

Based on Theorem 3.3.2 we may assume that the first path explored by α-β from the root to a leaf follows an optimal line of play, the *principal variation* (PV), such that all nodes w along the path have value $f_{v_0}(w) = f(v_0)$. Clearly this initial path will be explored using bounds $\langle \alpha, \beta \rangle = \langle \infty, -\infty \rangle$. There may be many optimal lines of play, but for clarity we shall assume that the PV refers to one particular such path in an MPG.

The following theorem essentially characterizes the types of bounds used when solving *cut* nodes and *all* nodes. That is, nodes where only one child must be examined and nodes where all children must be examined.

Theorem 3.3.3. *For an alpha-beta examination of a DAG G that minimizes the proof graph size, it will be the case that for each $v \in G$ not on the PV, with initial bounds α and β:*

(i) $\langle \alpha, \beta \rangle \in \{\langle \pm f(v_0), \infty \rangle, \langle -\infty, \pm f(v_0) \rangle\}$

(ii) $\langle \alpha, \beta \rangle = \langle -\infty, \quad f(v_0) \rangle \quad \Rightarrow \quad f(v) \geq \quad f(v_0)$ *(Player 1 cut node)*

(iii) $\langle \alpha, \beta \rangle = \langle -\infty, -f(v_0) \rangle \quad \Rightarrow \quad f(v) \geq -f(v_0)$ *(Player 2 cut node)*

(iv) $\langle \alpha, \beta \rangle = \langle \quad f(v_0), \quad \infty \rangle \quad \Rightarrow \quad f(v) \leq \quad f(v_0)$ *(Player 1 all node)*

(v) $\langle \alpha, \beta \rangle = \langle -f(v_0), \quad \infty \rangle \quad \Rightarrow \quad f(v) \leq -f(v_0)$ *(Player 2 all node)*

Proof. We will proceed by induction on the maximum distance, n, from the principal variation (PV). For a given vertex v, if v is on the PV then define its maximum distance to be 0; otherwise define the maximum distance to be the length of the longest path in G from some vertex on the PV to v.

If $n = 0$ then the result is vacuously true. Assume the result holds for all distances less than or equal to some n, and let v be a vertex with distance $n+1$ (and thus v's parents have distance at most n). Let u be a parent of v, such that u is examined as part of some MPG.

Case 1: u is on the PV. After the first child is examined, α will be $f(v_0)$ if $p(u) = p(v_0)$ and $-f(v_0)$ otherwise. The following table summarizes the possible bounds seen from both u's perspective before v is examined, and from v's perspective.

Does $p(u) = p(v_0)$	$\langle \alpha, \beta \rangle_u$	$\langle \alpha, \beta \rangle_v$
N $\Rightarrow f(u) = -f(v_0)$	$\langle -f(v_0), \infty \rangle$	$\langle -\infty, \quad f(v_0) \rangle$
Y $\Rightarrow f(u) = \quad f(v_0)$	$\langle \quad f(v_0), \infty \rangle$	$\langle -\infty, -f(v_0) \rangle$

Since u is on the PV, $f_u(v) \leq f(u)$, and the desired consequents of (ii) and (iii) follow by using the equality from column one and straightforward symbol manipulation.

Case 2: u is not on the PV. Then u's distance is > 0 and the inductive hypothesis applies. We now enumerate the possible bounds for u, and show that the desired results must hold for v. Define $\langle \alpha, \beta \rangle_u$ to be the incoming α-β bounds for some node u, from the perspective of that node.

Case 2a: $\langle \alpha, \beta \rangle_u = \langle -\infty, f(v_0) \rangle \Rightarrow f(u) \geq f(v_0)$ by (ii)

$\Rightarrow f_u(v) \geq f(v_0)$ since u is a cutting node

$\Rightarrow f(v) \leq -f(v_0)$

and $\langle \alpha, \beta \rangle_v = \langle -f(v_0), \infty \rangle$ by u being a cutting node

Case 2b: $\langle \alpha, \beta \rangle_u = \langle -\infty, -f(v_0) \rangle \Rightarrow f(u) \geq -f(v_0)$ by (iii)

$\Rightarrow f_u(v) \geq -f(v_0)$ since u is a cutting node

$\Rightarrow f(v) \leq f(v_0)$

and $\langle \alpha, \beta \rangle_v = \langle f(v_0), \infty \rangle$ by u being a cutting node

Case 2c: $\langle \alpha, \beta \rangle_u = \langle f(v_0), \infty \rangle \Rightarrow f(u) \leq f(v_0)$ by (iv)

$\Rightarrow f_u(v) \leq f(v_0)$ holds for all v by definition

$\Rightarrow f(v) \geq -f(v_0), \quad \langle \alpha, \beta \rangle_v = \langle -\infty, -f(v_0) \rangle$

Case 2d: $\langle \alpha, \beta \rangle_u = \langle -f(v_0), \infty \rangle \Rightarrow f(u) \leq -f(v_0)$ by (v)

$\Rightarrow f_u(v) \leq -f(v_0)$

$\Rightarrow f(v) \geq f(v_0), \quad \langle \alpha, \beta \rangle_v = \langle -\infty, f(v_0) \rangle$

Since we have enumerated all the ways in which $\langle \alpha, \beta \rangle_v$ can occur, implications (i) through (v) must always hold. Therefore, by induction, the result holds for all n. □

3.4 NP-Completeness Results

In this section we show that determining the inclusion of an element in MinSetCover, which is NP-complete, can be reduced to determining the inclusion of an element in MinProofGraph-1 or to determining the inclusion of an element in MinProofGraph-2.

Sahni has shown that computing a minimum proof for an AND/OR tree (which is equivalent to MinProofGraph-1) is NP-Complete[37]. To help illustrate the source of the complexity we present a slightly different reduction. To see that MinProofGraph-1 is in NP, observe that a valid proof graph $H \subseteq G$ is a sufficient certificate. H is clearly polynomial in the size of G, and all constraints listed in Section 3.2 corresponding to H being valid are easily checked in polynomial time along with computing $f^{(H)}(v_0)$. By a similar argument MinProofGraph-2 is in NP.

In preparation for proving that all problems in NP can be polynomial-time reduced to MinProofGraph-1 and MinProofGraph-2 we introduce the following definition for the Minimum Set Cover problem:

Definition 3.4.1. **MinSetCover** is a set of succinct encodings of $\langle X, S, k\rangle$ where X is a finite set, and S is a collection of subsets of X such that there exists $S' \subseteq S$ where $|S'| \leq k$ and $X = \bigcup_{T \in S'} T$.

Theorem 3.4.2. *MinSetCover $\leq_p$ MinProofGraph-1 and MinSetCover $\leq_p$ MinProofGraph-2.*

Corollary 3.4.3. *MinProofGraph-1 and MinProofGraph-2 are NP-complete.*

Proof. We describe a function $\mathcal{F}$ that maps arbitrary code words w into the encoding of an associated MinProofGraph-1 instance with the following properties:

1. If w encodes a MinSetCover tuple $\langle X, S, k\rangle$, then $\langle X, S, k\rangle \in$ MinSetCover if and only if $\mathcal{F}(w) \in$ MinProofGraph-1.
2. Otherwise, $\mathcal{F}(w)$ encodes a fixed tuple not in MinProofGraph-1.
3. There exists a Turing machine **F** and a polynomial p such that for all w, **F** started on input w computes $\mathcal{F}(w)$ in at most $p(|w|)$ steps.

Given an element of Minimum Set Cover $\langle X, S, k\rangle$ consisting of $X = \{x_0, \ldots, x_{n-1}\}$ and sets $S = \{S_0, \ldots, S_{m-1}\}$, let $\mathcal{F}(\langle X, S, k\rangle) = \langle G, p, f, s, t\rangle$. $\mathcal{F}$ constructs a graph with four levels corresponding to the root node t, a dummy node u, the elements of X ($v_0, \ldots, v_{n-1}$), and the elements of S ($w_0, \ldots, w_{m-1}$), as in Figure 3.1. Let the edge set be $\{(t,u), (u,v_i) : 0 \leq i < n\}$ $\cup \{(v_i, w_j) : x_i \in S_j\}$. Let $p(t) = p(v_i) =$ white and $p(u) = p(w_i) =$ black, with white nodes having value 1 and black nodes having value -1. Finally, let $t = 1$ and $s = 2 + n + k$.

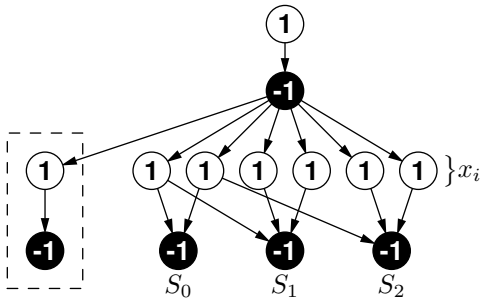

Figure 3.1: Sample construction of a DAG corresponding to an instance of Minimum Set Cover, with vertices labeled with their minimax value. Vertices x_i correspond to elements in the MinSetCover universe and S_i to the potential covering sets. Vertices in the dashed box are used when reducing to MinProofGraph-2.

We now show that $\langle G, p, f, s, t\rangle$ is accepted if and only if $\langle X, S, k\rangle$ is accepted. Any solution to the MinProofGraph-1 instance will include the first three levels and some fourth level nodes. By

definition the minimum number of fourth level nodes will be included, and as each third level v_i node has at least one associated fourth level w_j node, the fourth level nodes chosen correspond to a subset of S which forms a minimum set cover of X.

The proof for MinProofGraph-2 being NP-complete is almost identical, except that two new vertices are required as shown in Figure 3.1. In the minimax case at least one third level node must have all its children examined, but after that, due to the given node values, all other third level nodes need only examine any one of their children to produce an α-β cut. As the two new vertices must necessarily be included in any proof subgraph, we can assume without loss of generality that they are examined before the other vertices on their levels.

□

3.5 Computing Minimum Proof Graphs

Intuitively, the computational complexity of finding minimum proof graphs in the presence of nodes with in-degree ≥ 2 arises from deciding whether to choose a child with a "larger" proof graph, but which may be reused in multiple sub-proofs (via transpositions), rather than a greedily smaller child. As we will see in this section, the minimum proof graph of a tree (which must itself be a tree) can be computed bottom-up in linear time. We further distinguish between computing target and minimax proof trees.

Computing Target Proof Trees

For a MinProofGraph-1 instance $\langle G, p, f, s, t \rangle$ we may restrict the vertices of G that need to be considered by removing all white vertices v_i such that $f(v_i) < t$ and then removing components disconnected from the root as a preprocessing step in $O(|V| + |E|)$ time. If $f(v_0) < t$ then the instance is rejected.

To compute the minimum target proof graph size one need only recursively determine this value for the subgraph rooted at each vertex. For first player (OR) nodes this is the minimum over the sizes of the child proof graphs plus 1. For second player (AND) nodes this is the sum of the sizes of the child proof graphs plus 1. Extracting an associated minimum target proof graph is clearly trivial.

Computing Minimax Proof Trees

The α-β algorithm has been well studied in the case of trees [23], and it has been stated that the best-case move ordering is to examine children in order of decreasing score. This does not, however,

Function MPT_SIZE(v, α, β)

Data: subtree root vertex $v \in V(G)$, and α-β bounds.
Result: Minimum proof tree size for the subtree rooted at v, given α, β.
begin
  if $\alpha \geq \beta$ **then return** 0
  if $c(v) = \emptyset$ **then return** 1 // leaf node
  alias $T_w[a,b] :=$ MPT_SIZE$(w, -b, -a)$
  Assume MPT_SIZE *values are memoized across calls.*
  if $f(v) \leq \alpha$ **then return** $1 + \sum_{w \in c(v)} T_w[\alpha, \beta]$
  if $f(v) \geq \beta$ **then return** $1 + \min_{\substack{w \in c(v) \\ f_v(w) \geq \beta}} T_w[\alpha, \beta]$
  return $1 + \min_{\substack{w \in c(v) \\ f_v(w) = f(v)}} \left(T_w[\alpha, \beta] + \sum_{\substack{x \in c(v) \\ x \neq w}} T_x[f(v), \beta] \right)$ // search an optimal child first, then the rest
end

Figure 3.2: Algorithm for computing the minimum size of a minimax proof tree, if the input graph is a tree. The current subtree root is given by $v \in G$, and the current bounds are α and β. Minimax value function f and tree G are assumed to be global.

necessarily minimize the total proof tree size in the case of multiple best moves, where "best" refers to either absolute numerical score or ability to generate a cut. Using Theorem 3.3.3 we can see that there are at most three possible incoming bounds with which a given vertex can be explored in a minimal proof tree (MPT). We may thus compute, bottom-up, a table containing the MPT size for each vertex given the incoming α-β bounds. This process therefore runs in linear time and space in the graph size. Moreover, the table entries need only be computed as needed, and the set of potential best moves is easily constrained using the observations from Theorem 3.3.2's proof. The algorithm is listed in Figure 3.2, and the size of the minimum minimax proof tree is the result of $\mathrm{MPT\_size}(v_0, -\infty, \infty)$. Modifying the algorithm to actually construct an MPT is straightforward.

Computing Proof Graphs

The jump in algorithmic complexity going from trees to DAGs arises from the presence of transposition nodes (nodes with two or more parents). However if the structure of those transposition nodes within the graph is sufficiently amenable, then some hope may be had for computing a minimum proof graph.

To compute a minimum target proof graph in the case of a DAG, first observe that transposition nodes (t-nodes), must either be included or excluded in any proof graph. If a t-node is included then we require that an associated minimum target proof graph rooted at that node be included as well. If

the number of t-nodes is k, then it is straightforward (albeit expensive) to consider all 2^k cases where each node is either included or excluded. In each of these cases we may apply a slightly modified version of the minimum target proof tree (MTPT) algorithm. The modification being that, because we explicitly state whether a t-node is included or not, the MTPT algorithm already knows whether the value of each t-node is available to be used in a proof (although it still needs to compute the proof graph size, if not already cached).

Note that, in each of the 2^k cases, the choice to include or exclude a given t-node may be rendered moot by choices higher up in the DAG. By excluding one t-node, some or all of its descendant t-nodes may be excluded as well. As such, the number of actual cases that need to be considered may be significantly fewer than 2^k. Moreover, subgraphs which are disjoint may be anaysed independently. Given this, a tighter upper bound on the number of cases that need to be considered can be determined by looking at the structure of the t-nodes within the graph. Specifically, only the ancestor-descendant information of the t-nodes needs to be extracted, and which can be computed efficiently. A t-node cannot be reached if all of its ancestor t-nodes (including the root, if applicable) are excluded. Irrelevant t-nodes (OR-node children with value less than the threshold) should have already been pruned during pre-processing.

We now extend this algorithm to the case of computing minimax proof graph for an arbitrary DAG. This new algorithm combines ideas from the minimum target proof graph case with the idea of the α-β tables used in computing minimax proof graphs on trees. Specifically, in addition to deciding a priori whether a t-node is included or excluded, we decide on the α-β bounds it will be explored with. As per Theorem 3.3.1, there are three such possible α-β bounds. If we ever encounter a situation where a t-node's subgraph, as arising from our choice, is insufficient to prove a needed result, we backtrack. Any node which does not have a t-node as a descendant may of course have its table entries persist across iterations when the global set of t-node bounds change.

Note that because a t-node may be encountered multiple times, it may be searched with several incoming sets of bounds. Some incoming bound orderings are either impossible or result in equivalent subgraphs; a representative list of the interesting cases is as follows:

1. $\langle -\infty, \infty \rangle$
2. $\langle -\infty, f_v(v_0) \rangle$
3. $\langle f_v(v_0), \infty \rangle$
4. $\langle -\infty, f_v(v_0) \rangle, \langle f_v(v_0), \infty \rangle$

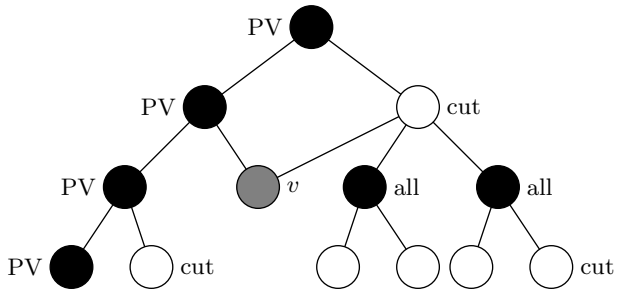

Figure 3.3: Alpha-beta tree where node v can be both an "all" node and a "cut" node, if the value of v is equal to that of the PV.

5. $\langle f_v(v_0), \infty \rangle$, $\langle -\infty, f_v(v_0) \rangle$

As noted in the preamble to Theorem 3.3.1, the first case, $\langle -\infty, \infty \rangle$, will occur for vertices on the PV and cannot require less work (a smaller subgraph) than any other range. The second case is a cut node, and the third case is an "all" node. The fourth and fifth cases correspond to a node v being searched as both a cut node and an all node, as can be seen to occur in the example of Figure 3.3. This can only occur if the node's value is equal to that of the principal variation. In the example, v is examined using $\langle -\infty, f_v(v_0) \rangle$ when first encountered from the principal variation, and a second time using $\langle f_v(v_0), \infty \rangle$. The first examination will prove v's value to be at least $f_v(v_0)$, while the second examination will search all children to prove v's value is no more than $f_v(v_0)$. Because α-β must prove the exact value, the fourth and fifth cases result in subtrees that are equivalent to a full-window α-β search. Whether the fourth of firth case applies depends on the node's distance from the PV.

3.6 Fastest-Cut-First Heuristics

In this section we develop a move sorting heuristic for alpha-beta search based on the results for minimum proof graphs we have presented. In real-world game applications, usually none of the quantities used in $\mathrm{MPT\_size}$ are known at the time when the decision about the next move to be searched is to be made. One therefore has to resort to move ordering heuristics. In addition, search is often performed in DAGs, which complicates the computation of minimum proof graphs. Even so, if estimates for values *and* sizes are available, we could base move decisions on those, and hope to approximate minimum proof graphs sufficiently well. This idea is not new. Plaat et al. [33], for instance, suggest to use enhanced transposition table look-ups and bias moves towards those with

small subtrees, to exploit properties of inhomogeneous DAGs, such as varying out-degrees and leaf depths. Closer in terms of using both value and size estimates comes what is known since the mid-1990s as "fastest-first" search in the Othello programming community. The idea is to search moves first that have a considerable chance to produce a cutoff, and among those to prefer moves with small sub-DAGs. Employing such heuristics has improved Othello endgame solvers considerably (http://radagast.se/othello).

Because both move values and DAG sizes are taken into account we prefer to call heuristics of this type "fastest-cut-first" (FCF) heuristics. For example, we could sort moves according to $(V_i - C \cdot S_i / \max_j S_j)$, where V_i and S_i are the estimated values and sub-DAG sizes for each move, and $C > 0$ is a constant. Alternatively, moves can be restricted to those with a reasonable chance of producing a cut, e.g., $V_i \geq \beta - C$, and from this set a move with minimal size estimate is selected.

Here, we propose a new fastest-cut-first heuristic which is motivated by the following theorem:

Theorem 3.6.1. *Consider a naive version of alpha-beta search that does not adjust α based on returned values and searches a game tree starting at cut-node v with successors $v_1, \ldots, v_n$. Let S_i be the expected size of the subtree rooted at v_i, and P_i the probability of move i leading to a β-cut. Then, visiting v_i in ascending order of S_i / P_i minimizes the expected number of visited nodes starting at v, if all P_i are independent.*

Proof. For move ordering $(1, \ldots, k, \ldots, n)$ the expected number of visited nodes is

$$1 + \sum_{i=1}^{n} \Big[S_i \prod_{j=1}^{i-1} (1 - P_j) \Big] \tag{3.1}$$

$$= A + (S_k \cdot \pi) + (S_{k+1} \cdot \pi(1 - P_k)) + C, \tag{3.2}$$

for $\pi := \prod_{j=1}^{k-1} (1 - P_j)$ and suitable $A > 0$ and $C \geq 0$.
Switching move k and $k+1$ leads to expected search effort

$$A + (S_{k+1} \cdot \pi) + (S_k \cdot \pi(1 - P_{k+1})) + C.$$

Thus, if $\pi > 0$ searching move $k+1$ before k while keeping the other moves' ordering constant is not detrimental iff

$$S_{k+1} + S_k(1 - P_{k+1}) \leq S_k + S_{k+1}(1 - P_k)$$

$\Leftrightarrow$

$$\frac{S_{k+1}}{P_{k+1}} \leq \frac{S_k}{P_k}, \tag{3.3}$$

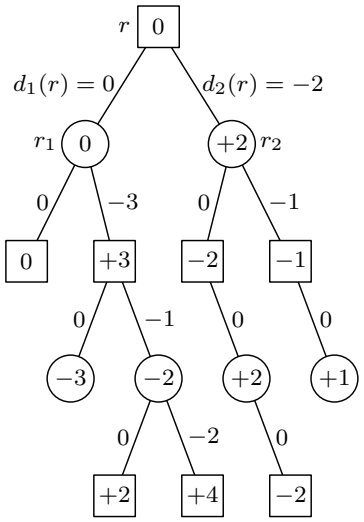

Figure 3.4: An incremental game tree. Node values indicate the current path value $p(v)$ in the view of the player to move. Edge values represent the damage $d_i(v)$ self-inflicted by playing potentially non-optimal moves.

where we define $c/0 := \infty$ for $c > 0$ to cover the cases $P_k = 0$ and/or $P_{k+1} = 0$. If $\pi = 0$ then switching moves k and $k+1$ does not change the expected search effort. This shows that when starting with an arbitrary move ordering we can continue switching moves k and $k+1$ for which (3.3) holds and not increase the expected search effort. Therefore, the expected number of searched nodes is minimized when visiting nodes v_i in ascending order of S_i/P_i. □

Picking the move with the lowest S_i/P_i ratio first accomplishes the goal of producing a cut with high probability, while keeping the node count low, in a principled way, unlike the ad-hoc fastest-cut-first heuristics presented earlier. The expected number of searched nodes established in the theorem is an upper bound on the actual number of nodes the true alpha-beta algorithm would visit, because with rising α values, the expected tree sizes S_i do not increase. So, by minimizing this number we can hope to decrease the alpha-beta search effort — also when searching DAGs. We will present experimental results on the performance of the novel fastest-cut-first heuristic in Section 3.8.

3.7 Prefix Value Game Trees

In this section we present a synthetic game tree model that we will use to gauge the performance of the fastest-first heuristic we proposed in the previous section.

Search algorithm improvements are often driven by specific applications. A classic example is the development of forward pruning and parallelization in computer chess. When working in particular search domains, state space properties — such as the average branching factor, the density of terminal nodes, state value distributions and correlations — are fixed. To gain insight into how such properties generally affect search algorithms, several synthetic game tree models have been proposed in the literature, e.g.[13, 31, 30, 32, 21, 39], with emphasis on studying pathology in alpha-beta search.

In [13] Fuller et al. describe a game tree model in which node value correlations are introduced by assigning values from $\{1, \ldots, N\}$ to edges and assigning a value $p(v)$ to leaves by summing up edge values along the unique path from the root node. One drawback of this so-called incremental game tree model is that values increase with depth and therefore don't have the usual interpretation as representing the expected payoff. In [39] this bias is offset first by considering move values from $\{-N, \ldots, N\}$, and then in a second step by realizing that playing a move can never increase a position's value, i.e.

$$p(v_i) = -p(v) - d_i(v), \tag{3.4}$$

where $p(x)$ denotes the accumulated value along the unique path when reaching node x in view of the player to move, v_i is the i-th successor of v, and $-D \leq d_i(v) \leq 0$ is the bounded damage self-inflicted by playing move i in v. Figure 3.4 shows an example. To evaluate search performance it is useful to know exact values of intermediate states. However, in this incremental model — when d_i is chosen randomly from $[-D, 0]$ — searching entire subtrees is required to compute the true negamax value of interior nodes, which can be very costly in terms of space and time requirements. With a simple yet effective refinement of the incremental tree model, we are able to determine exact node values *on the fly* while generating trees. Being able to (partially) reveal true state evaluations in interior nodes at no additional cost greatly helps analyzing search algorithms in general settings because no time has to be spent on engineering domain dependent heuristics. For instance, noise can be added when testing heuristics that depend on heuristic state evaluations of interior nodes, such as ProbCut [3], e.g.$\hat{f}_C(v) = f(v) + C \cdot e$, where $f(v)$ denotes the true negamax value of v and e is $\mathcal{N}(0,1)$ normally distributed. In addition, move preferences can be leaked probabilistically by means of the parameterized softmax function

$$\text{Prob(move } k \text{ is best in state } v) = \frac{\exp(C \cdot f(v_k))}{\sum_i \exp(C \cdot f(v_i))},$$

which spans the range from uniform random ($C = 0$) to perfectly informed choices ($C \to \infty$) and can be used to design synthetic move sorting routines for alpha-beta search or playout policies for

UCT [25].

Before we formulate the observation our model refinement is based on, we define the *prefix negamax value* of a node and show how to compute it. W.l.o.g., we assume that moves are alternated from now on.

Definition 3.7.1. In the incremental game tree model the *prefix negamax value*, $p(v)$, of node v is defined recursively based on (3.4): If v is the root r of the tree, then $p(v) = f(r)$. Otherwise, $p(v) = -p(v') - d_i(v')$, where v' is the predecessor of v on the path to the root, $v = v'_i$, and $d_i(v') \leq 0$.

Lemma 3.7.2. *For the incremental game tree model just described*

$$p(v) = (-1)^n f(r) + \sum_{k=0}^{n-1} (-1)^{n-k} d_{i_k}(u_k) \tag{3.5}$$

holds, where r is the root of the tree and $u_0, \ldots, u_n$ is the unique path from r to v, i.e. $u_0 = r, u_n = v$, and u_{k+1} is the i_k-th successor of u_k.

Proof. (Induction on depth n of v). For $n = 0$, we have $v = r$, so the statement trivially holds. Assuming (3.5) holds for nodes with depth n, consider the value of a node v' at depth $n+1$ with predecessor v and $v' = u_{i_n}$. Then:

$$\begin{aligned} p(v') &= -p(v) - d_{i_n}(v) \\ &= -\Big((-1)^n f(r) + \sum_{k=1}^{n-1} (-1)^{n-k} d_{i_k}(u_k)\Big) - d_{i_n}(v) \\ &= (-1)^{n+1} f(r) + \sum_{k=1}^{n} (-1)^{n+1-k} d_{i_k}(u_k) \end{aligned}$$

□

Our refinement of the incremental game tree model is based on the following observation:

Theorem 3.7.3. *If for all interior nodes v in trees generated by the incremental model there exists a move i with $d_i(v) = 0$, then for all v, $f(v) = p(v)$.*

Proof. (Induction on node height $h(v)$). If $h(v) = 0$, then v is a leaf and $f(v) = p(v)$ holds by definition. Now consider a node v with height $n+1$ and suppose $f(v') = p(v')$ holds for all v' with $h(v') \leq n$. W.l.o.g., we assume $d_1(v) = 0$. Among the $p(v_i)$ values, $p(v_1)$ is one of the smallest, because $p(v_i) = -p(v) - d_i \geq -p(v) = p(v_1) + d_1(v) = p(v_1)$. By the induction hypothesis we also know $f(v_i) = p(v_i)$ for all i. Therefore, $f(v) = \max_i(-f(v_i)) = \max_i(-p(v_i)) = -p(v_1) = p(v) + d_1 = p(v)$. □

With this result, evaluating nodes while generating trees is trivial because (3.5) can be maintained incrementally. This property in invaluable for analyzing search performance in large trees based on true node values. We call our refined model the "Prefix Value Game Tree Model". As usual, the number of successors and move values can either be fixed or sampled from distributions to better model real-world games. We will use prefix value trees to compare the performance of alpha-beta and fastest-first search in the next section.

3.8 Experiments

To investigate the effect on search effort resulting from the FCF heuristics, we constructed synthetic trees using various branching factors, edge value ranges, and heuristic evaluation errors. For each set of parameters we generated 500 trees and performed an iterative deepening (ID) search up to depth 10 using plain alpha-beta, NegaScout, and MTD(f) [35]. The expanded tree nodes were annotated with search information, such as estimated value and tree size, which was used for ordering the next ID iteration.

Our move ordering function is a weighted combination of the estimated node value, the S_i / P_i value, and S_i. For a given node we took S_i to be its out-degree. Looking deeper in the node's subtree was actually slightly detrimental since, ideally, large portions of that tree will never actually be expanded during search. Moreover the out-degree is essentially free to compute, given that we have expanded the node anyway. The value of P_i is initially a hard threshold based on whether the node value (from the last ID pass) is $\geq$ beta. Computing P_i empirically by training on trees from the same generating family produced a slight ($\sim$3%) improvement. The ordering of child nodes which have not yet been expanded is uniform random.

Performance results were consistent across the tree generation parameters, although the weights for combining heuristic components needed to be tuned. We made a best effort to determine the optimal weights in each case. In all cases, even when the node evaluation functions were given perfect accuracy, we observed a significant reduction in nodes expanded, compared to only using the estimated node value. Representative results are presented in Figure 3.5, which shows the *cumulative* number of search nodes expanded, including previous ID passes, relative to the MPG size. Values are averaged over 500 trees. The results shown are produced by trees with branching factors uniformly drawn from $[4..12]$ at each node, evaluations errors drawn from $[-4..4]$, and edge values drawn from $[-6..0]$. There was no significant difference in performance between using S_i / P_i and S_i as the FCF weighting term, with node reductions of 35% and 39% for NegaScout and MTD(f)

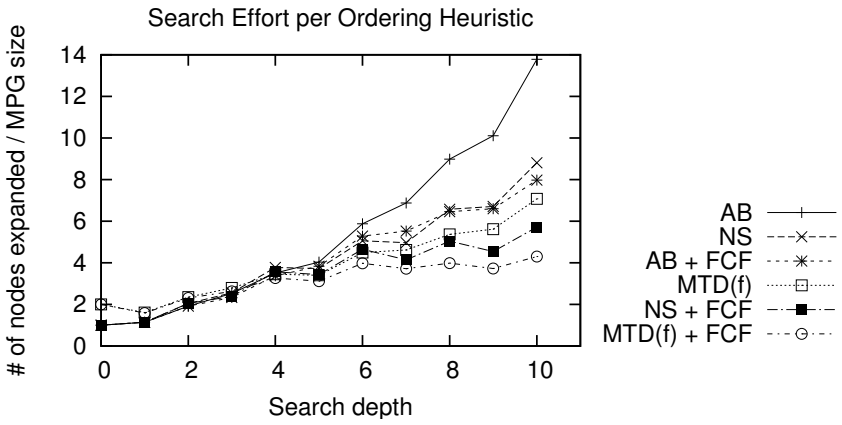

Figure 3.5: Number of search nodes expanded in synthetic trees when using different ordering heuristics. Node counts are shown as multiples of the minimum proof graph size. "AB" and "NS" refer to α-β and NegaScout respectively. "+FCF" denotes the inclusion of fastest-cut-first heuristics.

Ordering Heuristic	Avg. Time (ms)	Avg. Nodes
X	77.0	396,386
$X - C \cdot S_i$	44.7	243,090

Table 3.1: Effect of fastest-cut-first heuristics on the search effort of open-hand Skat initial cardplay positions.

respectively at depth 10. Note that the baseline MPG size grows exponentially with depth.

In addition to synthetic trees we used FCF heuristics in solving open-handed cardplay for Skat. Our search implementation sorts moves by way of a linear combination of features corresponding to the current value of a trick and the ability of teammates to contribute points. Our search also employs various pruning techniques and a transposition table — no iterative deepening is performed. Our lack of a forward-looking state evaluation function for Skat precluded an easy way to measure expected value or P_i. As such, we include an FCF term in our linear combination by subtracting S_i := the branching factor for each move, times C, a weighting term optimized independently. Results are shown in Table 3.1, with X being the portion of the move evaluation which does not consider tree size. Results are computed from 4000 initial cardplay positions from human games. Note that these results date from 2009, and so do not reflect the performance of our current search engin.

3.9 Conclusions and Future Work

In this chapter we have presented properties of minimum proof graphs, NP-completeness results, and a linear-time algorithm for computing minimum proof graphs for trees. We also introduced the class of fastest-cut-first heuristics which sort moves dependent on sub-DAG size and value estimates. Our experimental results using a novel synthetic game tree model indicate that standard alpha-beta search algorithms gain exponentially when using fastest-cut-first heuristics for move ordering, but an exponential gap between the minimum proof tree size and the size of the trees visited by alpha-beta search remains.

Chapter 4

Partition Search

> The room, the table, the Baron, a pair of terrified eyes - blue within blue, the eyes - all compressed around him in ruined symmetry.
>
> *Dune*
> FRANK HERBERT

4.1 Introduction

Modern high-performance search algorithms rely heavily on transposition tables to avoid duplicate search effort. The "game tree" is more often than not a graph, and the same position may be reached via multiple ancestors. Search algorithms like MTD(f) [35] are only effective because transposition tables can maintain upper and lower bounds across multiple queries (α-β searches with different bounds). But what constitutes a transposition? The most straightforward answer is that two positions transpose if they are identical in all respects. But in many search domains a given state may be coerced into an isomorphic "canonical" form. This canonical state is then used as a representative for all positions mapped to it via some MAKECANONICAL function. Such a function is usually hand-coded based on domain knowledge of what makes two states similar.

In this chapter we discuss methods for computing state isomorphisms based on the proof trees induced by each state. We will say that two states are equivalent if we can apply the same proof tree to each one, such that their minimax values are the same.

4.2 Partition Search

Ginsberg's partition search algorithm is a method for generalizing the evaluation of a particular state, to a class of related states with provably identical value. At a high level, the algorithm operates by using three functions. Given a domain S of game states, $R\colon S \rightarrow 2^S$ is a "reach" function such that the player to move in $R(s)$ can force a line of play that reaches a state (or set of states) s. For example, $R(s)$ can be the parents of state s. Intuitively, if player p wins in state s, then he also wins

in all $R(s)$. Similarly, a "constraint" function $C\colon S \rightarrow 2^S$ specifies which predecessor states, $C(s)$, *must* reach s in one or more moves. Thus, if Y is a losing set for player p, and X is constrained to reach Y (i.e., $X \subseteq C(Y)$), then X must also be a losing set for p. Finally, an equivalence function $P\colon S \rightarrow 2^S$ maps from a given state to a superset of states, all having the same value. A transposition table under such a scheme stores sets of equivalent states, and their associated bounds.

Because it is unreasonable to expect (efficient) perfect descriptions of the R, C, and P operators — as this would be tantamount to having solved the game in question — partition search allows for using correct approximations, which compute subsets of their optimal counterparts. These approximate functions should also be computationally tractable, such that sets of equivalent states can be computed and propagated back up the search tree. Applying partition search in the minimax sense to a root state s returns both the minimax value of s, and a superset of states, T, with identical value. Partition search may also be more generally applied, as with α-β, such that it returns an α-β evaluation and a set of states which satisfy the same bounds (e.g. $\forall s \in T : \text{VALUE}(s) \geq 5$, in the case of a β cut).

In multi-valued search settings (games with more than 2 outcomes), Ginsberg proposes the use of iterative zero-window search (e.g., MTD(f)). This is because the size of the equivalence sets produced is expected to decrease as the result-constraints become more restrictive — there are fewer states with value exactly x than those with value $\geq x$. Compared to a full-window α-β search that asks for the exact value of a state, the binary question of an MTD(f) iteration ($\text{VALUE}(s) \leq t$ or $\text{VALUE}(s) > t$ for some threshold t) should thus produce larger equivalence sets, resulting in more effective future transposition table lookups (i.e., having an increased likelihood of hitting the table).

In Bridge, because tricks which have already been played do not affect future min/max decisions, the current score is usually ignored when determining if states are identical — transposition tables store only the marginal value of a subtree (or bounds thereof). However, this means that we may reach the "same" state along different lines of play, with the declarer taking a different number of tricks along each line. Thus in attempting to answer the high-level (root-node) question of "does the declarer make 7 or more tricks?", search may need to ask "does the declarer make 3 or more tricks in (subgame) state s?", and, along a different line of play (i.e., a transposition), "does the declarer make 5 or more tricks in state s?". Thus we note that although a high-level MTD(f) query might be binary, the sub-problems encountered during search may have multiple and different thresholds.

Continuing this example, clearly we might expect that there will be fewer s-equivalent states with value ≥ 5 than there are with value ≥ 3, but what do we store in the transposition table? Assume we know that the value of s is ≥ 3, and that we have a corresponding set of equivalent states stored

in the transposition table, all proven to have value ≥ 3. If subsequent search (say, along a different transposition path) reveals that the value of s is ≥ 5 then we now have a choice as to whether we keep the larger equivalent set with the weaker bound, or store the smaller set with the tighter bound, or if we store both. Storing both provides the best performance in terms of nodes searched, but can result in more overhead when querying the table. The answer will necessarily depend on the domain and engineering constraints, but illustrates that a partition search transposition table is probably more complex than a standard α-β transposition table (although perhaps negligibly so, considering the amount of domain-specific engineering that goes into any high-performance search implementation).

Unfortunately Ginsberg offers little insight into how the critical *reach*, *constraint*, and *partition* operators may be constructed. He provides some intuition by way of example — a Bridge endgame in which the location (ownership) of certain low cards is irrelevant to the open-handed analysis — but does not offer any concrete operator descriptions. One must infer how these low cards are determined via his use of the phrase "cards that win tricks by virtue of their ranks". Thus while Ginsberg's implementation of partition search is highly effective at reducing the search effort in Bridge, applying the techniques to other domains, and even replicating them for Bridge, remains challenging.[1]

There may even be some confusion as to what partition search actually is. A description given by Smith et al. [40] describes partition search as "combining similar branches", with the example that if a player holds ♠56 then partition search would generate only one branch. Partition search is certainly capable of doing this, but the example seems highly misleading, as partition search is capable of finding much deeper isomorphisms. Indeed any trivial canonical move filtering would only explore one representative move in this situation. An enlightening practical implementation of partition search for Bridge is available from Haglund [19, 18] as open-source software, although the details in the accompanying technical report are somewhat cryptic.

4.3 Lowest-Winning-Rank

The important concept used in Haglund's approach, which is consistent with Ginsberg's example, is that of the *lowest winning rank* (LWR). Because partition search can essentially refer to any search algorithm which makes use of state equivalencies, the term itself is ambiguous. We shall refer to the

[1] We cannot discredit the possibility that there is a rich body of Bridge lore concerned with this topic, of which we are simply unaware.

particular variant described here (which we believe is the approach taken by Ginsberg) as *lowest-winning-rank search* (LWRS).

Ginsberg spoke of "cards that win tricks by virtue of their ranks"; we say that a card "wins by rank" if it wins a trick *because* it had a higher rank than another card played in the same trick. Consider the following tricks from a Bridge game where clubs is trump:

- ♡A – ♡K – ♡8 – ♡J ♡A wins by rank.
- ♡A – ♣K – ♡8 – ♡J ♣K win the trick, but not by rank.
- ♡A – ♣K – ♣8 – ♡J ♣K wins by rank.

In the first case, ♡A wins the trick because it was the highest card of the suit that was lead (and no player trumped). Thus ♡A wins by rank. In the second case, a trump card wins the trick, but only because it was the only trump played. No card wins by rank in this case. In the final case, two trump cards are played, so the higher trump card wins by rank. In a particular line of play, each trick may have at most one card which wins by rank. The LWR in each suit is the smallest card in that suit that wins by rank.

In determining whether a card wins by rank, it is always within a particular context — say, a fixed sequence of cardplay actions or a proof (sub)tree. In the case of a proof tree, LWR constraints are passed back along each edge. Proof tree OR nodes need only keep the constraints of one edge, while AND nodes take the union of all their children's constraints. Thus we may modify α-β to propagate the constraints corresponding to the proof tree that they implicitly construct during search.

Note that in general we must wait until after a trick is completed before we can determine whether a particular card won by rank. It is not meaningful to update or compute LWR values mid-trick. It is the LWR that determines whether or not a particular card is a "low" card — a card whose ownership is unimportant to the analysis of the proof tree. Low cards within a suit may be freely exchanged between players without affecting the value of the root state. The reason that the low-card ownership is unimportant is that, during the analysis of the position, the rank of those low cards was irrelevant.

If a card never wins by rank in the entire proof tree, then we may conclude that it was unimportant which player was holding the card — our proof never made use of that fact. Thus the proof tree is applicable to all positions in the same class (i.e., those states with a legal assignment of the low-ranked cards). Because the same proof is applicable, all members of a class must have the same open-handed value. This is not to say that all low cards may be indiscriminately interchanged. It is

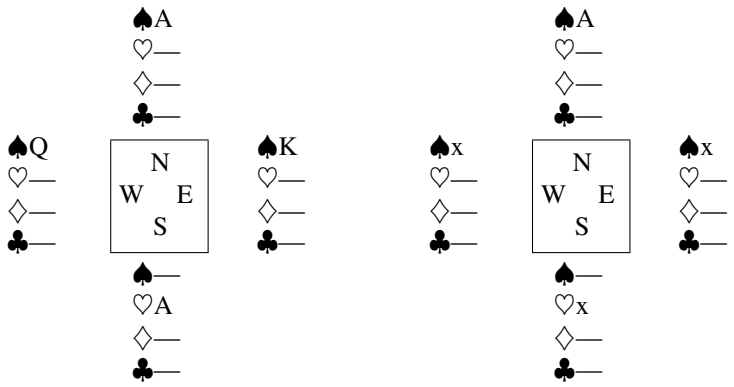

Figure 4.1: A simple Bridge endgame in which the locations of the ♠KQ may be exchanged without consequence. The set of equivalent states is depicted on the right, with small cards written as "x". Void suits are denoted by a line. In this case it does not matter which player is to lead, and even the location of the ♠A is irrelevant if South is to lead (and spades isn't trump).

usually important to a proof whether a particular player is void in a suit or not, as being void directly affects that player's ability to, say, trump or discard high ranks in another suit (to avoid acquiring the lead in the future). Exchanging low cards from different suits can affect this property, and is thus disallowed. Figure 4.1 gives a trivial example in which the ownership of certain low-ranked cards is unimportant ("low-ranked" with respect to the remaining cards in each suit).

Analysis

To prove the correctness of the LWRS algorithm we will first prove a more general result. Consider an N-player trick-taking card game in the spirit of Bridge or Skat, with an optional trump suit.

Let $\mathbb{D} \subset \mathbb{N}^2$ be a finite deck of cards with suit and rank. Define $\langle s_1, r_1 \rangle < \langle s_2, r_2 \rangle$ if and only if $r_1 < r_2$ and $s_1 = s_2$. Cards with different suits are incomparable.

Let $s = \langle C, \tau, p \rangle$ be a game state where p is the player to act, $C_i \subseteq \mathbb{D}$ is the set of cards held by player $i \in 1 \dots N$, and $\tau = \langle \tau_1, \dots, \tau_N \rangle$ are the cards in the current trick, indexed by who played them. Define $\tau_i = \emptyset$ if player i has no card in the current trick.

Let S^* be the game tree rooted at s, and let $S \subseteq S^*$ be an arbitrary subtree (e.g. a proof tree).

Let $\phi \colon \mathbb{D} \to \mathbb{N}_{|\mathbb{D}|}$ be an arbitrary bijective ordering of the cards, such that we may refer to the cards of each $s' \in S$ by index, rather than by their particular value.

Consider distinct cards $x_i \in \mathbb{D}$, with $i \in 1 \dots |\mathbb{D}|$. Define $s|_x$ to be the state obtained by remapping the cards in s (i.e., those in C and τ), in the natural manner, according to x and ϕ (i.e., replace the card with index i by card x_i). Note that $\text{PLAYER}(s|_x) = \text{PLAYER}(s)$. We wish to define

a set of (near) minimal constraints for x such that, for any satisfying assignment, $s|_x$ has the same subtree structure as S — namely, that there is a subtree rooted at $s|_x$ that is isomorphic to S. Call such a subtree S_x. If $\mathcal{C}$ is a set of constraints then define

$$[s]_{\mathcal{C}} := \{\, s|_x : x \text{ satisfies } \mathcal{C} \,\} \tag{4.1}$$

We now define the necessary constraints and then prove their correctness. To begin, define a set of suit constraints

$$\mathcal{C}_0 := \{\, \textsc{suit}(x_{\phi(c)}) = \textsc{suit}(c) : \forall c \in \mathbb{D} \,\} \tag{4.2}$$

such that each x_i has the same suit as its associated card in s. Note that the legal actions in any particular state are defined only by the suit of the led card (if any), the player to move, and the suits of the player-to-move's cards — players must follow suit if able, otherwise they may play any card.

It remains to ensure that the same player is to act in each node of S_x as in the equivalent node of S. Consider an arbitrary edge $\langle s_i, t\rangle$ between two states s_i and t in S, such that for move m, $\textsc{successor}(s_i, m) = t$. If t is not the start of a new trick then no constraints are necessary beyond $\mathcal{C}_0$. Assume t does start a new trick (or equivalently is a terminal node); we define a constraint to ensure that the trick winner is the same. Say $s_i = \langle C, \tau, p\rangle$, and $t = \langle C', \tau', p'\rangle$. Then $w := p'$ won the last trick. Let each player's card in the finished trick be given by

$$\tau_i^* := \begin{cases} \tau_i & \text{if } i \neq p \\ m & \text{otherwise} \end{cases} \tag{4.3}$$

and now require that card ranks result in the same player winning the trick:

$$\mathcal{C}_{s_i} := \{\, x_{\phi(\tau_w^*)} > x_{\phi(\tau_i^*)} : \tau_w^* > \tau_i^*,\ i \in 1 \ldots N \,\} \tag{4.4}$$

Let $\mathcal{C} := \bigcup_i \mathcal{C}_i$ be the union of all such constraints from (4.2) and (4.4). Note that $\mathcal{C}$ always has a satisfying assignment, since the identity mapping, $x_{\phi(c)} = c$, trivially satisfies all constraints. We will refer to the partition search variant which propagates these particular constraints as *rank-constrained search* (RCS).

Theorem 4.3.1. *Consider a root node $r = \langle C, \tau, p\rangle$, with a subtree S, and $\mathcal{C}$ as previously defined. Then $\forall r' \in [r]_{\mathcal{C}}$, S has an isomorphic equivalent, S', in the subtree rooted at r', such that the isomorphic states have the same player to move, and the same set of legal actions*[2].

[2]Where we identify actions by the *index* of the original card in r, not the actual card.

Proof. Proceed by induction on the distance from the root node. If the distance is 0 then s' is trivially equivalent to s, since the same player is to move by construction, and legal actions are defined only by the card suits, which are identical.

Assume the result holds for distance n. Let $s \in S$ and $s' \in S'$, with $s \cong s'$, have distance n from their respective roots. As noted, the legal actions are the same. Consider an arbitrary action $m \in \mathbb{N}$ such that $\text{SUCCESSOR}(s, \phi^{-1}(m)) = t \in S$. Let x be such that $r|_x = r'$. By construction, $\mathcal{C}_s$ forces $\text{SUCCESSOR}(s', x_m) = t'$ to have the same player-to-move as t. Thus $t \cong t'$ and the result follows. □

Corollary 4.3.2. *Lowest-winning-rank search is correct.*

Proof. Consider an arbitrary game state, s, which is searched by LWRS. Let the corresponding proof tree be S, and $\mathcal{C}$ be the constraints induced by S. The lowest-winning-rank in suit u is equal to the minimum r such that $c := \langle u, r \rangle \in \mathbb{D}$ satisfies $(x_{\phi(c)} > x_i) \in \mathcal{C}$, for some i.

LWRS returns a set of states corresponding to the high-rank cards (those greater than or equal to the lowest-winning-rank in their suit) having the same ownership as in s (i.e. $\mathcal{C}_{LWRS} = \{\, x_{\phi(c)} = c : c \text{ is a "high" card} \,\}$). These equality constraints subsume the ordering constraints between high cards given by $\mathcal{C}$. If we assume that $\mathbb{D}$ is restricted to only those cards in s, then any card ranks not explicitly assigned by $\mathcal{C}_{LWRS}$ must be "low" ranks, and thus the constraints that high cards have higher rank than low cards (within a suit) is satisfied within any isomorphism satisfying the LWRS constraints. In particular, the LWRS constraints are a superset of $\mathcal{C}$, and therefore LWRS returns a subset of $[s]_{\mathcal{C}}$ and is correct. □

Whereby the constraints impose a partial ordering of the cards, the reader may think of RCS as returning equivalence sets corresponding to all legal linear orderings, and LWRS as only returning those in which the low cards are reordered. In practice this difference may not be as large as one might think. Consider that a proof tree must consider all possible refutation actions. This necessarily tends to create a number of pairwise constraints. We speculate that such constraints tend to be centered around the middle-ranked cards, and that there may be a significant number of positions where the high-ranked cards are as or more irrelevant than the low cards (say, for misère games). This speculation is motivated by the efficacy of certain compression schemes for Skat Null tables, which depend on the card ordering used.

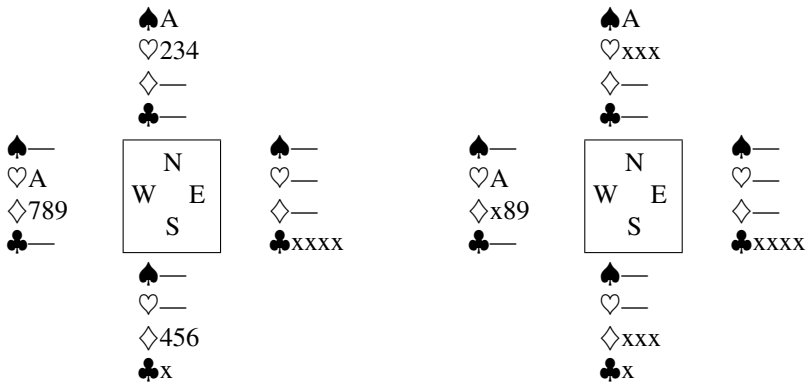

Figure 4.2: The Bridge position on the left is equivalent to the abstracted position on the right (North to lead). The location of the ♢7 is actually unimportant, but the computed LWR depends on which of ♢789 is chosen as the representative move in response to the ♠A lead.

4.4 Pruning

An obvious search optimization in the open-handed setting is to consider a single representative move from each group of power-equivalent cards. Two cards are power-equivalent if no other player (including the current trick) holds a card with intermediate rank. Note that we only need to consider the current trick-winning card; played cards which cannot win the current trick (and thus affect the future) may be ignored. For example, if the current player is holding ♠789, then playing ♠7 has the same open-handed value as playing ♠8 or ♠9. However care must be taken to ensure that such pruning does not interact badly with LWRS. Consider the following Bridge example: we are holding ♠789, and playing the ♠7 leads to a subtree where the lowest winning spade rank is the 8. This is equivalent to saying that the LWR is the lowest remaining card in that equivalence block. Thus, playing the ♠8 (or the ♠9) would result in a subtree where the LWR is the ♠7.

If the current state is a cutting node in the search tree, such that we need only examine one child capable of producing a beta cut, then there is no correctness problem. That is, because the proof tree only requires a single refutation move, any move that produces a refutation will suffice, although the LWR returned by examining the ♠8 may be needlessly conservative — we could have had a more general result (higher LWR) by examining the ♠7. If the current state is an "all" node, then we must examine all possible moves. In this case it is critical that we consider the effects of selecting a power-equivalent representative. In practice, a careful analysis of the LWR returned and the representative card played can be used to make this correction.

It is not the case that always playing either the highest- or lowest-rank equivalent card is suffi-

cient to guarantee the best (least constrained) LWR values. First consider the case where we play one card to win the current trick, and the remaining cards never win by rank: it is obviously best to play the highest-ranked card in this instance. The converse occurs when the current card does not win the trick, but the remaining cards all win by rank. This case is shown in Figure 4.2, where responding to the ♠A lead with ♢7 is the best play, from an LWR perspective.

Rank-Constrained Search

As an alternative to explicitly correcting for equivalent-rank effects, consider rank-constrained-search. It is straightforward to see that any cards which are power-equivalent must have the same RCS constraints. The desired property of minimal LWR constraints naturally falls out from replicating constraints amongst all power-equivalent cards. In the RCS sense, the pruning of equivalent moves can also be seen as an optimized case of performing a partition transposition table lookup at the level of the immediate children.

4.5 Transposition Tables

As a practical matter, the LWRS ranks must be used in transposition tables (or similar lookup tables) to be truly effective. We wish only to note that because the transposition table entries should be canonical (i.e. with missing and/or played card ranks collapsed), the LWR values must be *relative* ranks, rather than the absolute card ranks normally propagated during search. It is fairly straightforward to convert back and forth between the two, but it makes for an additional complication that must be kept in mind.

4.6 Conclusions and Future Work

In this chapter we have presented a theoretical framework with which to exploit the symmetries that arise in trick-taking card games, and in so doing have provided a clearer description of Ginsberg's partition search algorithm.

Further symmetries could be exploited by not imposing high-level constraints on the suits of individual cards. We could instead maintain constraints with respect to each card's suit — specifying whether cards have the same suit or not. If the "true" card played followed suit, then it must have the same suit as the led card. If it did not follow suit then the player must have no such cards with that suit. This would result in much more bookkeeping, and may impair equivalence checks, but

has the potential to construct much larger equivalence classes. Whether those larger classes would significantly improve search performance is unclear, as this may only occur for degenerate states that are trivial to analyse.

It also remains to extend these equivalencies to Skat, although some work in that direction has occurred with respect to compressing lookup tables. Further details are provided in Chapter 8.

Chapter 5

Static Analysis

> Chani took the largest ring, held it on a finger. "Thirty liters," she said. One by one, she took the others, showing each to Paul, counting them.
>
> *Dune*
> FRANK HERBERT

5.1 Introduction

The majority of this dissertation is concerned with quickly evaluating the open-handed (perfect information) value of card game positions. Open-handed evaluations are the core of perfect information Monte Carlo (PIMC) agents, and comprise most of their computational effort. Although PIMC-based approaches lack satisfying theoretical guarantees as to their regret or exploitability, they are the best technique currently known for producing strong agents — on par with human experts, depending on the domain. Search may also be used to bias the worlds sampled by PIMC, by considering the move our agent would have made and applying Bayes' rule. Recursive move evaluations which play out real games according to a given (PIMC) policy require significantly more computational resources, and are challenging to use under real-time tournament constraints. Increasing the speed of open-handed evaluations has direct practical benefits, including but not limited to faster experimental turnaround, and allowing for bootstrapping approaches which may require an agent to play many millions of games.

Evaluating open-handed positions may be accomplished in a straightforward but suboptimal manner by using standard α-β search techniques. As with most high performance search implementations, significant effort must be spent tuning the algorithm to the particular domain. In the coming chapters we will discuss several search extensions for increasing the speed of α-β as applied to Skat, but first it is worthwhile to discuss methods for avoiding or mitigating α-β search. To wit, we present some *static analysis* techniques — which employ little or no search — for computing bounds on the value of a position. If these bounds are exact, or if they lie outside the current α-β

window, then we may return immediately without searching the current subtree. Otherwise we may tighten our α-β window to increase the likelihood of future cuts. We will assume that the Skat and Bridge positions dealt with are perfect information.

5.2 Skat

In the case of trick-taking card games such as Skat, players accumulate cards during the course of the game. If the scoring function is sufficiently reasonable (e.g. monotonic with respect to winning additional cards) then a trivial upper bound may be computed by assigning the remaining tricks (card points) to the current team, and a lower bound by assigning them to the opponents. With some additional logic it is clear that this technique may be easily extended to certain non-monotonic scoring functions, such as Hearts, where "shooting the moon" is possible. In Skat, these trivial bounds rarely have any effect in terms of reducing the number of nodes searched.

Playing From the Top

A more detailed analysis is possible for games such as Skat or Bridge, by considering the actual distribution of cards held by each player. This involves what may be called "playing from the top". Assume it is the start of a trick. A lower bound for the team to act may be computed by considering only lines of play in which that team does not sacrifice the lead. This will usually involve playing the highest card in each suit, such that the opposing team cannot possibly win the trick. For the opponents to win the trick, they would have to either play a higher card of the led suit, or play a trump card. If the current player runs out of such guaranteed trick-winning moves, we pessimistically assign the remaining cards to the opposing team. For convenience we assume that moves which don't guarantee winning a trick are illegal — the team to act simply stops when no winning moves are available.

Kupferschmid and Helmert present such a method for computing bounds in Skat subgames [26]. Their method is an application of the general principle that (assuming rational agents) the current player can only be hurt by restricting her legal actions and/or by increasing the legal actions of her opponents.

Consider the case where the soloist is to lead, and we wish to compute a lower bound. For simplicity, first consider the case where we wish only to maximize the number of tricks that the soloist wins. Observe that, when playing for guaranteed tricks, it is optimal for the soloist to always play the highest card in a suit — all cards that cannot be overtaken are power-equivalent, and all

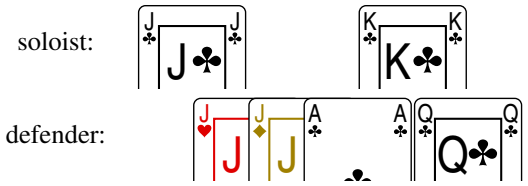

Figure 5.1: Static analysis Skat position where having the defender play his lowest-ranked card (♣Q) in response to ♣J sacrifices more points than is necessary.

cards that may be overtaken cannot be considered. Also note that it is optimal for the defenders to always play their lowest card in a suit when losing a trick, thus making it more difficult for the soloist to guarantee future tricks.

If the defenders cannot follow suit then they must discard from a different suit. For reasons of pessimism we allow the discard decision to be deferred. That is, we keep track of the number of off-suit cards each defender must discard, but only assign precise cards when the soloist has finished playing winning moves. In the case where we are only counting guaranteed tricks, this last step is unimportant. In the case of Skat-like games where cards have point values, the decision does matter and the deferred cards are chosen to be the least valuable ones remaining.

The soloist cannot play a suit if a defender is both void in that suit and has trump cards, as the trick would not be guaranteed. For this reason it is preferable for the soloist to first play from the top in the trump suit, with the goal of removing all defender trump cards. The order in which the remaining suits are played is irrelevant.

The algorithm currently stated is that given by Kupferschmid. However there is one complication which has not yet been addressed. When playing a suit game, the value of the remaining trump cards may not be monotonic. Specifically, the Jacks have higher rank than the Ace, Ten, King, and Queen, but are worth fewer points. Thus, if Jacks remain, having the defenders play their lowest-power card when losing a trick may not be optimal, since it can cause them to lose more points than necessary. This is illustrated in Figure 5.1. Kupferschmid addresses this problem by allowing the defenders to exchange the values of their cards, essentially re-valuing them to be monotonic in accordance with rank. In Figure 5.1 this would produce the correct result.

Non-Monotonic Card Values

We present a different approach which is more generally applicable to non-monotonic card values, and which can produces tighter bounds. We restrict ourselves to the case of non-negative card

values. If the soloist may safely play all of his cards in a given suit then the defender optimization is trivial: simply discard the least-valuable cards. If the soloist may not safely play out a suit then it is because a defender is preventing him from doing so, by virtue of trump or having a higher card. In the case of winning by trump, the other defender (the one who isn't preventing the soloist from winning all the tricks) need only discard her least-valuable cards. In the case of winning by rank, the overtaking defender must keep at least one card of sufficiently high rank (without loss of generality, her most valuable such card), and may otherwise discard her k least-valuable cards, where the soloist is guaranteed k tricks. As before, the non-winning defender may simply discard her least-valuable cards. If both defenders are able to prevent the soloist from taking all the tricks, then we simply consider both cases and choose the one which is most beneficial to the defenders.

We note that it is impossible for the defenders to save points by throwing off a low-value high-rank card, if doing so causes them to lose one or more additional tricks.

Lemma 5.2.1. *In Skat it is never optimal for the rank-winning defender to play a winning-rank card such that the soloist is then guaranteed one or more additional tricks.*

Proof. First note that the soloist winning any additional tricks can only cause the other (non-rank-winning) defender to lose more points. Now consider the rank-winning defender; let the multiset D be the values of that defender's cards, and $x \in D$ be the value of the card that wins by rank. Let $\Sigma_{(k)}\{D \setminus x\}$ be the sum of the k smallest elements in $D \setminus x$ (i.e., the k cards the defender throws off). Assume there is a better discard ordering that loses at least one additional trick, with y as the new winning-rank card, then $\Sigma_{(k+1)}\{D \setminus y\} < \Sigma_{(k)}\{D \setminus x\}$. Let $C := D \setminus \{x, y\}$, which gives

$$\begin{aligned}\Sigma_{(k)}\{C \uplus y\} &> \Sigma_{(k+1)}\{C \uplus x\} \\ &\geq \Sigma_{(k+1)}\{C \uplus 0\} \\ &= \Sigma_{(k)}\{C\}, \quad \text{given non-negative card values}\end{aligned}$$

which is a contradiction since $\Sigma_{(k)}\{C \uplus y\} \leq \Sigma_{(k)}\{C\}$ for any y. □

Applied to Skat, Kupferschmid's approach may undervalue the soloist lower bound by a non-trivial margin, as seen in Figure 5.2. By revaluing the card ranks the defender is assumed to discard $2 + 3 + 4 = 9$ points, rather than the correct $3 + 4 + 10 = 17$.

The algorithm for computing a defender lower bound when the defenders are to lead is similar. For convenience we require that the trick leader not change (one defender remains in control the entire time). The non-leading partner always discards his most powerful/valuable cards when able

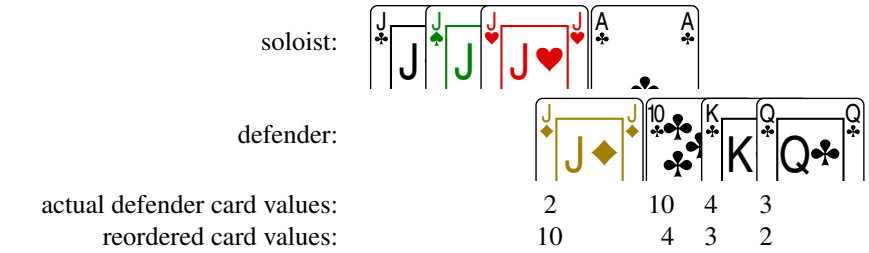

Figure 5.2: Situation where revaluing card ranks is detrimental to static analysis.

to do so. This has the side effect of making it easier for the leading defender to maintain control. Being able to transfer control would certainly make our lower bounds tighter, but the difference is not large, as seen in the next section.

Results

The heuristics we have presented for soloist and the defenders are approximations of the "optimal" control-maintaining lower bound — the number of points each team can accumulate without losing a trick. For instance, the soloist may be able to exploit having knowledge of the defenders' off-suit discards, rather than having them be deferred. The defenders could likewise exploit the declarer's discards, and would also be able to transfer control between themselves. We investigate the extent to which knowing this exact lower bound would improve performance by giving our search algorithm access to such an oracle. The oracle itself executes a search with the constraint that the team leading each trick cannot change (although control may pass within a team). Because the oracle itself is quite slow, results are presented in terms of nodes searched. Nodes searched by the oracle are not counted towards the total.

Results for suit and grand games, averaged over 2000 random deals, are shown in Figures 5.3 and 5.4. We consider all combinations of soloist and defender static analysis, as well as lookup tables of various sizes. When the soloist is to move, the difference between our static analysis implementation and the optimal analysis is negligible, with both providing a benefit over not using static analysis. The benefit decreases when using better lookup tables, and is greater for grand games. Our defender bounds have little to no effect, but the optimal defender bounds see reductions of about 20-25%. This suggests that a more detailed analysis that allows the defenders to transfer control may result in worthwhile speedups, even after taking overhead into account. We have only considered node reductions here, not wall-clock time, which must necessarily have lesser gains.

Compared to the approach of Kupferschmid, our static analysis bounds reduce the number of

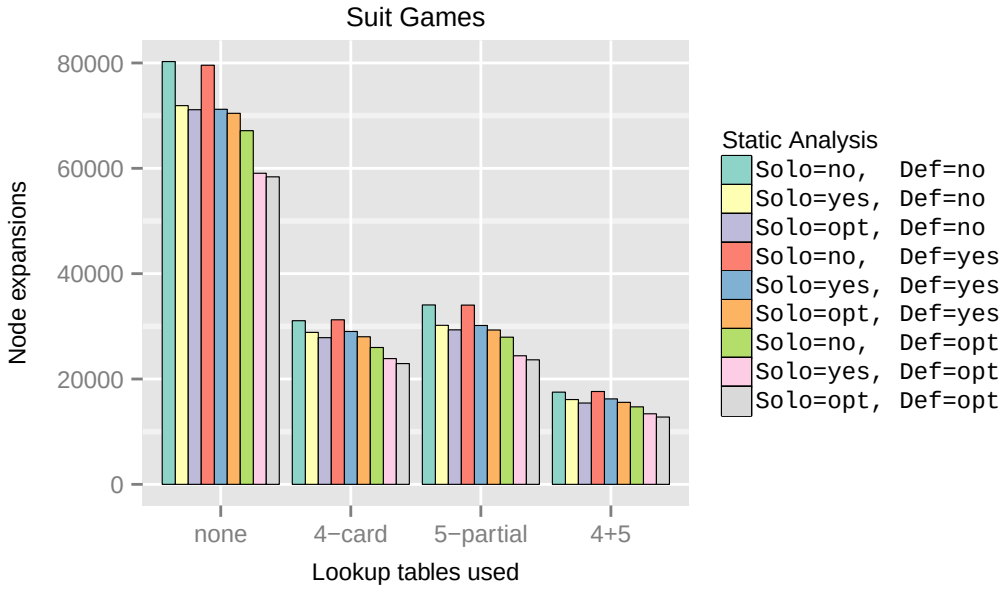

Figure 5.3: Effectiveness of static analysis bounds for suit games.

nodes expanded by slightly more than 1%, with a reduction in wall-clock time of about 1%.

5.3 Bridge

In considering possibilities to improve defender static analysis, it is worthwhile to consider the game of Bridge. The situation for static analysis in Bridge is similar, although the declarer and defender positions are symmetric. Because the cards in Bridge do not have point values, a large amount of the previously considered complexity disappears. Chang [6] describes using a lookup table to store the results of a "single suit" analysis, for solving perfect information Bridge positions. This approach is similar to the Skat approach, except that players may pass the trick lead to their partner. Because partners may have the option to overtake a card (either the trick leader's or the first defender's), each single suit analysis is solved via search, rather than deterministic discard rules. As in Skat, when players cannot follow suit, their particular choice of card is deferred, except to note if they could have played trump (in which case the original trick lead would be invalid). Deferring in this manner neither helps the current team, nor impairs the opponents, and thus is a logically sound pruning rule.

Single suit analyses provide lower bounds on the number of tricks that a team can make in each suit. These lower bounds are then combined into an aggregate lower bound for the open-handed (*double dummy*) position. As before, these bounds are intended to generate inexpensive cuts within

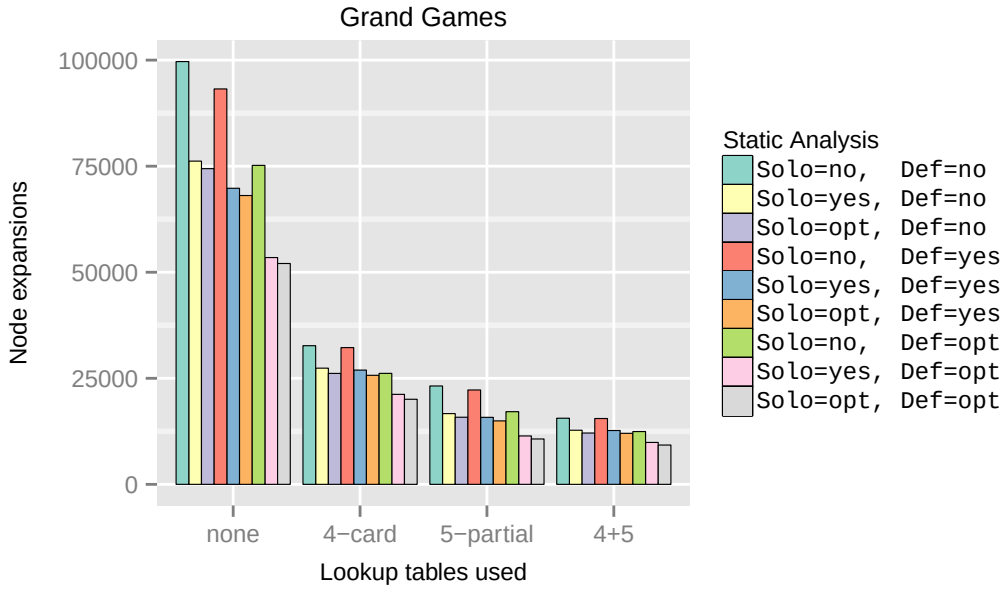

Figure 5.4: Effectiveness of static analysis bounds for grand games.

an α-β search.

What Chang's analysis fails to explore fully is the notion of *control* — namely which partner wins the final trick in the suit. Specifically, Chang fails to note that the winner of the final trick need not be fixed. There exist single suit contexts in which the lead team may choose which partner will win the final trick, by adjusting the order in which they play their cards. Such choices may however affect the number of guaranteed tricks in that suit. Chang appears to only consider one control possibility — resulting from myopically maximizing the number of guaranteed tricks.

The notion of control is important because it is required in the process of combining the lower bounds from multiple suits. That is, because the number of guaranteed tricks in a suit may change depending on which partner is to lead, a small amount of search may be warranted to optimize the global order in which suits are played, and by which partner. For example, it may be better to stop early in a suit, winning fewer tricks than is strictly possible, in order to transfer the lead and achieve even more tricks (total) in a different suit.

We now present a more comprehensive list of the possible control options within a single suit. For clarity we shall assume that North is the player who leads the first trick in the suit. Because player roles are symmetric, we also assume without loss of generality that North and South are the declarer team.

- KEEP — North wins the final trick in the suit.
- TRANSFER — South wins the final trick.
- THERE AND BACK (TAB) — North wins the final trick, and South wins at least one trick.

We construct a lookup table for each $\langle n, \text{control}, \text{trump} \rangle$ tuple, where n is the number of cards in the given suit, and "trump" describes which (if any) of the defenders hold trump cards. Each table has one entry for each of the 4^n possible card ownership assignments.[1] For each entry we optimize the number of guaranteed tricks under the given control constraint. These tables may be efficiently constructed in a bottom-up, dynamic programming manner.

A precise analysis of the combined lower bound (over all suits) requires distinguishing between the three control options. Intuitively we may see that North playing TAB allows for South to play one suit as KEEP, without North ever playing a suit as TRANSFER. The full analysis considers all legal combinations of control.

To use these lookup tables in the context of partition search, we also need to keep track of the lowest-winning-rank (LWR) for each table entry. Using standard LWR propagation rules we can compute the LWR value for each entry during the dynamic programming. If the team to move cannot guarantee making any tricks (due to the suit split) then the table entries have an LWR value of infinity. This situation occurs when either North has no cards in the suit, or an opponent is void and has trump cards.

5.4 Conclusions and Future Work

We have not yet been able to test the effectiveness of this more precise trick-counting in Bridge. Because high performance double-dummy Bridge solvers rely heavily on lowest-winning-rank search, it is not obvious that improving the static analysis would yield significant benefits. However, the techniques presented are applicable to Skat without substantial modification, and provide a direction for future work. This technique will still not be able to match the "optimal" analysis in general, as that approach is able to make tactical use of trump. Including this aspect in a table lookup would involve substantial extra context, and the additional overhead of considering all control options may negate any benefits. It would be worthwhile to construct a less-than-optimal oracle which does not

[1] In half of these configurations the opponents have the highest card, and thus no tricks can be guaranteed. We exclude these entries from our tables.

use trump to transfer the lead in this manner. Such an oracle would give a more meaningful upper bound on the potential improvements.

Chapter 6

Paranoid Search

"You were about to . . . make a mistake."
"To save one from a mistake is a gift of paradise"

Dune
FRANK HERBERT

6.1 Introduction

Often during a trick-taking card game one encounters a position in which one player (usually the soloist/declarer) is guaranteed to win the remainder of the tricks. Rather than forcing the other players to continue to make decisions, the soloist will shortcut the game by opening up her hand and announcing that she makes the remaining tricks. When humans are involved then shortcutting is simply a matter of courtesy.

A less extreme instance of winning-all-tricks is when the soloist has a strategy which is guaranteed to win the game, irrespective of the true card distribution. It is thus a "paranoid" strategy, since it assumes both the worst possible arrangement of defender cards and optimal defender play. More formally, a position p is unlosable for player i if there exists a strategy σ_i, such that, over all possible worlds (from player i's perspective) there is no legal sequence of opponent actions that can cause player i to lose (or obtain a score less than some threshold).

Sturtevant and Korf have previously [43] described a different "paranoid" algorithm for the case of perfect information multi-player games. In that work the player to act is paranoid with respect to the *preferences* of the other players, assuming that they are in a coalition against the agent. This reduces the multi-player game to a 2-player game, such that α-β pruning may be applied. Our work differs from Sturtevant and Korf's algorithm in that we assume the agent does not have perfect information. Moreover, because our agent does not know the true world, it is also paranoid with respect to the outcome of any stochastic events (chance nodes), namely the actual distribution of any unobserved cards.

Having a dedicated search module for finding a paranoid strategy is useful for a number of

reasons. First, it may be significantly faster to search against an abstract worst-case hand than to perform a separate α-β search for all possible worlds. Second, as noted previously, even if we could search all possible worlds, PIMC may not choose a paranoid strategy even if one exists. Due to strategy fusion, PIMC might return a move corresponding to lines of play that require knowing the true world.

6.2 Skat

In Skat we wish to recognize unlosable game announcements during bidding/declaration, and unlosable game states during cardplay. However, during cardplay, the soloist might not wish to reveal her hand, since the defenders may still misplay and hence be made schneider or schwarz. If they are already out of schneider/schwarz, then shortcutting the game may save time.

Having a winning strategy in all consistent worlds is a necessary but not sufficient condition for guaranteeing that a position is unlosable (cf. PIMC strategy fusion). Detecting a paranoid win using α-β search is conceptually simple: the root node of the search is essentially an information set, $\mathcal{I}_i(p)$ — the set of states consistent with position p's implied history *from player i's perspective*, where player i is the player looking to compute a paranoid strategy, and which need not be the root player. The implied history is simply those actions which led to state p. Every time an opponent is to act, the legal moves are all possible moves over the current set of consistent worlds. Whenever an opponent makes a move, we restrict the consistent worlds to only those worlds where the given move was actually legal. This algorithm is illustrated in Figure 6.1.

Paranoid search is not particularly meaningful from the perspective of the defenders in Skat or Bridge. As it is defined, it would require one defender to assume that his partner will intentionally play poorly. Although it may be worthwhile to search for lines of play in which it is impossible for one's partner to make a cardplay error, we feel that such situations do not arise often enough to justify this overhead. Thus we consider only paranoid search from the perspective of the soloist (or potential soloist, in the case of bidding).

Note that although we defined paranoid search in terms of an information set containing all consistent worlds, it is straightforward to apply it to the case where we are concerned with only a subset of the possible worlds. For instance we can sample k worlds, as in PIMC, and solve the resulting relaxed paranoid problem. This idea is related to Ginsberg's achievable sets, in which he searches for a strategy that maximizes the number of worlds in which a win is guaranteed.

In the case of Skat (and a large number of other card games), we need not maintain an explicit

```
Function PARANOIDSEARCH(S, α, β)
  Data: Set of possible worlds and current alpha-beta bounds.
  begin
    if CHILDREN(S) = ∅ then return min_{s∈S}{v(s)}
    ... Return if (paranoid) static analysis produces a cut...
    bestval ← −∞
    for m ∈ LEGAL(S) do
      Resulting states after making move m.
      S_m ← {SUCCESSOR(s, m) | s ∈ S and m ∈ LEGAL(s)}
      Assume alternating moves.
      v ← −PARANOIDSEARCH(S_m, −β, −α)
      if v ≥ β then return v
      α ← max{α, v}
      bestval ← max{bestval, v}
    return bestval
  end
```

Figure 6.1: Paranoid alpha-beta search

listing of the possible worlds. Instead, the possible worlds are implied by keeping track of the possible cards each player could be holding. A player's legal moves are any of his (possible) cards, but if a player fails to follow suit, we impose the constraint that he cannot be holding any cards of the led suit (i.e., we enforce consistency). To ensure that making a defender move does not result in an empty set of consistent worlds (because there was no world in which the defender move was actually legal), it suffices to check only the following:

Lemma 6.2.1. *Let A and B be the set of possible cards held by defender 1 and defender 2 respectively, possibly with non-empty intersection. Then there is at least one legal assignment of cards if and only if $n_1 \leq |A|$, $n_2 \leq |B|$, and $(n_1 + n_2) \leq |A \cup B|$, where n_1 and n_2 are the number of cards held by the respective defender.*

Proof. First assume a legal card assignment $D_1 \subseteq A$ and $D_2 \subseteq B$ for defenders 1 and 2. Then

$$
\begin{aligned}
&|D_1| = n_1 \quad \text{which implies} \quad n_1 \leq |A|, \\
&|D_2| = n_2 \quad \text{which implies} \quad n_2 \leq |B|, \\
&\text{by definition} \quad D_1 \cap D_2 = \emptyset \quad \text{so} \quad |D_1 \cup D_2| = (n_1 + n_2), \\
&\text{and} \quad (D_1 \cup D_2) \subseteq (A \cup B) \quad \text{so} \quad (n_1 + n_2) \leq |A \cup B|.
\end{aligned}
$$

Now consider the other direction, and assume $n_1 \leq |A|$, $n_2 \leq |B|$, and $(n_1 + n_2) \leq |A \cup B|$. If $|A \setminus B| \geq n_1$ then we may simply choose appropriate $D_1 \subseteq (A \setminus B)$ and $D_2 \subseteq B$. The case where

$|B \setminus A| \geq n_2$ is analogous. Now consider the case where $|A \setminus B| < n_1$ and $|B \setminus A| < n_2$. There is a satisfying assignment if

$$|A \cap B| - (n_1 - |A \setminus B|) - (n_2 - |B \setminus A|) \geq 0$$
$$|A \cap B| + |A \setminus B| + |B \setminus A| \geq n_1 + n_2$$
$$|A \cup B| \geq n_1 + n_2 \quad \text{which is true by assumption.}$$

□

If such a one-step consistency check was unavailable or not complete (in the logical sense), we could easily modify our paranoid α-β search to return a flag whenever an inconsistent state is detected, or if all moves lead to similarly inconsistent subtrees. In the case of Skat, our search infrastructure uses bitmasks for each player's cards. The paranoid variant simply relaxes the constraint that the defender bitmasks be non-intersecting. It should however be strongly emphasized that search extensions and performance optimizations from the regular case may break in both obvious and subtle ways when applied to the paranoid setting.

Consider the optimization that quickly evaluating the final trick. In the non-paranoid case, each player has one card remaining, and the winner of the trick may be quickly determined without using the full recursive α-β search infrastructure. In the paranoid case however, a defender may hold multiple possible cards, and each must be considered as a potential legal move.

Note that in the case of paranoid *hand* games, the soloist generally has no knowledge of the cards in the skat. Thus, from a search perspective, the skat cards may only be assigned after all other cards have been played (barring forced cards due to void inference). This tends to make static analysis bounds substantially worse during the midgame, relative to paranoid non-hand games. Paranoid null games are straightforward to compute, and may be solved without search by looking at the number of "safe" soloist cards in each suit.

In the case of Bridge, a weak version of paranoid play is described by Bethe [2]. This is intended for use in identifying situations where the declarer is guaranteed to make her contract, and then shortcut the cardplay. Bethe's method only considers lines of play where the declarer gives up the lead at most once (although the generalization to losing more tricks is obvious). His method considers a combined defender, that has all the defender cards and acts whenever a normal defender would be to play. Bethe's approach does not enforce consistency in terms of void suits, and is thus overly pessimistic.

6.3 Bidding

Kermit's bidding module traverses an expectimax tree of all possible skat pickups and all possible discards and game declarations. If we wish to make use of paranoid search during bidding then we must either be able to search quickly, or to quickly query stored results. Although we believe our search algorithm to be quite optimized, the number of searches required has driven us to use a lookup table.

The values stored in such a lookup table are lower bounds on the number of card points the soloist can achieve. From the perspective of bidding, we are only concerned with whether a game is a guaranteed win, schneider win, or schwarz win (i.e. 61+, 90+, or 120 points[1]). Because the paranoid value is rather pessimistic, in practice we treat all values of 57 or more points to be a win.

Empirically, the difficulty of a paranoid search is strongly correlated with its resulting paranoid value. The phase transition at which search becomes difficult lies around 30-40 declarer card points. Intuitively the soloist can often only guarantee winning a small number of tricks, and with those tricks having low value. Thus the search spends a large amount of effort determining just how badly the soloist can lose.

To apply paranoid search to Skat bidding we must consider two cases. The first is whether we have an unlosable hand game in each of the 5 trump suits, given our initial 10 cards. However we must also consider which player is to lead, for an additional factor of 3. Thus there are $3 \times 5 \times \binom{32}{10}$, or about 968 million such searches (table entries) possible — a moderately-sized lookup table. The second case involves knowing the skat, and has $3 \times 5 \times \binom{32}{10,2}$ or about 224 billion — much harder to justify on current commodity hardware.

There are certainly symmetries in the off-trump suits which may be exploited, but this is at most a factor of 6 for suit games, and 24 for grand. The $\binom{32}{10} \approx 64.5$ million initial hands becomes 8,164,748 after symmetry reduction. Fortunately, the lookup table compresses quite well using off the shelf general compression algorithms, specifically bzip2 as used by the Boost C++ library. However, because our bidding search must consider all $\binom{22}{2}$ pickups and all $\binom{12}{2}$ possible discards, the relevant entries in our lookup table have poor locality. Decompressing and scanning the entire lookup table would be quite slow without additional modifications. To wit, we replicate the table entries multiple times, placing values for both pickups and resulting discards next to each other. Thus a compressed table — one for each soloist position — is first indexed by the 10 initial cards,

[1] Although schwarz is concerned with winning all tricks, in this particular case a paranoid score of 120 is equivalent. Not so from a midgame position.

and a table "line" contains $\binom{22}{2} \times \binom{12}{2} \times 5$ entries, plus 5 more for hand game values. This "bloated" table is 66 times larger than the raw paranoid data. However, a high compression ratio makes this approach feasible. Moreover, this data need only be stored on disk, rather than in-memory.

Due to the nature of the bzip2 algorithm, we cannot seek to particular raw-table locations directly. Thus we group the data into blocks of 10 table lines, and compress the blocks separately, allowing us to access individual lines without significant overhead. The compressed blocks are then concatenated together. This approach requires a supplementary offset table, to identify where each compressed block begins (alternately this could be offloaded to the filesystem by having each block be a separate file, with additional overhead for page size alignment). Note that we only need to read in a table line once, after receiving our initial 10-card deal, as a line contains all the information we need for the subsequent computations. A single table line requires a negligible amount of memory. Our choice of 10 as the blocking factor roughly doubles the space required, compared to directly compressing the lookup tables that don't use blocking. The final on-disk tables require no more[2] than 5 GiB in total, which corresponds to a compression factor of over 200.

6.4 Results

We define a paranoid error for the soloist as a soloist move that turns a paranoid win into a paranoid loss. The per-trick paranoid error rate for suit games, is shown in Figure 6.2. We only consider moves which could have resulted in a paranoid error — positions where all moves are equally good do not affect the rate. Note that stronger players (searching 320 worlds instead of 10) have lower, but not vastly different, error rates. If it were the case that all paranoid errors were capitalized on by the defenders then the effect of these errors would be a significant drop in winning rates. That is, by using paranoid search to discard moves which are paranoid errors we could significantly improve the number of games won.

However, for the defenders to capitalize on these errors they must not only make the correct play, they must have the proper card distribution to do so. Thus if the errors can only materialize in a tiny fraction of possible worlds, then the expected effect would be negligible. In Figure 6.3 we show the effect of modifying Kermit to use paranoid search, and thus never make a paranoid error. Any differences in tournament points are not statistically significant. Perhaps the greatest benefit of paranoid filtering is in avoiding the embarrassment of making such a mistake, no matter how

[2]Because our table entries contain lower bounds with values less than 57 declarer points, the compression ratio is somewhat reduced and we only have convenient access to an upper bound.

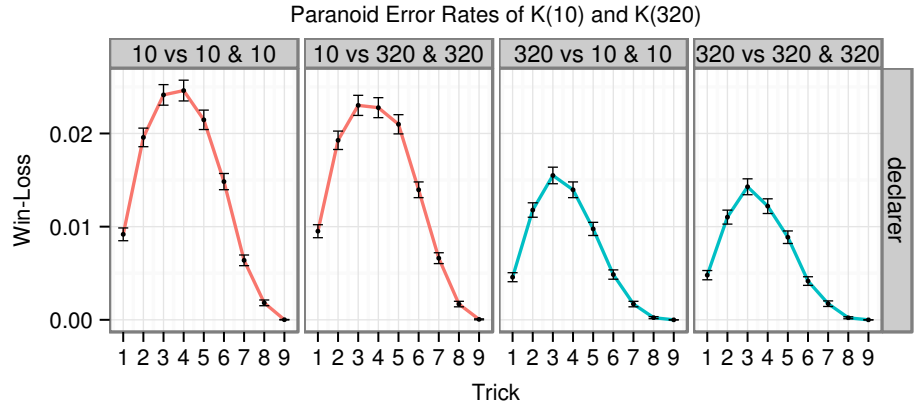

Figure 6.2: Paranoid error rate for suit games. Errors are defined as soloist moves that turn a guaranteed win into a possible loss. Facet headings denote the number of worlds sampled by the declarer and the two defenders.

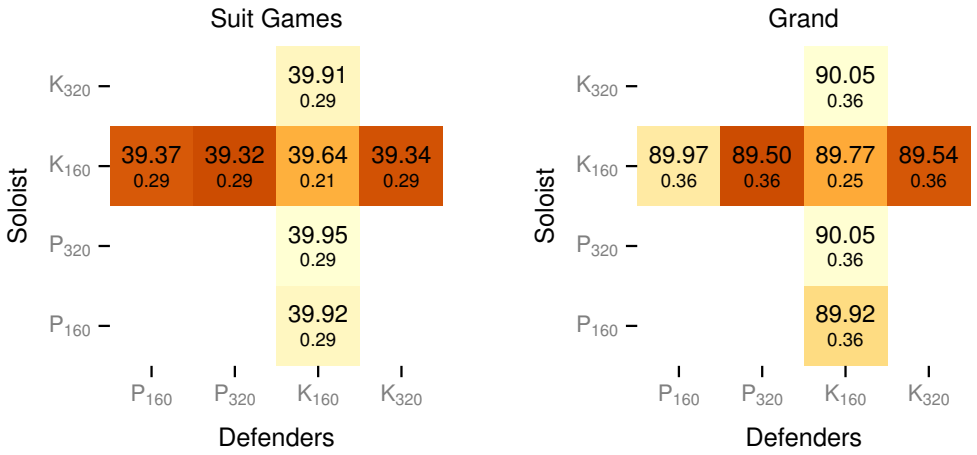

Figure 6.3: Kermit (K) versus Kermit with paranoid move filtering (P). Scores are given as soloist game points, with the value of one standard error in small print.

infrequently they occur. Similarly, in the case of bidding, we do not wish to play a low-value suit game when a guaranteed grand win is available.

6.5 Achievable Sets

An achievable set is a group of consistent worlds/states for which a guaranteed (paranoid) winning strategy exists. That is, the strategy guarantees a win in each world, not that each world has its own, possibly distinct, winning strategy. Ginsberg's GIB player makes use of achievable sets when determining its declarer strategy. Given a set of worlds, it is straightforward to compute whether a

satisfying paranoid strategy exists. Thus we can greedily grow an achievable set, given an ordered list of worlds. The size of the resulting achievable set depends on the initial ordering — it may be the case that guaranteeing a win in one world prevents a win in several others. It is currently intractable to compute a maximum achievable set for a moderate number of worlds, but multiple orderings may be attempted so as to compute several lower bounds.

Ginsberg makes use of his "squeaky wheel" heuristic to reorder worlds, moving those that cannot be satisfied in the current ordering to an earlier position. In Skat we instead make use of the additional information provided by the paranoid card point evaluation. Any Skat world in which we do not have a paranoid strategy guaranteeing 61 or more card points cannot be part of any consistent set. Thus we perform a filtering pass where we exclude any such worlds. We then arrange worlds in increasing order of card points, with the intuition that worlds where we can achieve a large number of card points likely have many winning strategies and thus are less constrained.

In practice we saw no benefit from using achievable sets. We considered both the obvious algorithm that tries to find a move with the best achievable set, constructed over all worlds in our budget (say, 160), as well as hybrid approaches that computed the average move score over k achievable sets of size n/k. Cardplay performance was almost always substantially worse. We speculate that this is due to the related effect that only considering the win-level of a move leads to poorer PIMC performance. Using the number of card points attained as a tie-breaker results in significantly stronger cardplay. We speculate that incorporating card points into the achievable set analysis could result in performance that is on par with vanilla PIMC. Certainly in the limit the hybrid technique should approach PIMC as the sets become sufficiently small.

6.6 Conclusions and Future Work

In this chapter we have shown how to compute a paranoid (worst-case) evaluation of a Skat position, and how such evaluations can be compressed and stored for use in bidding. These paranoid lookups can be used to augment feature-based function approximators which may suffer from a lack of training data or more general approximation errors. Current attempts to incorporate paranoid evaluations into Skat move selection have not been as successful as previously reported results in Bridge, but it is possible that somehow combining paranoid win-levels with card points may produce better results.

Chapter 7

Canonical Hashing

> He had seen this desert. But the set of the vision had been subtly different, like an optical image that had disappeared into his consciousness, been absorbed by memory, and now failed of perfect registry when projected onto the real scene.
>
> *Dune*
> FRANK HERBERT

7.1 Introduction

In the past 20 years there have been great successes for AI in popular abstract games. For example, we now have programs running on personal computers that can defeat the best humans at chess, checkers, Scrabble, Othello, and backgammon; computer Go has been revolutionized by Monte Carlo tree search; and computer poker has advanced to the point where humans no longer dominate. In most perfect information games the α-β algorithm and its various enhancements have been instrumental to these achievements. One important idea is to use transposition tables that store information about previous search results to improve move sorting in subsequent search iterations and to avoid repeating computations in case a state is reached via different move sequences.

In this chapter we first introduce a more general concept of sharing data between search trees — payoff-similarity — that can save search effort in game domains in which encountered subtrees may be identical but player payoffs can be different. We then show how payoff-similarity can be used in Skat to speed up open-handed computations by almost an order of magnitude. Finally, we discuss further search reductions based on approximating minimax values and forward pruning.

7.2 Payoff-Similarity

By definition, a transposition table (TT) is used to detect when search reaches a state that is identical to one that has previously been explored, usually via a different path (i.e., a *transposition*). However the functional benefit of a transposition table lies in how it relates the *value* of a previously seen

state p to the value of the current state c. Namely, if $c = p$ then $V(c) = V(p)$. More generally, one may consider only the portions of c and p that affect whether their values differ.

This idea leads to a further generalization that if we can compute a bound on the difference between the values of two (arbitrary) states, then we could potentially extract information about the value of the current state from *all* previously seen states. Practically this is infeasible, but in certain domains it is possible to exploit local structure to attain non-trivial bounds relating the values of states within a neighbourhood of each other.

Consider a two-player zero-sum perfect information game in which we want to relate the minimax values $V(p)$ and $V(c)$ of two states p and c, with subtrees T_p and T_c. If $p = c$ and $V(p)$ has already been determined and stored in a transposition table, then when reaching c we can use this information and return $V(c) = V(p)$ immediately, without searching T_c. If we have not encountered c before, T_c needs to be searched.

Now suppose $c \neq p$ but T_c and T_p are similar. Then we may be able to bound $|V(c) - V(p)|$, and knowledge of $V(p)$, or a bound on $V(p)$, could produce α or β cuts and save work. While finding bounds for $|V(c) - V(p)|$ small enough to create cuts may be hard in general, for particular types of trees this goal is attainable. Suppose, for instance, that T_c and T_p are structurally equivalent, i.e., there is an isomorphism between T_c and T_p that respects which player is to move. The payoffs in corresponding leaves may be different. We call such states c and p **payoff-similar** and can prove the following statement:

Theorem 7.2.1. *Let states s and s' be payoff-similar and $|V(l_i) - V(l_i')| \leq \delta$ for all corresponding leaf pairs (l_i, l_i') in subtrees T and T' rooted in s and s', respectively. Then $|V(s) - V(s')| \leq \delta$.*

Proof. Because s and s' are payoff-similar, T and T' are structurally equivalent. We proceed by induction on the height of corresponding nodes in T and T'. For corresponding leaves l and l' we know $|V(l_i) - V(l_i')| \leq \delta$. Now suppose the claim is true for all corresponding node pairs of height $\leq h$. Consider corresponding states s and s' with height $h+1$. W.l.o.g. let s and s' be states with max to move and successors $s_1, \ldots, s_k$ and $s_1', \ldots, s_k'$. Then, applying the induction hypothesis

yields:

$$\begin{aligned} V(s') &= \max_i \{V(s'_i)\} \\ &\leq \max_i \{V(s_i) + \delta\} \\ &= \max_i \{V(s_i)\} + \delta \\ &= V(s) + \delta \end{aligned}$$

and analogously $V(s') \geq V(s) - \delta$, and therefore $|V(s) - V(s')| \leq \delta$. □

Seeding the Transposition Table

In a manner similar to building an endgame database, we may pre-populate the transposition table (or a separate data structure) with a selection of states for which we have computed their exact value, or even just lower and upper bounds. If our domain is sufficiently amenable then search need not directly encounter any of these "seeded" positions, as long as it gets close enough to make use of their values via the similarity bounds. This can allow for a standard endgame database to be replaced by a transposition table of equivalent efficacy but with significantly fewer entries. Thus we can pay a one-time cost to compute transposition table entries and then use those results for all future searches, in the form of a lookup table. This is explored further in Chapter 8.

Application Domains and Related Work

The payoff-similarity property occurs often in trick-taking card games such as Hearts, Bridge, Sheepshead, and Skat. In these domains certain cards may become "power-equivalent", in terms of their ability to win tricks. For example, if one player is holding ♠78, then (from a perfect information perspective) playing ♠7 is equivalent to playing ♠8. Less obviously, we may transpose into a state where the relative ranks of each card are the same as in a previously seen state. In games such as Bridge where only the number of tricks made is important, positions may be converted into a "canonical" form, by relabeling card ranks. Indeed in the case of Bridge such a canonical form is the natural choice for transposition table indexing.

Recall that partition search (LWRS) constructs sets of states that are in the same equivalence class. Our framework is analogous to nearest-neighbour classification in that, instead of constructing an explicit representation of the sets, we implicitly define them in terms of previously seen states.

Rather than attempting to hand-craft complicated partitioning functions (as with partition search) we require only a similarity metric, which may be significantly easier to construct.

7.3 Application to Skat

Unlike in Bridge, where the value of a position depends only on how many tricks each player can make (and thus all states with the same canonical representation have the same value), the value of a Skat position also depends on the *value* of those cards. We can say that Skat and related card-games such as Sheepshead and Pinochle have the payoff-similar property in the non-trivial sense, where δ may be greater than 0.

Payoff-Similarity in Skat

In this section we will refer to the value of Skat positions in terms of the number of card points achievable by the *max* player (the soloist) minus the card points achievable by the *min* players (the defenders), assuming perfect information and that all players act optimally. Let $V(\cdot)$ be the corresponding state evaluation function.

Theorem 7.3.1. *Let s be a Skat position with card values $v_1, \ldots, v_k$. Let s' be the same position, except with card values $v'_i = v_i + \delta$ for exactly one i, and $v'_j = v_j$ for $j \neq i$. Then $|V(s') - V(s)| \leq |\delta|$.*

Proof. Consider the game trees T and T' rooted in s and s'. Then T and T' are structurally equivalent, because cards have only changed in value, not rank. For each corresponding leaf pair (l_i, l'_i) in T and T', $V(l'_i) = V(l_i) \pm \delta$, where the sign of the "$\pm$" depends on which team won the card with changed value. It follows that s and s' are payoff-similar and thus $|V(s) - V(s')| \leq |\delta|$ by Theorem 7.2.1. □

Corollary 7.3.2. *Let s be a Skat position with card values $v_1, \ldots, v_k$, and let s' be the same position with arbitrary card values $v'_1, \ldots, v'_k$. Then $|V(s) - V(s')| \leq \sum_{i=1}^{k} |v_i - v'_i|$.*

Proof. This follows immediately from Theorem 7.3.1 by applying it to each card value. □

We may use these results to construct a more generalized transposition table, indexed only by the canonical representation. That is, within each suit, we only care about which player owns the most powerful card, the next most powerful, etc., and not the values of those cards. Like a standard transposition table, the entries store lower and upper bounds for the value of a given position. Unlike

a standard transposition table we may have multiple entries for each index, with each entry annotated with the card values that were used to produce those bounds. A canonical transposition table lookup then consists of converting the current state into a canonical form, computing the corresponding index of that form, and then looping over all (card value) entries for that index.

This gives us a range $\langle \alpha^*, \beta^* \rangle$ within which the true value of the state provably lies, with $\alpha^* = \max_i(\alpha_i - \delta_i)$, and $\beta^* = \min_i(\beta_i + \delta_i)$, for each entry i. If this range is outside the current search window ($\beta^* \leq \alpha$ or $\beta \leq \alpha^*$) then we may immediately prune the state. Otherwise we can use it to tighten our α-β search window. If we ever encounter the exact same state (including card values) then $\delta_i = 0$ and the transposition table operates in the standard manner. Note however that because there may be multiple entries, the actual bounds returned may be tighter than those corresponding to the $\delta_i = 0$ entry.

As an aside, if we can determine when the team that acquires a particular card i does not change, then we can add δ_i to $V(s)$ directly, rather than just increasing the δ bound. In Skat, this can be seen to occur for the highest remaining trump card, and all subsequent (in rank) uninterrupted trump owned by the same team.

In this canonical transposition table regime we also have the property that if we ever encounter a move with value $v = \beta^*$ then we know that we have found the true value of the state (namely β^*), rather than just a lower bound. More precisely, evaluating the move provides a lower bound (as usual), but now we have an upper bound on the true value which agrees with our lower bound. Similarly, if we search all moves without triggering a β cut, then we are returning an upper bound. If this bound has value α^* then $V(s) = \alpha^*$. If we had just used $\langle \alpha^*, \beta^* \rangle$ to tighten our α-β window then this would not be apparent.

Experimental Results

To illustrate the effectiveness of payoff-similarity, we incorporated the described canonical indexing into our baseline C++ MTD(f) solver that computes the exact card-point score of an open-handed position. This solver is a high-performance program that uses hand-tuned move-ordering heuristics, but generally cannot collapse card ranks into a canonical form due to the differing values of cards within a suit.

Figure 7.1 shows the results of adding payoff-similarity to the baseline solver, with each data point averaged over 10,000 positions. Suit and grand games see a wall-clock time reduction of 85% and 77% respectively, for positions at height 30, the hardest positions to solve. The payoff-similarity

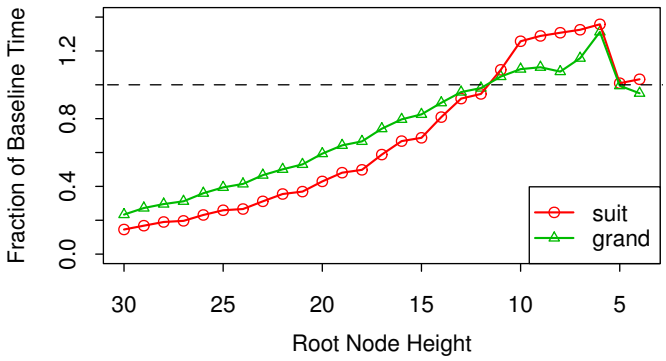

Figure 7.1: Skat α-β search times using payoff-similarity.

overhead does not outweigh the corresponding gains until the positions become essentially trivial (hands with 4 cards or fewer).

7.4 Estimating Perfect Information Solver Results

We have shown how the advanced search enhancements previously described enhance the speed of our perfect information solver without compromising correctness. However, although computing a single α-β result is a relatively fast operation, there are a number of inference techniques which would benefit from being able to compute more results than are currently practical with a modest number of commodity processors. To this end, there exists a time-accuracy trade-off, where a fast estimator may be used to replace either an entire DDS search, or, more generally, some subtree(s) within a search. If the estimated values are not too far from the exact values, then the value at the root node of the search must also necessarily be close to the true value. Since search is usually only needed to distinguish the relative quality of the possible moves, exact values are only necessary insofar as they can do so and an approximation cannot. Recall that for PIMC we need to compute the value of each successor state from the current position.

In our framework, value estimation can be viewed as a probabilistic analogue of bounded similarity. Instead of comparing our current state against a previously searched TT position, we have some function that tries to generalize across all positions (usually involving local features from a training set), and an associated empirical error distribution.

For highly tactical positions we expect that selecting moves using the values returned from α-β searches is in some sense the "right" thing to do. However this also assumes a level of coordination

between players which is usually not the case in most situations. If the estimated search values more accurately reflect the real imperfect information values, then the move selection process could be both faster and stronger when using estimated or heuristic values.

Linear Approximators

To estimate the result of an α-β search we trained several feature-based linear approximators, one for each combination of: game type (suit, grand), player to move (we assume that the soloist is in a fixed seat), and number of remaining cards in hand. Because positions with small numbers of cards are already fast to solve, we only constructed estimators for start-of-trick positions where all players have 6, 7, 8, 9, or 10 cards in their hand. Mid-trick positions have more variables to consider and would require additional memory, disk space, and training.

Each approximator works in the standard manner: given a collection of feature functions $\Phi = \{\phi_1, \ldots, \phi_k\}$, the estimated value of a state s is $\sum_{i=1}^{k} \beta_i[\phi_i(s)]$, where each ϕ_i returns an index (effectively partitioning the states), and $\beta_i[]$ is an array of learned values.

The β values are trained using least-squares linear regression, with the training set being hundreds of millions of randomly generated states, labeled with exactly computed DDS values. To avoid overfitting, we generated at least 20 positions corresponding to each β_i entry. That is, we sampled 20 positions from each $\phi_i^{-1}(j)$. Due to overlap and different partition sizes, β entries may have more than 20 training points.

Features

The majority of the features used are so-called "constellation" features. Given a list of cards (rank and suit), a constellation feature tracks the location of each card (i.e., the cross product). Each card may be in one of 4 locations: the hand of player 1, 2, or 3, or *out*. Since we are only trying to predict the number of points yet to be made by the declarer, we do not need to keep track of which team won each of the *out* cards. In the case of constellations restricted to a single player, each card may be in only two locations. We also use counting features such as the number of low cards each player has, and the number of trump each player has. A list of features used for suit and grand games is given in Tables 7.1 and 7.2.

The size of a constellation can be smaller than 4^k, for some k, due to certain symmetries. For instance, the Jacks and 789 cards may both be reduced to a canonical form, as all cards within a group have the same value. In the interest of storage space and training time, similarly-valued

Table 7.1: Features for suit games, assuming clubs is trump.

Suit Feature	Size
constellation of (Jacks + ♣A + ♠AT + ♡AT + ♢AT)	394,944
constellation of Aces + Tens	13,056
constellation of trump cards (with Ace-Ten and King-Queen equivalences)	817,960
player i non-trump Aces + Tens constellation $\times$	
player j trump constellation, $j \neq i$	(6x) 40,960
soloist (Jacks + ♢/♡/♠ATKQ) constellation $\times$	
number of ♢/♡/♠ each defender has $\times$	
if each defender has any trump	(3x) 36,864
constellation of each ♢/♡/♠ suit	(3x) 16,384
constellation of 789s over all suits	459,200
number of cards each player has in each non-trump suit	295,240
non-trump suits each player is void in $\times$	
number of ♢A♡A♠A each player has $\times$	
constellation of (Jacks + ♣A)	1,161,600
number of ♢/♡/♠ each player has $\times$	
constellation of Jacks $\times$	
number of ♣ATKQ987 each player has	1,742,400

Table 7.2: Features for grand games.

Grand Feature	Size
constellation of Jacks + Diamonds	1,239,040
constellation of Jacks + Hearts	1,239,040
constellation of Jacks + Spades	1,239,040
constellation of Jacks + Clubs	1,239,040
constellation of Jacks + Aces + Tens	468,996
number of cardpoints remaining	121

cards (such as Ace-Ten and King-Queen) are sometimes compressed into a canonical form as well. Finally, we may relabel non-trump suits (e.g., ♢♡♠ when playing a clubs game) into a canonical form.

Where applicable, such canonical relabelings are used to share feature weights. These features were chosen by hand, so as to fit within our memory budget and hopefully be predictive — we do not claim that they are optimal. Note that these features are also distinct from those used for the 10+2 bidding evaluator, since we are now concerned about predicted open-handed results where the location of all cards is known. The 10+2 evaluator, by contrast, evaluates only the soloist's own ten cards plus the two cards in the skat.

Table 7.3: Linear regression estimator statistics — one estimator per: game type, number of tricks played, and player to move. Two measures of accuracy are listed: the average absolute error and the standard deviation (in parentheses).

		suit game			grand		
NT	to move	LR	LR+SA	LR_{bdd}+SA	LR	LR+SA	LR_{bdd}+SA
0	Solo.	2.2 (3.1)	2.0 (3.0)	1.8 (2.9)	2.8 (3.9)	2.3 (3.6)	1.9 (3.3)
0	Def.1	3.5 (4.7)	3.2 (4.5)	2.9 (4.3)	4.1 (5.4)	3.7 (5.1)	3.2 (4.8)
0	Def.2	3.6 (4.7)	3.2 (4.5)	2.9 (4.3)	4.1 (5.7)	3.7 (6.0)	3.2 (5.8)
1	Solo.	2.4 (3.4)	2.1 (3.3)	1.9 (3.1)	2.7 (3.8)	2.2 (3.5)	1.7 (3.1)
1	Def.1	3.7 (4.9)	3.3 (4.7)	3.0 (4.4)	4.0 (5.3)	3.6 (5.0)	3.0 (4.6)
1	Def.2	3.7 (4.8)	3.3 (4.6)	2.9 (4.3)	4.0 (5.3)	3.5 (5.0)	3.0 (4.5)
2	Solo.	2.5 (3.5)	2.1 (3.3)	1.9 (3.1)	2.8 (3.9)	2.3 (3.6)	1.8 (3.3)
2	Def.1	3.7 (4.9)	3.3 (4.7)	2.9 (4.3)	4.1 (5.5)	3.7 (5.2)	3.1 (4.7)
2	Def.2	3.6 (4.8)	3.2 (4.6)	2.8 (4.2)	4.1 (5.4)	3.6 (5.2)	3.0 (4.7)
3	Solo.	2.6 (3.6)	2.2 (3.3)	1.8 (3.0)	2.9 (4.1)	2.3 (3.6)	1.8 (3.3)
3	Def.1	3.4 (4.5)	3.0 (4.3)	2.5 (3.9)	4.1 (5.5)	3.6 (5.3)	3.0 (4.7)
3	Def.2	3.3 (4.4)	2.9 (4.2)	2.5 (3.9)	4.1 (5.5)	3.6 (5.3)	3.0 (4.7)
4	Solo.	2.5 (3.4)	2.0 (3.1)	1.6 (2.9)	3.0 (4.2)	2.3 (3.7)	1.7 (3.3)
4	Def.1	3.0 (4.0)	2.6 (3.8)	2.3 (3.6)	3.9 (5.3)	3.3 (5.0)	2.8 (4.6)
4	Def.2	2.9 (4.0)	2.6 (3.8)	2.2 (3.6)	3.9 (5.3)	3.4 (5.0)	2.8 (4.5)

Accuracy

Due to our aforementioned static analysis function we can (and do) check if the value returned by our linear estimator is wildly inaccurate. We can also cap the approximation error of each data point *during* the linear regression, instead of just using $\hat{y} - y$. Thus, we can help prevent the regression from "chasing" a data point where the estimate is very inaccurate.

Because our linear regression is performed via gradient descent, capping the error of a training point does not produce a plateau where we don't care about better approximating that point — rather, it caps the contribution of that data point to the slope of the regression. The regression will still attempt to correct for these errors, but it won't try as hard.

Table 7.3 shows the training error for several data sets, with and without using error capping during regression (LR_{bdd} and LR, respectively). We can see that using this method results in a significantly lower average absolute error and standard deviation. Although not shown, the errors for LR_{bdd} with static analysis disabled are significantly greater than for LR.

Pruning

The ProbCut algorithm [3] works by correlating the result of a shallow search with the value returned by a deeper search. That is, we can experimentally determine the expected value and variance of $v_{\text{shallow}} - v_{\text{deep}}$. Then, by performing a shallow search, we can form a window of arbitrary confidence within which we expect the value of a deep search to lie. If this window is outside the current α-β bounds then we can immediately prune, saving the effort of a deep search for the price of a small probability of error. If time permits, one may re-search a position with a wider window to increase confidence in the result.

This technique was applied successfully to Othello, where solving positions exactly is intractable. However in our regime the cost of exactly solving a position is much lower. So much so that sophisticated incremental search techniques have not received much attention, largely due to effective pruning and move ordering heuristics that reduce the effective average branching factor to less than 1.65. A result of this low branching factor is that the overhead of incremental search techniques such as iterative deepening tends to outweigh any gains. Moreover, since we are using the PIMC algorithm there are many open-handed positions that we wish to solve, rather than solving one position with increasing confidence. That said, we may still select a ProbCut-style confidence threshold a priori, to use for searching many worlds.

Indeed, by taking the empirical standard deviation computed during the linear regression and combining it with our confidence threshold, we can form a window around the values returned by our estimators, within which we expect the true position value to lie. We do not use this window to tighten our the current α-β bounds (although we could), we only check if it is outside our cutoff bounds — if it is outside then we cut, otherwise we continue searching as normal.

Experimental Results

To measure the speed/error trade-offs of using ProbCut-style linear approximators we modify a baseline α-β implementation. This baseline solver is highly tuned and based on MTD(f). It does not use any lookup tables. Results are shown in Figures 7.2 and 7.3. The listed error is given in terms of the expected absolute difference between the α-β value produced with and without using ProbCut. Note that these error numbers are deceptively small, as the effect of evaluation errors on move selection is amplified, as will be seen shortly. The "window" parameter used in the figures refers to the size of the α-β bounds used by MTD(f). Normally this window is 1, causing MTD(f) to perform a series of zero-window searches. A larger window can result in fewer of these probes,

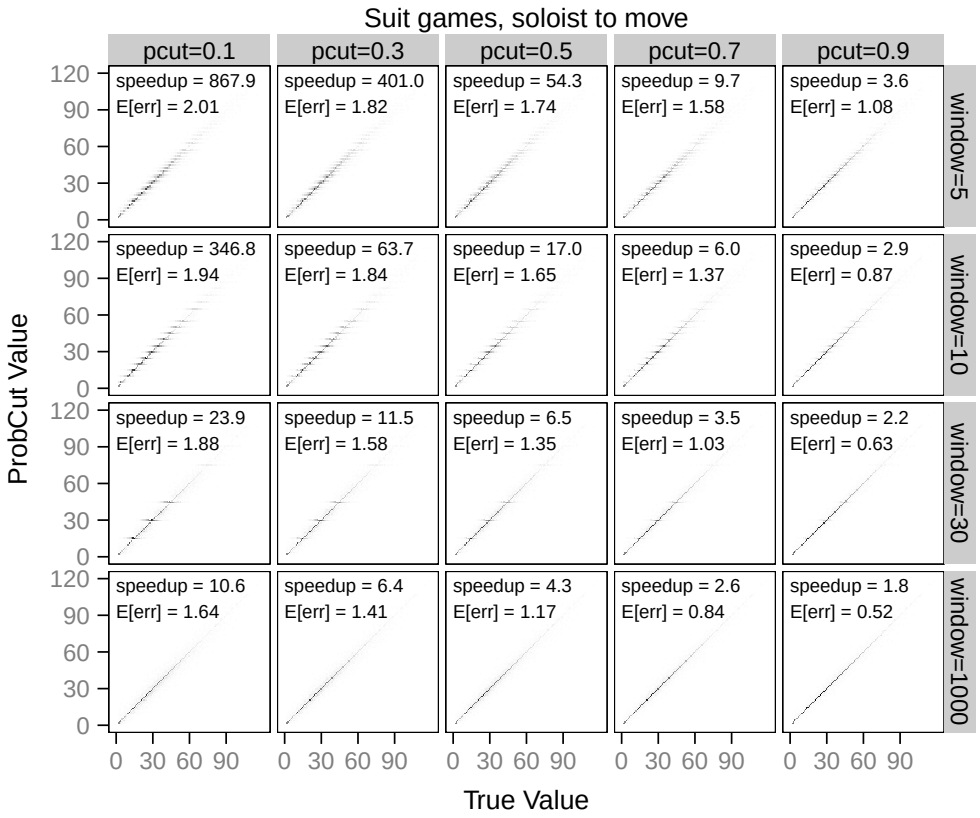

Figure 7.2: Speedup and evaluation errors from using ProbCut on suit games. ProbCut thresholds are given as "pcut", and "window" refers to the size of the α-β window used by MTD(f).

but each individual search tends to be slower.

The presence of zero-window searches causes our implementation of ProbCut to trigger many cuts at the start of search, essentially "snapping" to the boundaries of the window. Compare this to a wide window, which causes search to follow the leftmost branch until a leaf node is reached. If our move ordering heuristic is good, then this value is a good reflection of the true value. Subsequent MTD(f) iterations will then tend to snap to this better value when using aggressive cutting thresholds.

To see how the listed errors relate to tournament performance, we turn to Figures 7.4 and 7.5, which show the results of actual tournament games played by Kermit, using different ProbCut thresholds for both the soloist and the defenders. In all cases the number of worlds sampled by Kermit was 160. The values listed are the sample means of the soloist scores, which have standard errors of about 0.30 and 0.36 for suit and grand games respectively. Results were calculated using 18,962 Kermit-bid games, with the same thresholds used for both suit and grand games.

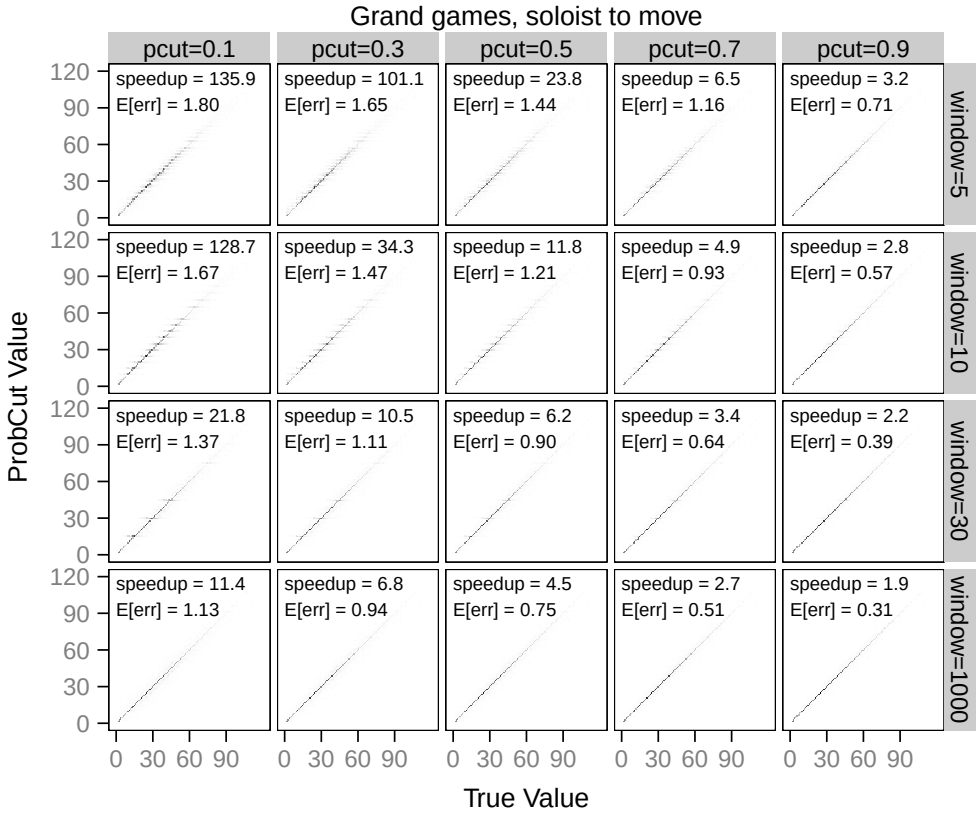

Figure 7.3: Speedup and evaluation errors from using ProbCut on grand games.

Because the same deals were used throughout each table, the significance of differences between neighbouring rows/columns are stronger than the confidence intervals may initially suggest. In Figures 7.6 and 7.7 we present graphs of whether one ProbCut threshold is statistically significantly better than another, at a 95% confidence level. Values are computed using two-tailed paired tests. The multiple bars within each cell correspond to the different types of opposing player — e.g., when comparing two soloists, $K^{0.5}_{160}$ and $K^{0.8}_{160}$, the defenders can be any of the 10 Kermit-ProbCut players tested.

The rightmost column (either orientation) clearly indicates that the apparent weakness of a 0.1 threshold player is not an illusion, such a player is statistically worse than all of the other thresholds tested, when playing as soloist. When playing as defender, the weaknesses of using ProbCut appear less pronounced overall. Prior to the construction of this graph we had selected a threshold of 0.5 for tournament usage, as this was deemed to be statistically indistinguishable from and notably faster

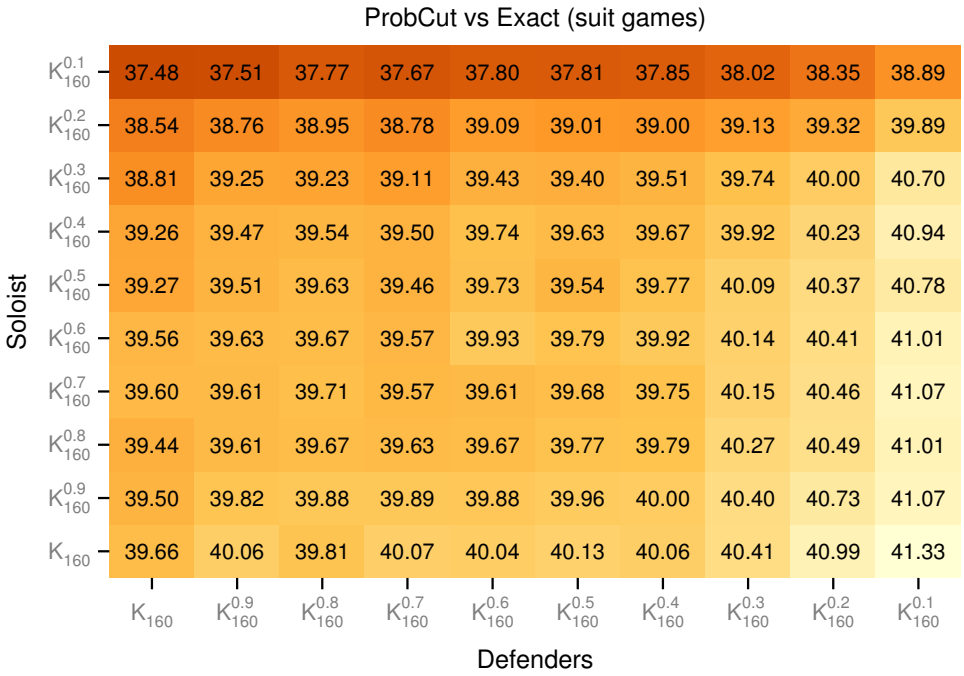

Figure 7.4: Suit game tournament results when using ProbCut within Kermit.

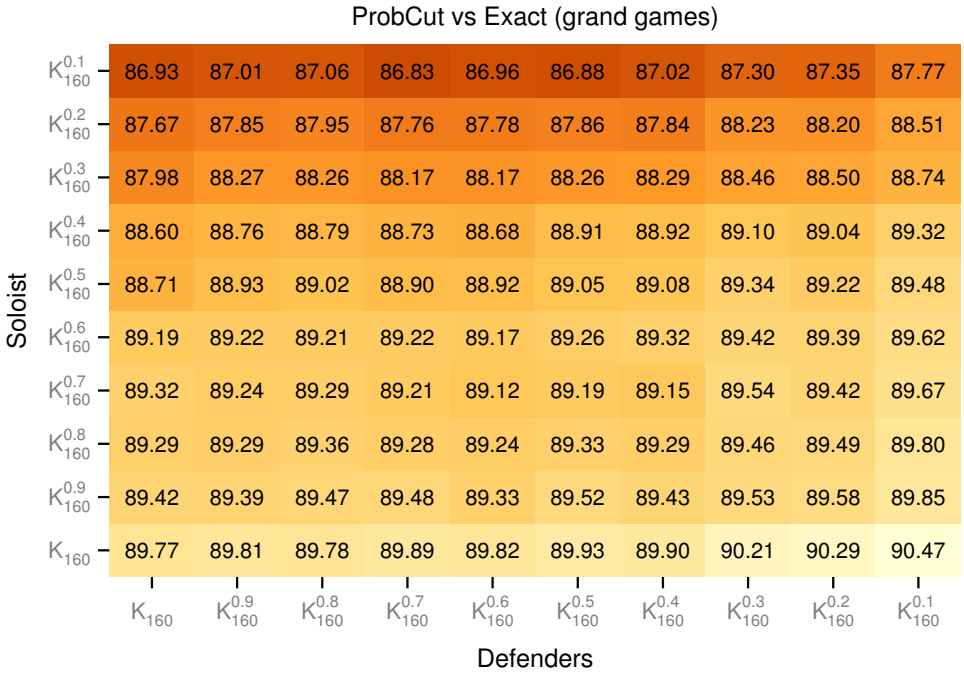

Figure 7.5: Grand game tournament results when using ProbCut within Kermit.

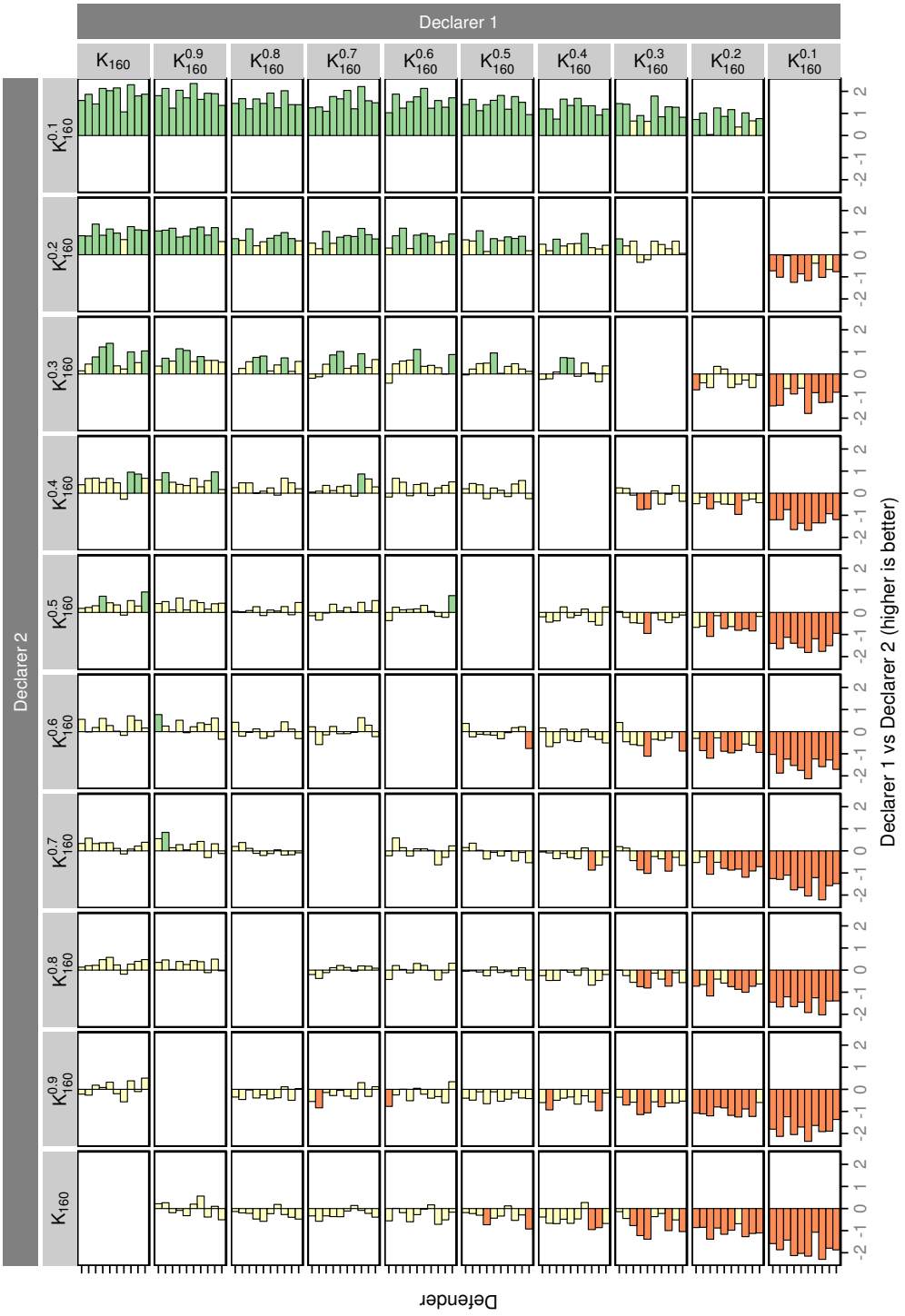

Figure 7.6: Graph of whether changes in soloist score when using different ProbCut thresholds are statistically significant.

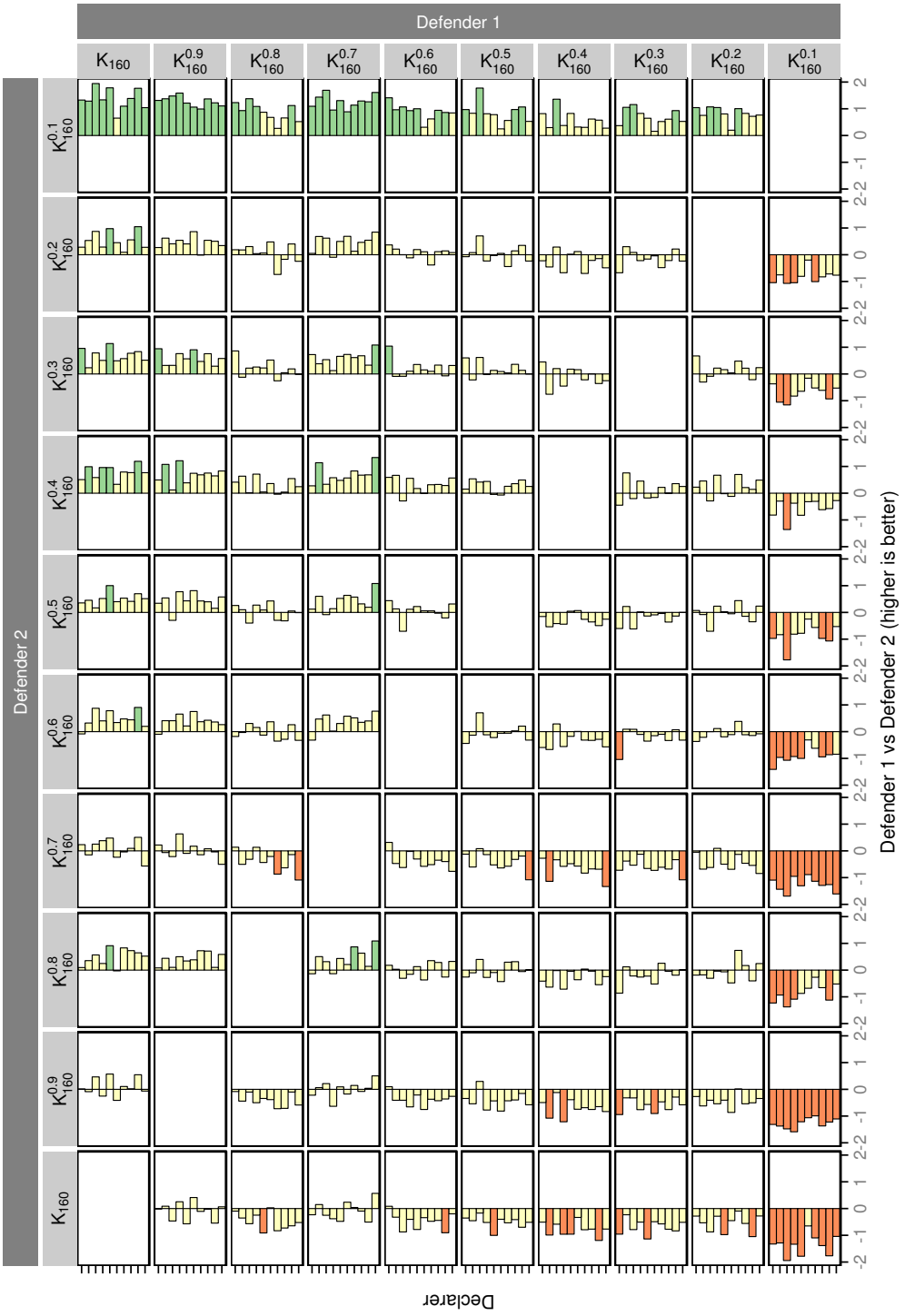

Figure 7.7: Graph of whether changes in defender score when using different ProbCut thresholds are statistically significant.

than Kermit. These recent results seem to validate that decision, although an argument could be made for being more conservative in light of the small number of $K^{0.5}_{160}$ vs K_{160} matches in which using ProbCut results in slightly worse performance.

7.5 Combining with Partition Search

Having introduced the concept of payoff similarity we are now in a position to consider combining this technique with the lowest-winning-rank symmetries from Chapter 4. The canonical symmetries we introduced in this chapter operate because the tree structure is the same if the relative ranks are the same. The LWR property tells us that the tree structure is also the same if the low ranks are rearranged. All that remains is to apply the payoff similarity property to each players' cards. Note that the rank rearrangement property implies that we can rearrange a particular player's low cards so as to improve our δ bound, however this does not let us safely exchange low cards between players.

We have not yet implemented this hybrid approach for Skat, but it promises to be an interesting line of investigation.

7.6 Conclusions and Future Work

In this chapter we have introduced the concept of payoff-similarity of game states which can be used to find exact bounds for minimax values of related states. Incorporating this technique into an already fast exact double dummy solver for Skat in conjunction with pre-computed lookup tables has shown speed-up factors of 10 and more in the beginning of the game. In a second step, we equipped our solver with a linear state value predictor to prune likely irrelevant parts of the search tree using a method similar to ProbCut. This addition increased the speed by another factor of 4 without losing playing strength, which is remarkable and worth future investigations into how forward-pruning for perfect information games performs in imperfect information games when using PIMC search. Moreover, with these speedups, state inference based on PIMC search is now becoming feasible in Skat. With this we expect further increases in the playing strength of Skat programs.

The concept of payoff-similarity may also be applicable to single-agent domains with changing operator costs, in which we could try to increase the value of search heuristics by querying values of related states.

Chapter 8

Constructing Midgame Lookup Tables

> Why should a fall of sand from a clifftop stick in the memory? she asked herself.
>
> *Dune*
> FRANK HERBERT

8.1 Introduction

Because the number of canonical Skat positions is sufficiently small, it becomes practical to precompute transposition table entries ahead of time, so that they may be used for all future searches. The effectiveness of this precomputation is ultimately determined by whether the reduction in search nodes is enough to offset the additional time required to query the resulting lookup table of entries. The reduction in search nodes is affected by two factors: the quality of the bounds returned, and the height within the search tree at which the entries occur. We would like to find an optimal balance between storing entries sufficiently high in the search tree, and having a sufficiently diverse collection of associated card values so that the returned bounds are non-trivial.

Lookup Table Structure

A midgame Skat lookup table (endgame database) for a suit or grand game consist of a number of entries indexed by three components: *split*, *ownership*, and *valuation*. The split refers to the number of cards remaining in each suit, without regard to whose hand they are in or the number of card points they are worth. The ownership refers to one of the $\binom{3k}{k,k,k}$ ways of assigning those cards to the three players. The split and ownership together define a unique canonical state. The valuation is an assignment of card values to individual cards (canonical ranks). Canonical states with different valuations are payoff-similar. Although the number of ownership assignments is independent of the split, the number of valuations is not.

For our purposes the split represents a hard partitioning of the table entries — we will not attempt to combine entries from different splits or to otherwise cleverly relate their values so as to improve

Table 8.1: Number of splits and valuations for suit games.

# of cards per hand	suit splits	$\lvert\mathcal{O}\rvert = \binom{3k}{k,k,k}$	# of valuations (over all splits)	$\lvert\mathcal{O}\rvert \times \sum_i \lvert\mathcal{V}_i\rvert$
1	7	6	378	2,268
2	23	90	16,898	1,520,820
3	50	1,680	200,146	336,245,280
4	79	34,650	870,300	30,155,895,000
5	97	756,756	1,647,718	1,246,920,482,808
6	93	17,153,136	1,452,283	24,911,207,809,488
7	70	399,072,960	588,274	234,764,246,471,040
8	40	9,465,511,770	99,362	940,512,180,490,740
9	16	227,873,431,500	5,641	1,285,434,027,091,500
10	4	5,550,996,791,340	73	405,222,765,767,820

Table 8.2: Number of splits and valuations for grand games.

# of cards per hand	suit splits	$\lvert\mathcal{O}\rvert = \binom{3k}{k,k,k}$	# of valuations (over all splits)	$\lvert\mathcal{O}\rvert \times \sum_i \lvert\mathcal{V}_i\rvert$
1	7	6	137	822
2	25	90	4,909	441,810
3	56	1,680	53,892	90,538,560
4	92	34,650	228,208	7,907,407,200
5	113	756,756	428,129	323,989,189,524
6	109	17,153,136	378,044	6,484,640,145,984
7	80	399,072,960	155,173	61,925,348,422,080
8	45	9,465,511,770	27,231	257,755,351,008,870
9	17	227,873,431,500	1,718	391,486,555,317,000
10	4	5,550,996,791,340	32	177,631,897,322,880

speed or compression. Consider a particular suit split. Let $\mathcal{O}$ be the set of possible ownership assignments, and let $\mathcal{V}$ be the set of possible card valuations.[1] The full lookup table (for this suit split) consists of the minimax values corresponding to the elements of $\mathcal{O} \times \mathcal{V}$. The full tables for 5 or more cards per hand are too big to naively store in-memory on current commodity hardware, as can be seen in Tables 8.1 and 8.2.

8.2 Selecting a Subset

Given a budget (usually this is in terms of memory usage, but possibly disk), our task is to select a subset of the table data such that we minimize our expected search time when using that subset.

[1] $\mathcal{V}$ is the cross product of the possible valuations for each individual suit after applying symmetry reductions. Thus $\lvert\mathcal{V}\rvert \leq \prod_{s \in \diamondsuit\heartsuit\spadesuit\clubsuit} \lvert\mathcal{V}_s\rvert$.

Put another way, we want to select the portion of the full table that causes the most alpha-beta cuts, either because a query returns an exact value or because the bounds are sufficient to trigger a cut. As such, we prefer to store table entries for subtrees that are both frequently occurring (across different searches, not multiple times within the same search) and expensive to compute. The difficulty arises in determining which entries satisfy this property. Note that the search time depends greatly on unpredictable factors such as the positions previously searched (and thus already in the normal transposition table). In general, the proof trees for two (or more) table entries may share a common (expensive) subtree. If we would naturally encounter these entries during search, then to avoid searching the expensive subtree those entries must all be in our lookup table.

We can see in Figure 8.1 that the greatest node reductions are achieved when the lookup tables return an exact value. Returning even a small (2 point) range of values results in notably more search effort compared to returning an exact value. To generate the data for Figure 8.1 we simply call a black box oracle when we would normally query the lookup tables. The oracle computes the exact value of a position, and then returns a range by adding and subtracting the given upper-/lower-bound tolerances. The oracle is simply a standard search, but the node expansions do not count towards the main algorithm's totals.

Our discussion of accuracy and node expansions has so far ignored the higher-level question of how much accuracy is actually necessary. It may be the case that we are satisfied with a small margin of error at the root, in which case we would be willing to tolerate errors that are within a small distance of the true value (probabalistically or otherwise). To achieve the node count reductions of exact evaluations we could simply return an arbitrary value within the computed range. Throughout this chapter we will however restrict ourselves to only using lookup tables in a "safe" manner, with the unsafe extensions being obvious to implement. We assume that each query returns a pair of true upper and lower bounds for a given position, necessarily in the range $[-120, 120]$, although normally restricted further by static analysis.

When selecting subsets from the full table, we may either select the same valuations for all ownership rows, or adapt our selection based on the particular row, as in Figure 8.2. In our previous work [15] we used the first method, which required only a single byte for each table entry (the minimax value), since the indexing scheme was thus implicit for each split's table.

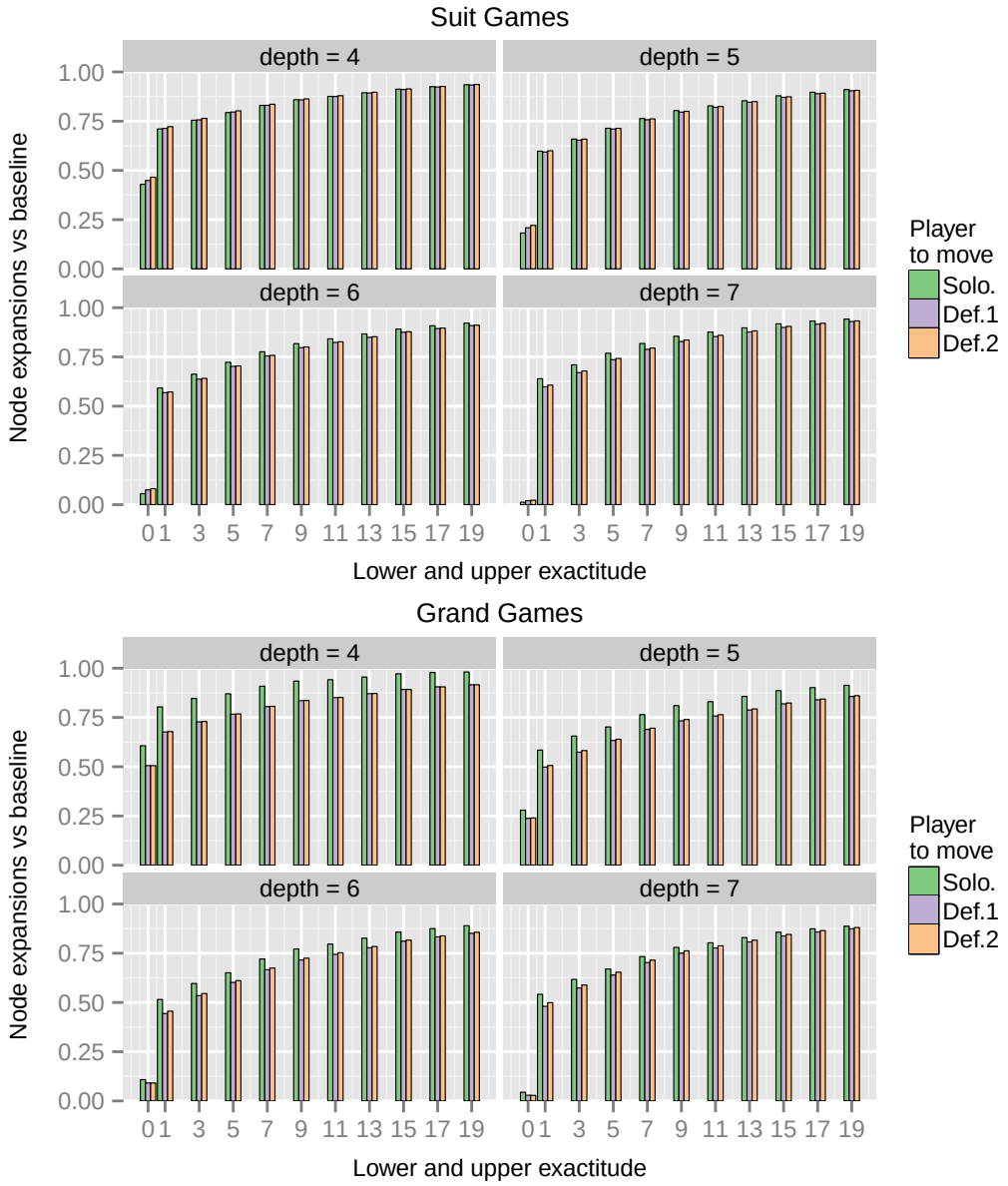

Figure 8.1: Node reductions when using lookup tables of various accuracies. The bounds returned are the true state value plus or minus the "exactitude". Sampled over 2000 initial positions.

Query Accuracy

In that same work [15] we attempted to minimize the expected value of $\delta = \min_i \delta_i$, the distance to the closest precomputed point, with the intuition being that this would result in many "good" lookups with small bounds. This intuition is misleading for the following reason: exact bounds

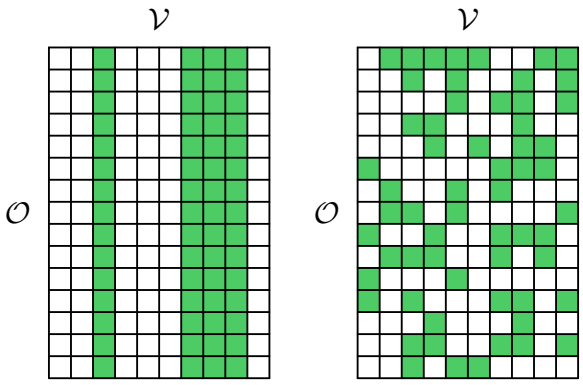

Figure 8.2: Selecting a subset of entries from the full ownership-valuation table. On the left, the same valuations are chosen for all rows. On the right, different rows optimize their choice of valuations independently.

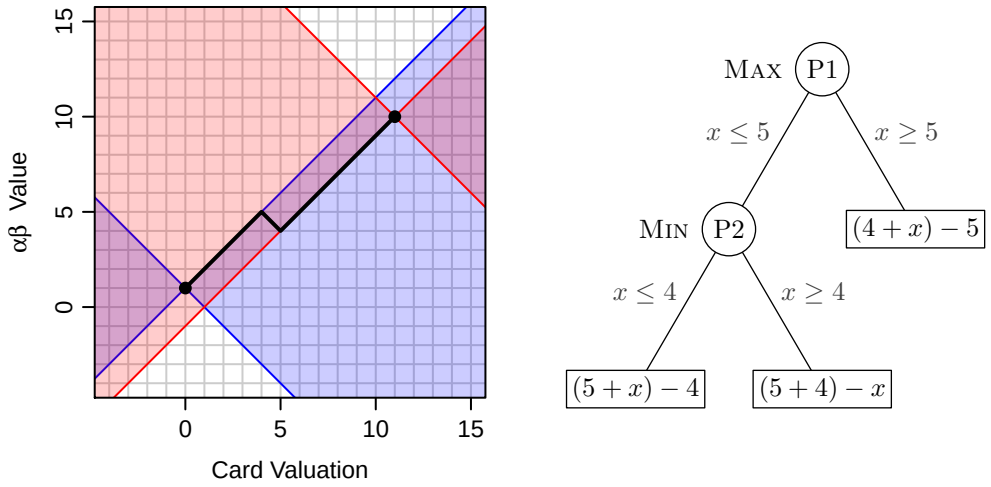

Figure 8.3: 1-dimensional example of payoff-similarity bounds. One card has variable value x. On the left, two known data points are shown, with shaded regions depicting the space of legal minimax values each point induces. The dark line is the true function. The game tree is shown on the right, along with each player's optimal action, given x.

may be obtained even for arbitrarily large values of δ. Consider the example in Figure 8.3. In this example, two data points at the extreme ends of the value-function's domain restrict the range of possible values to a relatively narrow region. If $|V(s) - V(s')| = \delta$, then all values "in-between" the two points would be known exactly. This observation extends to the multi-dimensional case, where more than one card value may vary.

We will now formalize some aspects of this observation. Let $\delta(x, y) = \sum_i |x_i - y_i|$ be a distance

metric on card valuations (L_1 distance), and let $d(x,y) = |V(x) - V(y)|$ be a distance metric on minimax values associated with those valuations (assuming some particular ownership and suit split). In general $\delta(\cdot,\cdot)$ is any upper bound such that $\delta(x,y) \geq d(x,y)$ — and in particular this holds for Skat, with δ and d as defined, due to the payoff similarity property.

Define a **hard edge** to be a pair of valuations $\langle x,y \rangle$ such that $d(x,y) = \delta(x,y)$, with $x \neq y$. This corresponds to having all points "between" x and y (within the hypercube those two points define) being exactly inferable from the two minimax values $V(x)$ and $V(y)$. For reasons which will become apparent, consider all hard edges to be directed edges, going from (w.l.o.g.) low minimax values to high minimax values.

Proposition 8.2.1. *Hard edges are transitive.*

Proof. Let x,y,z be valuations such that $\langle x,y \rangle$ and $\langle y,z \rangle$ are hard edges, with $V(x) < V(y) < V(z)$.

$$\begin{aligned} d(x,z) &= V(z) - V(x) \\ &= V(z) - V(y) + V(y) - V(x) \\ &= d(y,z) + d(x,y) \\ &= \delta(y,z) + \delta(x,y) \\ &\geq \delta(x,z) \end{aligned}$$

But $d(x,z) \leq \delta(x,z)$, so $d(x,z) = \delta(x,z)$, and $\langle x,z \rangle$ is a hard edge. □

Now consider the graph consisting of all hard edges. This graph consists of some number of source and sink nodes. Any node that is neither a source nor a sink must lie on an optimal path between a source and a sink, due to transitivity. These intermediate points are redundant, since their predictive powers are strictly subsumed by the sources and sinks, and may be removed in a preprocessing step.

8.3 Sparse Lookup Tables

We shall refer to the minimum subset of the full table that is capable of reconstructing all points as a *sparse table*. The points in this subset are in some sense "tentpoles" that describe the curvature of the minimax value hypersurface. The tentpoles represent points of inflection — the players' optimal policies depend on which "side" of a tentpole each card valuation lies. The value surface in

Figure 8.3 has four such tentpoles. Along each card value axis, when a card value becomes larger than the tentpole it becomes sufficiently valuable that players prefer to acquire it at the expense of certain other cards, whereas for smaller values this trade-off is not worthwhile. If we are interested in maximizing the number of points which are represented exactly, for a given data-point budget, then these tentpoles are the most efficient points to select.

Construction

Before we optimize our lookup table subset, we must first construct each sparse table. Although one could compute a full table and then remove redundant points in a post-processing step (via tentpole transitivity), we present a more efficient approach. Given a particular suit split and ownership row to compute the tentpoles of, start with an arbitrary (legal) valuation index within that row. Now look at the local neighbourhood of valuations around that point — points that can be reached by changing one card value — and see which (if any) of those neighbours correspond to hard edges. From the starting point, in both directions (increasing and decreasing minimax value) repeatedly walk along hard edges until a local maximum/minimum is reached. Both extrema points reached in this manner are tentpoles and must be included in the sparse table. The implied valuations "spanned" by these two tentpoles may then be identified and marked as redundant (i.e. the hypercube the valuations define). In this manner all valuations in a row may be checked, skipping over known redundant points.

We only care about identifying and computing distinct ownership rows, since duplicate rows from the same suit split may be merged in the lookup table. In the case where the current row is identical to a previously seen row, we would prefer to perform only the minimal amount of work necessary to verify which row it is identical to. In the case where the current row is unique, we will follow the previous approach of walking along hard edges. To check if a new row (where all minimax values are unknown) is identical to a previously seen row, it is sufficient to check if the tentpoles of the two rows match. Recall that the tentpoles of a sparse row are sufficient to define all row entries from the full table. Thus if the minimax values of the tentpoles in the old row match those in the new row, then the edges must also be hard edges, and both rows must be identical.

The naive approach to checking a potential duplicate row is to loop over all previously seen rows and evaluate the old tentpoles on the new row, computing minimax values as needed. This may be improved somewhat by using less costly narrow-window searches to refute or confirm that a minimax value matches. Rather than checking all entries, we take the approach of building a trie-like

structure. In this trie, the "characters" are tentpoles, and backtracking is performed in the event of a mismatch. This allows us to avoid repeatedly testing tentpole "prefixes" shared by multiple rows. New unique rows are added to the structure, with leaf values storing their canonical row index.

Pitfall

The following pitfall is both easy to overlook and easy to understand. It arises from erroneously simplifying duplicate ownership rows. When we simplify ownership rows we remove all points which are not tentpoles, since the tentpoles induce all points. However static analysis bounds may serve a similar purpose — proving accurate upper and lower bounds which might otherwise require a tentpole.

In both the general case and with respect to our particular flavour of static analysis, the bounds produced by static analysis can depend on the ownership of the cards. Consider two ownership rows. In one row, the principal variation depends on transferring control between the defenders (the particulars of which are not captured by static analysis). In the other row, the high-power cards are located in only one defender's hand, such that no control transfer is necessary, and which is correctly evaluated by static analysis. If these two rows are identical — having the same minimax value at each particular card valuation — then we prefer to store only one representative (canonical) row in our lookup tables.

As the reader may guess by this point, there is a danger that the static analysis bounds for the canonical row cannot be reproduced for one or more of the rows it is representing. Thus, if we eliminate certain tentpoles from the canonical row under the belief that static analysis renders them redundant, this belief may be violated when actually querying our lookup table, if the row we are querying has weaker bounds. The simplest solution is to not use static analysis to remove tentpoles, and is the approach we adopted. One could also consider only using those static analysis bounds that are applicable to all of the rows in question.

8.4 Storing Tables

Having generated the sparse tables and built our lookup tables, we must now represent those lookup tables in memory or on disk. Since the tentpoles of different rows tend to occur at different valuation indices, we cannot efficiently use an implicit indexing scheme. Instead the rows are stored as a contiguous list, with each list entry consisting of both the valuation index and the minimax value. Each entry thus uses 4 bytes of storage: 18 bits for the index, 8 bits for the value, and 1 bit for an

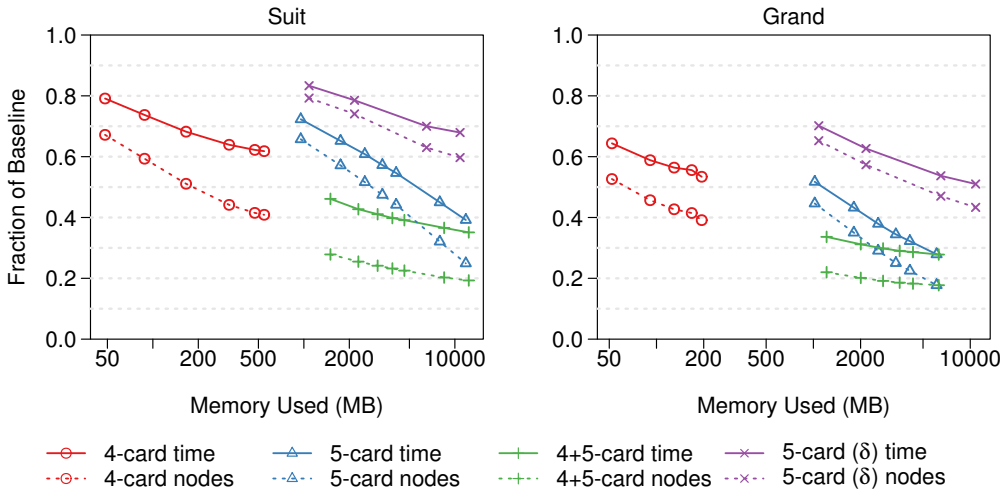

Figure 8.4: Reduction in node expansions and CPU-cycles when using precomputed lookup tables of various sizes. Results are shown for 4- and 5-card "sparse" tables, and for the previous 5-card tables that try to minimize the expected valuation δ of each lookup.

end-of-row flag. Although the per-entry storage requirements are 4 times greater than our previous scheme, we find that the node-reduction quality of the resulting lookup table overcomes this, as seen in Figure 8.4. Note that the lookup tables which use tentpoles substantially dominate the previous approach in terms of their performance/space trade-off. These lookup tables had their entries selected using a simple greedy heuristic that iteratively removed tentpoles while maximizing the number of entries represented exactly. Table rows were weighted according to the number of times that they occur within the full table. A more detailed discussion of optimizing this entry-selection process is postponed until section 8.5. Results shown are for solving start-of-cardplay positions, averaged over 10,000 random deals. The baseline solver uses payoff-similarity and no precomputed lookup tables.

Indexing

To store a pointer to the start of each row, we construct a ternary decision diagram (DD), where each node branches on the ownership of a particular card. The leaf values of this diagram are the corresponding row pointers. Because minimax values tend to depend more on the ownership of the higher-ranked cards, the decision diagram nodes are ordered first by trump suit, then by longest suit, and then by rank within each suit. Intuitively one might expect that the ownership of certain low-power cards would not affect the minimax values, since those cards will likely not win tricks.

Thus entire subtrees tend to point to the same row, allowing those subtrees to be collapsed. We experimented with computing the optimal DD variable ordering (to minimize the DD size), but found little practical difference to justify the computation.

Rather than using a decision diagram, we could have used a simple offset table for each ownership index. The offset table is about 1% faster (relative to the baseline) than the DD indexing used in Figure 8.4, but uses an additional 550–700 MB of memory for the 5-card tables, depending on game type and table size (larger tables have fewer identical rows, so the DD compression is less effective). Note that the index of each lookup table is hundreds of megabytes for the 5-card tables, and so those tables cannot be made much smaller than what is shown in Figure 8.4. We were however able to store in memory the entire sparse lookup tables for 4 cards, and 5 cards for grand.

Using the complete 4-card lookup tables results in wall-clock time reductions of 38% and 47% for suit and grand games respectively, at a cost of 200 MB and 550 MB. The reduction in nodes searched is more impressive, at 59% and 61%; the difference being the result of overhead from computing the row index offset and having to examine multiple entries. Even partial 4-card tables using only 50 MB for each suit cause time reductions of 21% and 35%.

In cases where the complete 5-card table cannot fit within a given memory budget, a hybrid approach using both the complete 4-card table and a partial 5-card table outperforms using just the partial 5-card table by a significant margin. Naturally this gap decreases as the partial 5-card table becomes more complete. As mentioned, our new approach dominates the lookup tables we previously constructed [15], with the resulting solver being approximately twice as fast.

8.5 Optimizing Table Entries

A cost function over (full) lookup table entries is a heuristic measure of how important it is to represent each table entry accurately. Equivalently it is the amount of regret or damage we suffer from not accurately representing a point. Define the cost curve of a table row to be the amount of regret suffered with respect to a given tentpole budget. Thus cost curves are monotonically decreasing (not necessarily strictly).

Define the zero-one regret as the ability of a sparse table to exactly represent a particular data point. Based on our previous observation regarding node reductions and lookup table accuracies, this is the class of regret functions we will be considering. More complicated costs functions can be constructed that depend on the range of values a point is restricted to. Using such cost functions would actually not be a terrible imposition, as this would only affect the cost curves of each row.

The majority of the optimization process discussed herein would not be affected.

Throughout the following discussion it is tempting to focus on achieving the optimal values for our cost functions. It is important to remember that the functions used are ultimately heuristic proxies that we hope will be correlated with what we are actually trying to achieve — a reduction in wall-clock time by reducing the number of nodes searched. It will be the case that some seemingly superior cost functions will produce corresponding lookup tables of inferior quality. Indeed there are many such functions which we have not even considered. That said, if we do stumble upon a good heuristic, we should probably know how to optimize it.

Half Rows

Before we proceed, we need to make one further observation. In the case of sparse rows which are represented exactly, we have two sets of tentpoles: the sources and the sinks. The low-value tentpoles (sources) provide useful upper bounds, and the high-value tentpoles (sinks) provide useful lower bounds. But the bounds from the low tentpoles only serve to confirm that the bounds from the high tentpoles are exact (and vice versa). If we know that we are representing a row exactly then we can simply mark all entries as being high or low tentpoles and treat the final lower or upper bounds as exact.

This optimization allows us to remove at least half of all source-sink tentpole pairs. Note that some tentpoles may be both high and low, if they do not form a hard edge with another point. We can always choose the larger of the two redundant sets to remove. In some cases this can result in requiring much less than half of the original space. We term such rows "half rows".

Optimizing Rows

Because half rows necessarily represent the entire row, they are easy to compute — simply remove the larger of the two redundant halves. It is straightforward to take a "normal" cost curve for a row, augment it with the data point corresponding to a half row allocation, and remove any dominated points. The following discussion relates only to optimizing the curves for non-half row entries.

Since pairs of tentpoles induce several table entries (valuations within a row), we first attempt to compute the cost curves for each individual row. That is, for each row and each $k \in \{1, \ldots, n\}$, we wish to compute the optimal subset of k tentpoles, and their associated utility (equivalently, the cost of those entries which are not accurately represented). Because the number of tentpoles may be in the hundreds, it is infeasible to naively enumerate all possible subsets. Thus, we start with

all tentpoles, and then greedily remove the tentpole whose removal causes the least damage. Sadly, even the greedy optimization is expensive, despite efficiently recomputing the removal cost at each iteration. The issue here is not so much that optimizing a single row is prohibitive, but that there may be hundreds of millions of rows. We leave the problem of computing optimal costs curves for each row as future work.

Because tentpoles only tend to be useful in pairs, removing one may drastically reduce the utility of its partner(s). A single tentpole removal will affect almost all points in its span(s), causing them to no longer be well-represented. Thus removing the second tentpole in a pair can often be done for little cost. Local search with a small (2 step) lookahead should be able to overcome most of the local minima arising from this pair-removal phenomenon, and would be a compromise between computing optimal cost curves and our greedy approach.

As mentioned, we only consider whether a point is represented exactly or not. Our initial experiments also looked at linear costs relating to the exactitude of each point, but the resulting lookup tables were notably inferior. However, a more informed interpolation cost should in principle be able to produce better results than the zero-one measure.

Clearly the selection of tentpoles at each step depends heavily on the utility function we choose when evaluating each point. The most natural and/or naive weighting function is to assign each point a value of one. An obvious alternative is to weight points by the frequency of their row. Because several rows within a lookup table may be identical, we only need to store a single representative. Using row-frequency weighting "corrects" for this compression. Another (orthogonal) alternative is to weight each valuation column by the number of ways it can occur. For example, because the 7s, 8s, and 9s all have value zero, and the jacks all have value two, there may be multiple card assignments that result in the same valuation. Our attempts to incorporate some measure of predicted search effort into the utility estimates of table entries have all failed (e.g., weighting table entries by the amount of work needed by α-β to evaluate them).

Greedy Table Optimization

Once we have computed the cost curves for each row, we must assign a budget to each row — the number of tentpoles each row may keep. Our first method of doing this is greedy tentpole removal, identical to the row optimization just presented, but acting over all rows simultaneously. This is easily and efficiently accomplished in an online manner with a stream heap — a regular heap where the entries are vectors (streams) of elements, and the value of an entry is the first element of each

stream (c.f. merge sort with a k-way heap).

In particular, we read in each row one at a time and add it to the stream heap, while keeping track of the number of elements currently held. If this number ever exceeds our table budget, we repeatedly pop elements off of the smallest stream until we are within budget. Streams which have all of their elements removed are simply discarded from the heap. After we have finished processing all of the rows, our final budget allocation is just the number of elements each row has remaining in the heap. In this case each stream's values are the *damage* caused by reducing that stream's budget by one.

Dynamic Programming

To compute the optimal budget allocation we must turn to some flavour of dynamic programming. We first note that computing the optimal error curve over any group of rows only requires space proportional to the maximum budget. Given two error curves, a and b, the combined error curve c is defined as $c_k := \min_{i \geq 0, j \geq 0, i+j=k} a_i + b_j$. Thus a naive merging algorithm will take quadratic time.

All error curves are necessarily monotonic, but this is insufficient to improve our efficiency — adding them together may result in arbitrarily many local minima. However, in our case (as will be seen), the curves being considered are in fact convex (or nearly so). Conceptually, computing the error for a budget i can be seen as adding curve a to a horizontally mirrored curve b, where the amount of overlap is equal to i.

Adding two convex curves in this manner preserves convexity, and finding a global minima is an efficient $((\log n))$ operation. Because our curves need not be exactly convex, we instead compute a convex lower bound for each curve — this is simply its convex hull, which can be computed in linear time. This lower bound allows us to efficiently prune large portions of the search space.

Our algorithm starts searching from the heuristic minimum provided by the convex bounds. We then scan left and right until the convex bound meets or exceeds our current best value, which allows us to prune. Often, the true value will be equal to the convex bound and we may return immediately without any local search. Assuming efficient pruning, the resulting algorithm is $(n \log n)$.

However we may note that computing the convex minimum can be done in amortized linear time. Starting from a budget of zero, it is sufficient to greedily increment the budget of the curve with the steepest slope — such that the total error is reduced the most. Because the curves are convex the rate of improvement never goes up, such that the greedy algorithm is optimal. Thus, if both curves are convex, the entire merge can be done in linear time.

It is important to note that individual table rows are not likely to closely match their convex bound. Rather, error curves empirically become more convex after merging with other curves. Thus our algorithm essentially builds a tree of rows — merging pairs of rows, and then pairs of level-1 error curves, etc. The low-level merge operations are very fast by virtue of having small constant factors. Without convex pruning the vast majority of the search effort would be spent merging the final few rows. For efficiency reasons our implementation performs most of these merges in parallel.

Budget Allocation

Now that we have explained how to merge error curves, we turn our attention to computing the optimal budget allocation. A naive algorithm would require a table of size $N \times (K+1)$, where N is the number of rows, and K is the maximum budget. Table row i would contain the error curve resulting from merging rows $1 \ldots i$. We could then reconstruct the optimal budget allocation by working backwards from row N, and reading back annotations from each table cell denoting how each entry's value was computed (i.e. the optimal budget split between row i and rows $1 \ldots i-1$).

Although each row may be sparse (once the error has reached zero it simply repeats), the amount of memory (and even disk space) required to store this table is prohibitive. Thus we modify the usual dynamic programming algorithm to act in a recursive manner, allowing us to "forget" large chunks of the table, and recompute them as needed.

Our algorithm begins by partitioning the rows into B blocks. We then compute the error curve for each block. The standard dynamic programming approach is then used to compute the optimal budget allocation over *blocks*, except now we only require a table of size at most $B \times K$. Given the optimal budget for each block, we call our modified recursive algorithm on each block individually to get the final allocation.

Curve Convexity

We have shown that merging two convex curves is an efficient operation. Because we are essentially merge sorting the incremental improvements, the resulting output curve is also convex. To justify our observation that curves become more convex after repeated merging, we now prove that the area between the true curve and the convex bound cannot increase through merging, and it may decrease.

First note that whenever there is a gap between the true curve and the lower bound, it is of the type shown in Figure 8.5, namely a linear lower bound with the true curve joining at the endpoints. There may be multiple such gaps, but each has a convex lower bound that is a straight line. Now

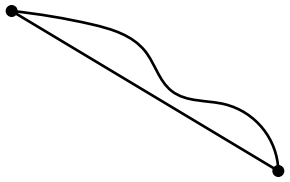

Figure 8.5: A gap between the true error curve and its convex lower bound.

consider cost curves a, b, and c, and let $\bar{a}$, $\bar{b}$, and $\bar{c}$ be the corresponding lower bounds. For each $\bar{c}_i$ there is a budget allocation $j + k = i$ such that $\bar{c}_i = \bar{a}_j + \bar{b}_k$ is minimized.

Let $\delta_i^a := a_i - \bar{a}_i$ be the gap between the true curve and the lower bound for budget i, with δ^b and δ^c defined similarly. Any such allocation for c or $\bar{c}$ produces a curve that is an upper bound on the optimal error. Consider the result of applying $\bar{c}$'s allocation to c, and call the resulting curve $\hat{c}$. Observe that $\hat{c}_i - \bar{c}_i = (a_j - \bar{a}_j) + (b_k - \bar{b}_k)$, where j and k are the allocation that minimizes $\bar{c}_i$. As i increases, whenever the (optimal) greedy allocation starts assigning budget to either $\bar{a}$ or $\bar{b}$ it is because that curve has the steepest slope. Without loss of generality, assume that the algorithm continues to assign budget to $\bar{a}$ or $\bar{b}$ until this is no longer the case. Thus if we are in the middle of a gap in $a_j - \bar{a}_j$ or $b_k - \bar{b}_k$, we will continue until we cross it, as $\bar{a}$ or $\bar{b}$ is locally a line, with unchanging slope (and is therefore still an optimal candidate for additional budget). In particular, we can never have a gap with respect to both a and b at the same time. Thus the cumulative error $\sum_i(\hat{c}_i - \bar{c}_i)$ is no greater than $\sum_j(\hat{a}_j - a_j) + \sum_k(\hat{b}_k - b_k)$. Finally, because $\hat{c}$ is an upper bound on c, we have that the cumulative gap size is non-decreasing under merge operations.

It is relatively straightforward to construct instances where the cumulative error does decrease. Consider $a = \langle 10, 8, 0\rangle$, and $b = \langle 10, 5, 1, 0\rangle$, such that $c = \langle 20, 15, 10, 5, 1, 0\rangle$. This example shows gaps of size 3 and 0, respectively, dropping to 0 for the combined curve. Here the intuition is that by the time it is optimal to allocate some budget to a, there is enough to essentially "jump over" the gap. This same effect can occur less completely when there is insufficient "slack" budget at the time the transition occurs, e.g., $a = \langle 10, 8, 4, 0\rangle$, $b = \langle 4, 1, 0\rangle$.

Thus, there is good reason to suspect that error curves will become more convex after repeated mergings, which agrees with our empirical observations. This suggests a further algorithmic improvement: repeatedly merging low-level rows until the aggregate is a (near) convex curve. Convex curves can be greedily (and optimally) merged without the complexity of dynamic programming. Note however that adding additional rows introduces additional cumulative error We have implemented this approach for This approach should be faster and more memory efficient than blocking,

but has not yet been implemented.

8.6 Results

To investigate the various combinations of cost function and optimization method we use the 4-card lookup tables, as these are sufficiently fast to optimize under different conditions. Note that any differences arising from the table entry weighting and budget allocation schemes must disappear as the budget becomes sufficiently large — no concessions need to be made if the entire table can fit in memory.

The effects of using dynamic programming versus greedy allocation are shown in Figure 8.6, for suit and grand tables. We compute the node reductions for lookup tables of various sizes, using all combinations of cost functions (with and without row weighting and valuation-frequency weighting). Although the reduction in wall-clock time is less than the node reductions shown, the general trend is the same. We can see that using dynamic programming for both naive and valuation-frequency weighting results in poorer performance, suggesting that those heuristics are poor choices. Row frequency weighting is the superior choice for both suit and grand games.

In the case of half rows and full rows, shown in Figure 8.7, dynamic programming is substantially better than the greedy allocation. We hypothesize that this is due to the following effect. By removing a single tentpole in the greedy case, the algorithm is strongly incentivized to keep removing entries from that row. However because the algorithm is myopic, it removes many points based solely on its initial decision of which row to damage. Obviously the dynamic programming approach does not suffer from this myopia.

To better see the differences between the various cost functions, we use the optimal choice of greedy or dynamic programming, and show the results in Figures 8.8 and 8.9, for full and half rows. In all cases simple row weighting is the superior choice, with its node/memory curves dominating the alternatives.

The difference between using full and half rows is shown in Figure 8.10. As expected, using half rows dominates the alternative. Storing the entire table using half rows requires about 40% of the space compared to naively storing all tentpoles. Note that these tables also include overhead for row indexing, so the amount of "useful" data is actually compressed by a larger amount.

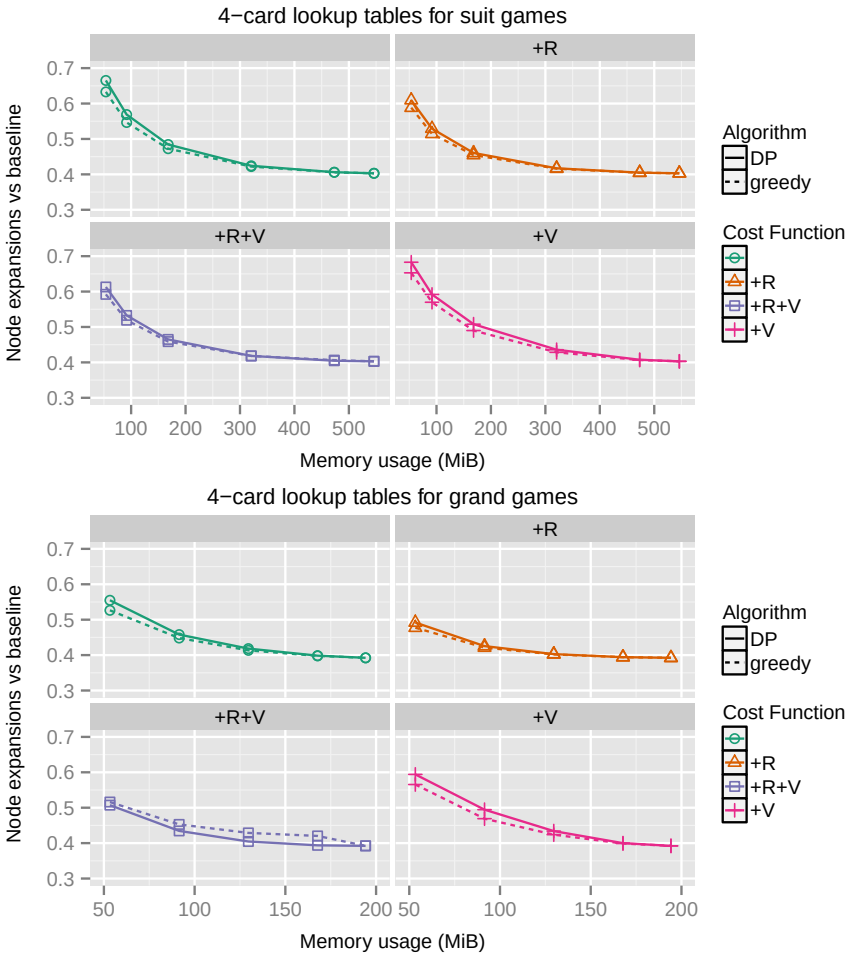

Figure 8.6: Row budget allocation comparing greedy selection and dynamic programming.

8.7 Null Tables

In a null game, the cards are not worth points — the value of a position is determined only by the relative ranks held by each player in each suit. For PIMC search, empirically it is the case that using only the win/loss value of a null position results in much weaker cardplay than using the soloist's time-to-live. Therefore we wish to construct null lookup tables capable of representing this, and which must therefore encode multiple values. In practice we only store eleven values: win, lose on trick 1, lose on trick 2, etc.

Using the same intuition that we applied to suit games — that the value of a position probably does not depend much on the location of the low-rank (high-rank) cards — we construct a decision

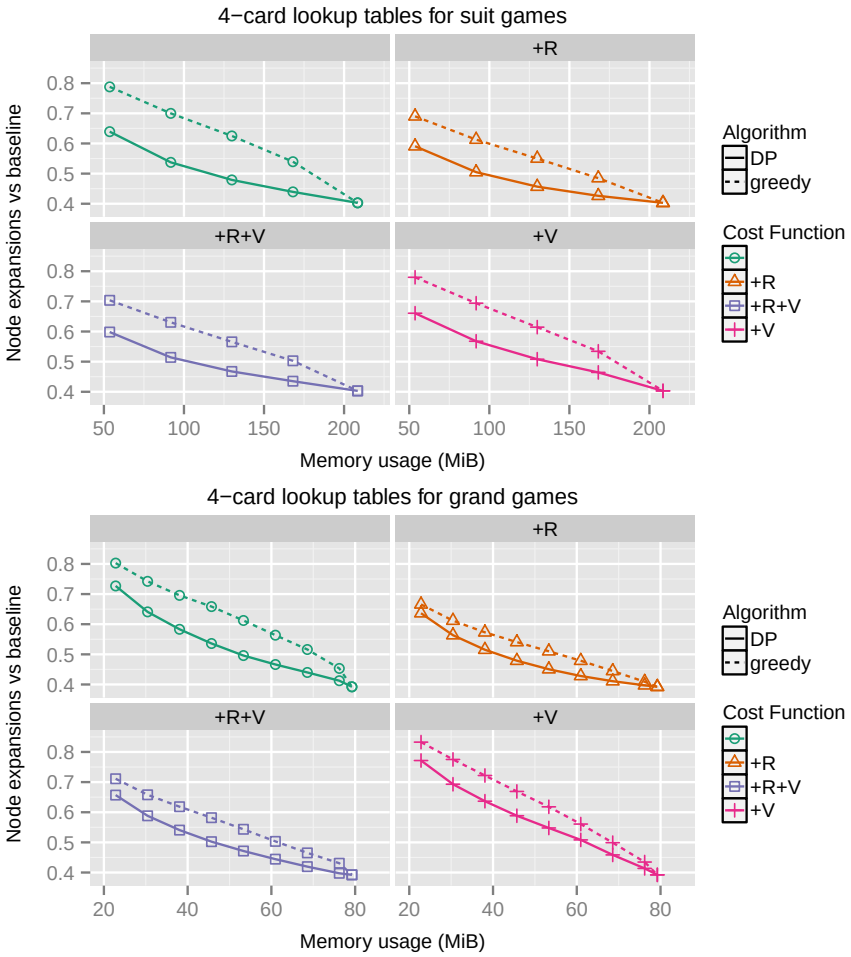

Figure 8.7: Row budget allocation with "half" rows, comparing greedy selection and dynamic programming.

diagram to compress the $\binom{3k}{k,k,k}$ ownership entries for each split. As with trump games, we only store start-of-trick positions, as these are easier to represent and including mid-trick lookups provides negligible performance gains (due to the constrained number of actions within a trick).

Exhaustive Construction

Recall that constructing a decision diagram involves storing the value for each (legal) combination of variables. In our case the variables represent the owner of each card. One natural algorithm for constructing these decision diagrams is to do so in a depth-first manner. Each node in the DD branches on the ownership of a particular card. Starting from a root node in which no cards have

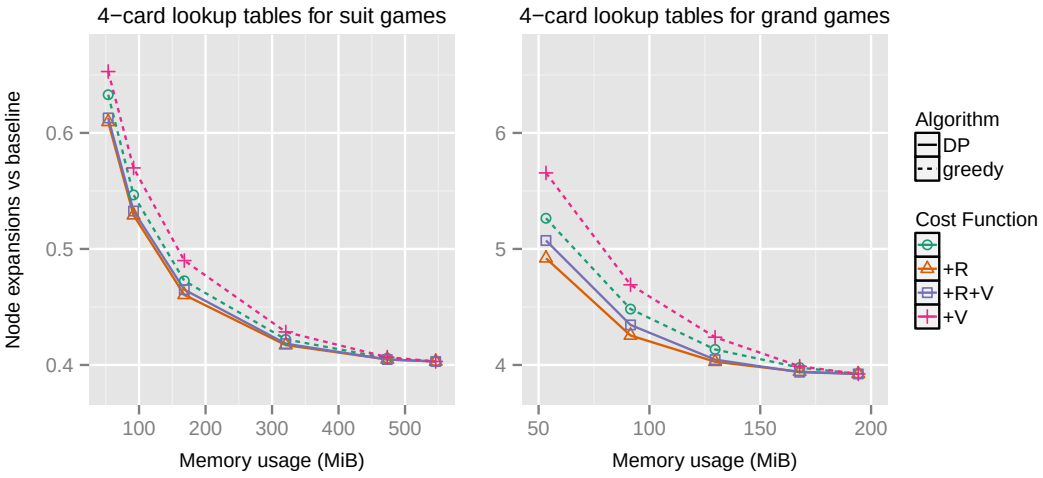

Figure 8.8: Effect of cost functions on lookup tables.

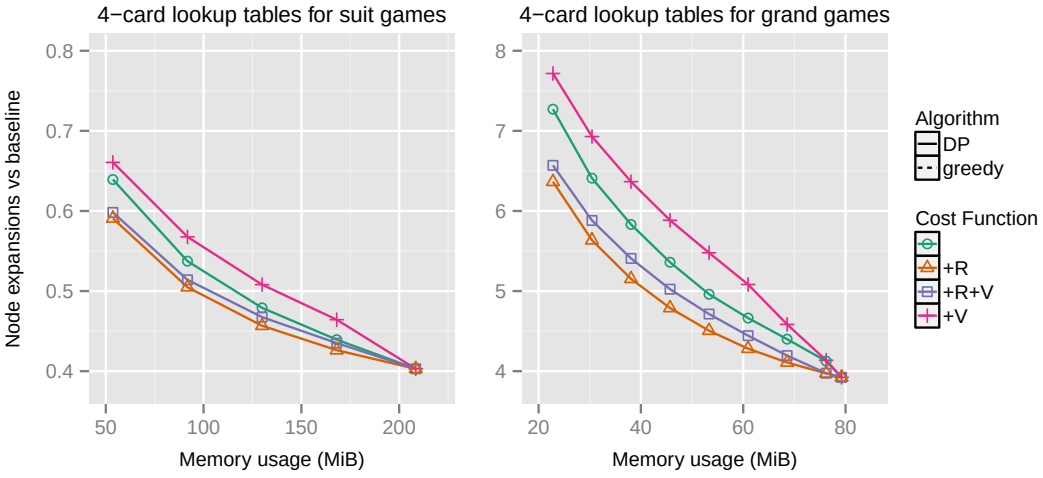

Figure 8.9: Effect of cost functions on lookup tables using “half” rows.

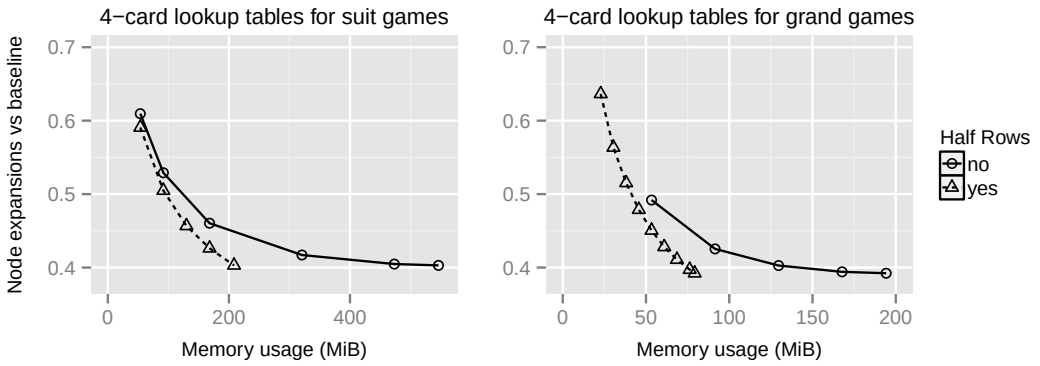

Figure 8.10: Speed/memory trade-off in lookup tables using half rows versus full rows.

been assigned, we walk the tree, assigning cards to players until all cards have been assigned. The resulting leaf nodes are thus fully-grounded positions, and may be solved in the usual manner. Thus we need not precompute (and store) the value of all states that we wish to encode.

The order in which we choose cards to assign is arbitrary, however we follow the DD convention that variables always occur in the same order throughout the DAG. Ordering by suit and then by rank within each suit (from low to high) seems to work well in practice.

As per standard DD minimization, if the children of an interior node all have the same value (or point to the same child node), then the parent node is redundant and may be collapsed. Certain paths in the DD correspond to illegal card assignments, such that one player has too many or too few cards. These branches are labeled *don't care*, as their value will never be queried. Such branches may be ignored when checking if a node's children are identical.

It is relatively straightforward to parallelize this tree traversal process near the root, say by creating a new execution thread for each ownership branch if the current search depth is less than some threshold d. To avoid contention or race issues for the DD global index of unique node identifiers, thread-local indices may be created and then merged with the global list when each thread terminates.

Such a process can construct lookup tables for null positions with up to 9 cards per player in a tolerable amount of time — several compute-node-days using commodity 8-core nodes. This is significant, because using 9-card tables practically eliminates search time as a bottleneck. All position evaluations return almost instantly. The majority of the computational effort is now spent constructing states to search, which involves inference and consistency checks. Results are shown in Table 8.3.

Table 8.3: Null search times using lookup tables of various sizes, compared to vanilla α-β search and lowest-winning-rank search. Averaged over 4000 random deals.

Algorithm	Nodes	CPU Cycles
Vanilla Null	506.0	280734
LWRS	445.3	247984
4-card	395.2	222786
5-card	302.0	170913
6-card	196.9	113001
7-card	103.9	63855
8-card	40.8	30158
9-card	13.2	12657

Perhaps even more significantly, the storage space for these tables is negligible. The resulting decision diagrams for hand sizes of 4 through 9 take up a total of 142 MB of disk space, without any optimization in terms of variable order. A complete optimization should be computationally tractable, and would reduce space/memory requirements and possibly traversal times during lookups. This would seem to be a case of chasing highly diminishing returns, however.

Going Further

Although essentially sufficient for null games, the previous algorithm has limited scalability, as all of the $\binom{3k}{k,k,k}$ leaves must be visited. If we instead solve the value of a leaf position using partition search then we have additional information — we know which card ranks affect the value of that position. Thus when we are propagating values up the search tree, we may refrain from searching sibling nodes if the current card rank was never used in the proof tree. That is, it allows us to backtrack further up the tree than would normally be possible, skipping over unimportant variables. However this simple explanation has some significant caveats.

To explain how our new algorithm works, we must first recall an implementation detail about our particular flavour of partition search. We are concerned only with those cards which win a trick by virtue of their rank. This rank information allows us to generalize to positions *where the number of cards a player holds in each suit does not change*, but where the ownership of certain "low" cards need not be specified.

This is somewhat troublesome, as the DD we are trying to create has no such constraint on the per-player suit lengths, only on the global suit lengths. This means that we have no way of directly applying partition search to a node in our DD traversal. Were we able to do so, then the "simple explanation" just given would suffice.

However the simple algorithm *does* hold if we restrict ourselves to a particular *fine split*, where the per-player suit lengths are specified. This allows us to construct a forest of decision diagrams, one for each fine split. Unlike before, we explicitly preserve *don't care* branches (which occur when the resulting suit lengths would violate the fine split). Because the search values returned are correct, and because we do not make any claims about the values which occur outside of our particular fine split (because we marked those with *don't care*), the fine split decision diagrams are also correct in the larger, global context.

Thus we can merge the fine split decision diagrams in a final post-processing step. Because the individual diagrams are efficient representations, this step is also efficient. Note that we could have resigned ourselves to having one decision diagram for each fine split, with lookups simply querying the relevant diagram, but this approach may miss opportunities for compression that would have occurred by merging nodes across diagrams. Also note that the construction of the individual diagrams is an inherently parallel procedure, suitable for cluster computation. We may also parallelize at the level of merging diagrams.

Detailed Analysis

Here we present a more detailed analysis of how the pitfall we just described can arise. If, during our tree traversal, a child is evaluated and its value (α-β or, more generally, an entire DD subgraph) does not depend on the current variable, it is tempting to consider the optimization of stopping early (ignoring any remaining sibling nodes) and returning that child value directly. This is incorrect. Consider the example in Figure 8.11. The number of cards each player has in each suit is fixed. At each node we assign ownership of a new card. If a player has not yet been assigned a particular card, the undetermined cards are indicated by a question mark. The card assignment ordering is static: ♡9TJ♠78. The current player-to-move is indicated by a star.

In the leftmost branch we assign the ♡9 to the soloist, and correctly conclude that the soloist cannot lose (cannot be forced to take a trick). Furthermore, LWRS is able to generalize that winning state to all similar states where the ♢J and ♡J are in the given locations. Specifically, the value does not depend on the location of the ♡9 that we initially branched on. However, by assigning the ♡9 to the soloist, we implicitly restricted the possible locations for the remaining cards. Thus the consistent worlds where the soloist holds ♡TJ were not considered along this path, and so pruning the rightmost branch would be an error.

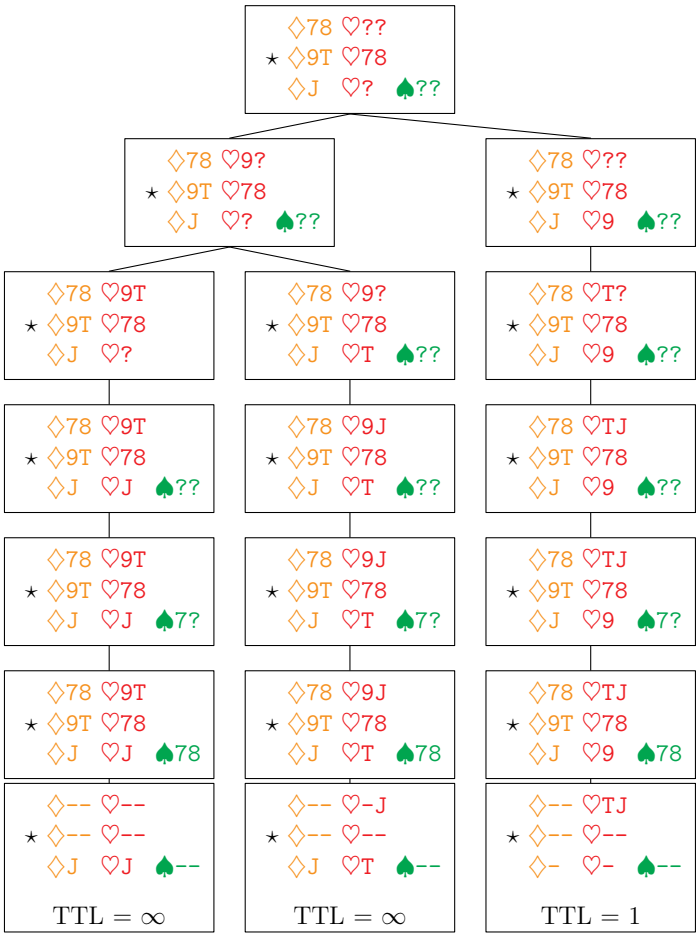

Figure 8.11: Example of how unsafe pruning can arise when constructing null lookup tables. The soloist's cards are listed in top row of each box, followed by the first and second defender. The player to move is denoted by a star. TTL is the time-to-live for the soloist.

8.8 Conclusions and Future Work

In this chapter we have presented a new state of the art method for constructing lookup tables in Skat, based on the observation that an entire space of card valuations can be described by a few well-chosen representatives. These results can be generalized to other domains that satisfy the given distance metrics. We described an efficient hybrid of dynamic programming and divide-and-conquer techniques to optimize lookup table construction, providing significantly better results than a previous greedy approach. Finally, we efficiently constructed lookup tables for null games, and encoded them using decision diagrams, effectively removing search as a bottleneck in null games.

It remains to apply the half-row technique to 5-card lookup tables for suit and grand games,

which promises to provide significant gains when using the same memory budget.

Chapter 9

Recursive Monte Carlo

> He knew this was among the alternatives
> today, a fact along lines of the future
> radiating from this position in time-space.
>
> *Dune*
> FRANK HERBERT

9.1 Introduction

In recent years AI research in the area of imperfect information games has flourished. For instance, in 2008 the Poker program "Polaris" defeated a group of six expert human players in two-player limit Texas hold'em in a duplicate match setting [20], and in 2009 "Kermit" reached expert playing strength in Skat [4]. The considerable progress in Poker is due to new techniques such as counter factual regret minimization for approximating Nash equilibrium strategies in smaller, abstract versions of the game [46], whereas in Skat fast perfect information Monte Carlo (PIMC) search combined with explicit state inference and heuristic state evaluation elevated programs to the next level.

Both approaches have distinct advantages and disadvantages. For instance, solving abstracted game versions off-line leads to fast on-line move computation because, essentially, available moves only have to be sampled from pre-computed probability distributions. However, finding good game abstractions that allow us to approximate move distributions in the original game well isn't trivial. A property of Poker is that the set of legal moves in each game state doesn't depend on the cards players are holding. Therefore, we only need to consider state abstractions (such as hand-strength bucketing) and we don't have to deal with move abstractions (except in those Poker variants where many legal bids are possible).

By contrast, in trick-based card games such as Bridge, Spades, and Skat, legal moves are defined by the cards players hold. Therefore, abstracting such games with the intent of using pre-computed move probabilities is non-trivial. PIMC search deals with this problem by sampling game states in accordance with observed moves and private state information, revealing game states to all players, and then evaluating moves by using search algorithms tailored for the perfect information setting.

The obvious drawback is that this method blatantly ignores players' ignorance and, consequently, for example has no concept of information gaining moves [11]. However, on the positive side, perfect information search algorithms such as alpha-beta search can be quite fast and, therefore, it may be possible to compensate for PIMC's shortcomings with regard to imperfect information game aspects by tactical performance.

In this chapter we focus on PIMC search and show how it can be improved by using move evaluators that are based on actual game play rather than analyzing perfect information scenarios. We begin by formalizing PIMC search and discussing related work. We then proceed by introducing imperfect information Monte Carlo search and presenting performance evaluations using a synthetic game tree model and the game of Skat. Finally, we conclude the chapter with a summary and suggestions for future research.

9.2 Related Work

An alternative to PIMC for dealing with imperfect information games is Monte Carlo Tree Search (MCTS), using UCT [24] variants that operate on information sets, rather than game states [38, 41, 8, 9]. We will discuss how UCT is generalized to searching over information sets in the experimental section, where we compare it with our new recursive imperfect information Monte Carlo search method. Although in Skat, UCT was unable to defeat PIMC search [38], in Hearts it proved to be stronger than the previous best known computer Hearts players [41].

Cazenave's nested Monte Carlo search for single-agent problems [5] is also highly relevant to this chapter. One can think of the recursive IIMC algorithm presented here as a generalization of nested Monte Carlo search applied to imperfect information games with more than one player.

9.3 PIMC Search

Algorithm 3 is a PIMC move-selection process which returns a move with highest average perfect information evaluation, given an information set I and a number of samples N. Function MOVES returns the set of moves available to the player to move in a given information set I. Note that all move sets are identical across all nodes in I. Function SAMPLE samples a node from I according to some state inference mechanism. This inference mechanism can take into account the game history, as known by the player to move. Function PERFINFOVALUE computes the perfect information value of making a move — again, with respect to the current player.

Algorithm 3: PIMC search pseudo-code.

```
PIMC (InfoSet I, int N)
for each m ∈ MOVES (I) do
    let val[m] = 0
end
for each i ∈ {1..N} do
    let x = SAMPLE(I);
    for each m ∈ MOVES(I) do
        let val[m] += PERFINFOVALUE(x, m);
    end
end
return argmax_m {val[m]}
```

The strength of PIMC hinges on the quality of the state inference module used by SAMPLE and the speed of PERFINFOVALUE. In the best case, when only one node is to be considered and all players share that knowledge, the game turns into a perfect information game, and thus, PIMC computes optimal moves. Furthermore, sampling more nodes empirically improves move value estimates.

The shown basic algorithm can be improved in many ways: In a tournament setting, checking a fixed sample size N can be replaced by an anytime algorithm that continues sampling until a given time threshold is reached. Also, if PERFINFOVALUE is fast, PIMC can easily be parallelized by assigning sampling and node solving to different cores or computers in a network. In the case where computing perfect information values takes considerable time, one can resort to parallelizing this task as well, for instance by using an efficient parallel alpha-beta search algorithm. Finally, it seems to be wasteful to continue evaluating moves that are likely inferior, if so implied by the sampling history. Using importance sampling, applying UCB [1] at the root node, or utilizing budget allocation algorithms such as OCBA [7] may improve basic PIMC performance in a given domain.

9.4 Imperfect Information Monte Carlo Search

In general one may consider replacing PIMC's perfect information move evaluation with an arbitrary evaluation function. In the imperfect information Monte Carlo (IIMC) algorithm, denoted Algorithm 4, we pass on an additional parameter: a player module used to finish playing out (imperfect information) games. This playout occurs immediately after sampling a node and making the particular

Algorithm 4: Imperfect Information Monte Carlo search pseudo-code.

```
IIMC (InfoSet I, int N, Player P)
for each m ∈ MOVES (I) do
    let val[m] = 0
end
for each i ∈ {1..N} do
    let x = SAMPLE(I);
    for each m ∈ MOVES(I) do
        let v = FINISHEDGAMEVALUE(x, m, P);
        let val[m] += v;
    end
end
return argmax_m{val[m]}

FINISHEDGAMEVALUE (Node x, Move m, Player P)
let y = MAKEMOVE(x, m);
while y is not a terminal node do
    let y = MAKEMOVE(y, COMPUTEMOVE(P, I(y)));
end
return value of y in view of the player to move in x
```

move we wish to evaluate. This computation is encapsulated in function FINISHEDGAMEVALUE. Note that the player module can range from random players, over rule-based systems, to quite sophisticated systems that execute PIMC or IIMC searches themselves. *Recursive IIMC* thus refers to an IIMC player using IIMC as a playout module, with the "recursion level" denoting the maximum recursive depth. For instance we may define $\mathrm{R}0$ as PIMC search, $\mathrm{R}1$ as IIMC calling PIMC, $\mathrm{R}2$ as IIMC calling $\mathrm{R}1$, etc. In even more general settings, one could specify player modules as a collection of individual players, which then would allow us to bias games to either exploit observed move biases or cooperate with partners in multi-player settings. Another important point is that IIMC suffers less from strategy fusion compared to PIMC provided we do not leak game state information to the player in the course of finishing games.

The concept of running MCTS on information sets — which collect game states the player to move can't distinguish — is quite similar to IIMC [38, 8]. In this setting the usual nodes of the MCTS tree instead correspond to information sets, and playouts involve sampling a world at the root and then playing the MCTS move for previously observed states. Outside of the tree a fast rule-based agent or random player can be used to complete the game. One important difference to IIMC lies in the fact that running MCTS on information sets leaks game state information implicitly, for if, at the root, we sample worlds consistent with the current player's view, and play out those

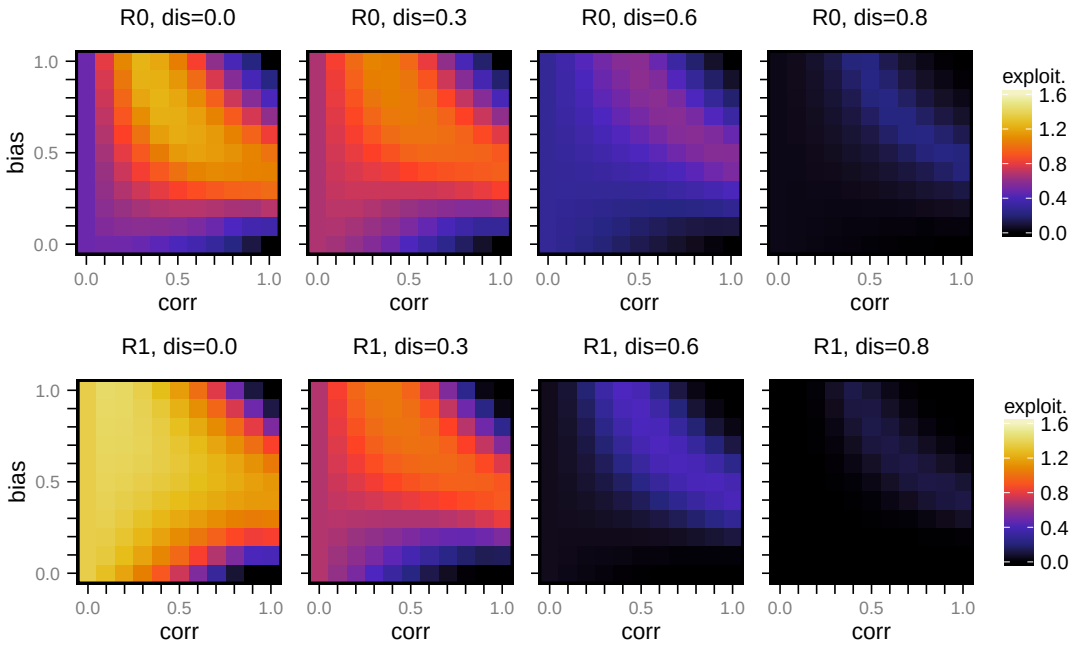

Figure 9.1: Exploitability of PIMC (R0) and RecPIMC (R1) on synthetic trees with varying levels of disambiguation.

worlds according to information set constraints, the other players will never see actions inconsistent with the true hand — their strategies will implicitly converge to "knowing" it. Moreover, the zero leakage "solution" of sampling inconsistent worlds is so detached from the real world as to be hopeless in terms of tactical play for all but small endgames. By contrast, IIMC, at least at the top-level, does not leak game state information because the agents' playout policies do not adapt across playouts. Another difference is that IIMC can use its playout module's inference mechanism in ComputeMove at any node during the search (because game tree nodes imply their game histories), whereas MCTS players we are aware of either sample nodes only at the root or simply assume uniform distributions for hidden state variables.

As an example of IIMC, consider an IIMC player that uses PIMC player P as its player module. We call the resulting player "Recursive P" or RP for short. RP has two parameters: number N of top-level nodes it samples, and number M of nodes it samples in the process of finishing games. For any top-level move m and sampled node x, RP completes the game by repeatedly calling PIMC $(I(y), M)$, which computes a move in information set $I(y)$ by sampling worlds and evaluating moves using perfect information search. Naturally, RP will be slow, as its runtime is roughly $C \cdot N \cdot M$ at the beginning of the game, for a suitable constant C that estimates the average time it takes

to solve positions along game trajectories. However, RP will suffer less from strategy fusion. So, it will be interesting to see for which choices of N and M RP can surpass P (which itself samples K nodes). Perhaps it is even possible to show that RP surpasses P's saturation point — the point at which increasing the number of sampled nodes K no longer increases P's performance significantly. In this case RP can benefit from hardware acceleration, but P cannot. We'll address these questions and more in the following sections.

9.5 IIMC Performance in Synthetic Trees

To investigate the asymptotic behaviour of the IIMC algorithm with PIMC as the base player, we used the generative tree model proposed by Long et al. [29]. This model defines a family of binary two-player zero-sum synthetic game trees parameterized by their *bias*, *correlation*, and *disambiguation* (BCD). Bias and correlation are properties defined in terms of the parents of the terminal nodes — they do *not* affect the correlation/bias of higher level tree branchings directly. Correlated leaves have the same values for their left and right actions. Non-correlated leaves have player-to-move left/right action values of $(1, -1)$ or $(-1, 1)$, chosen uniformly at random. Bias affects *only* correlated leaves, and assigns left/right values of $(1, 1)$ with probability *bias*, and $(-1, -1)$ otherwise. Disambiguation is a global property that affects how quickly an information set shrinks. A disambiguation value of $dis = 0$ results in maximum confusion, with information sets maintaining their full size throughout the game, while $dis = 1$ results in a perfect information game.

We divided the BCD parameter space into a number of triplets. For each triplet we generated several thousand synthetic trees and computed the policies produced by all recursion levels of IIMC (each level consumes one ply of the game tree), with $\mathrm{R}0$ = PIMC, $\mathrm{R}1$ = IIMC($\mathrm{R}0$), etc. We also refer to $\mathrm{R}1$ as RecPIMC. At each information set we exhaustively sample all consistent worlds, assigning equal weight to each (i.e., uniform inference). Where the playout/PIMC evaluations are the same, we assign equal weight to each move, otherwise the policy deterministically selects the move with the highest evaluation. Because each recursive policy uses the evaluations of the previous iteration, we may efficiently compute each policy in time proportional to the game tree size. For consistency with Long et al. [29], we chose a tree height of 8, with 8×8 nodes at the root (a *chance* node where each player can distinguish only 8 information sets). Experiments with tree heights of 7 and 9 showed no substantial differences.

To measure the exploitability of each of the $\mathrm{R}0$, $\mathrm{R}1$, etc., policies (collectively R^*), we compute a corresponding best response player — this is simply the expectimax strategy, given a fixed

opponent strategy. To eliminate any first-player bias, we evaluate with players in both positions and report the sum of the scores (thus the maximum possible exploitability is 4). Because the trees are sufficiently small and we have each player's policy, we compute the game theoretic score for each tree directly, rather than sampling playouts. In Figure 9.1 we show the exploitability for $\mathrm{R}0$ and $\mathrm{R}1$ at various disambiguation levels. For low-disambiguation (poker-like) games, $\mathrm{R}1$ performs notably worse than $\mathrm{R}0$. The precise reason for this is unclear, but we will discuss some possibilities.

First, consider the family of BCD trees with $dis = 0$ and $corr = 0$ (*bias* is irrelevant when $corr = 0$). The leaf player always has one move which wins and one which loses. Thus, to an $\mathrm{R}0$ player, the leaf player always has a winning strategy (it "knows" the true world and can always guess correctly). However, since all paths appear to lead to wins (losses), $\mathrm{R}0$ views all moves equally, *except at the leaves*. At the leaves, the $\mathrm{R}0$ player chooses the move which wins most often on average (over the nodes in the current information set).

Now consider the $\mathrm{R}1$ player on the same BCD family. Because $\mathrm{R}1$ is performing playouts using $\mathrm{R}0$ as a playout module, it sees (at all nodes) that not all moves lead to wins. Thus the $\mathrm{R}1$ policy becomes biased towards certain regions of the tree. It may be that it is simply the presence of deterministic moves which is allowing it to be exploited. Forcing R^* players to select an arbitrary (but fixed) move when both actions appear equal may shed light on the issue, since the number of deterministic actions would be more similar.

A second reason may be that the lack of an inference model is affecting our results. The nature of such an effect is non-obvious, but the possibility of its existence cannot be ignored. That said, any negative effects of using $\mathrm{R}1$ seem to disappear once disambiguation exceeds 0.3. Based on the measurements by Long et al. [29] of real game states, disambiguation in trick-taking card games such as Hearts and Skat can be expected to lie somewhere in the region of 0.6, with some noise due to structural discrepancies between the generative model and the real games. Thus we expect to see significant reductions in exploitability for this class of games by adding a single level of recursion to existing PIMC agents.

To gain a fuller understanding of what each level of recursion provides, Figure 9.2 shows the pairwise R^* performance, and the exploitability of each player. The largest drop in exploitability comes from the first level of recursion, with minor improvements occurring until $\mathrm{R}5$. This is promising, since going beyond $\mathrm{R}1$ or $\mathrm{R}2$ with current commodity hardware seems unlikely. There are clear signs that $\mathrm{R}1$ is performing better against $\mathrm{R}0$ than would be seen from simply being a "stronger" player. This is perhaps due to $\mathrm{R}1$ having a good implicit opponent model of $\mathrm{R}0$, but as $\mathrm{R}1$ is less exploitable against a best-response strategy, this seems a very fair trade-off.

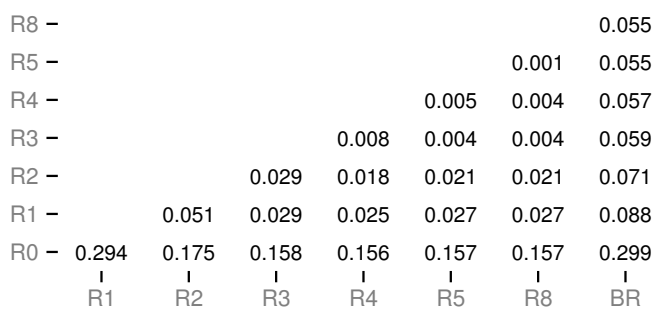

Figure 9.2: Exploitability details for the BCD-tree parameter space resembling Hearts. Values are the expected gain of the column player, over the row player, when playing pair games generated using the given BCD-tree parameters. BR indicates a best-response strategy to the particular row player.

9.6 IIMC Performance in Skat

In this section we conduct several experiments with the purpose of measuring the playing strength of PIMC-/IIMC-based players in a real game, under various world-sampling parameter settings. We discuss the Skat AI systems we used in our experiments, describe the experimental setup, and finally present and interpret the tournament results we obtained.

Skat AI Systems

By 2008, Kermit had reached expert playing strength [4] by combining fast PIMC search with card inference, and heuristic state evaluation whose parameters were fitted using millions of Skat games played by human players. Kermit's card inference is based on tracking void suits and learned histograms for important hand features such as suit length and high-card distributions given players' maximum bids. This inference module is used to sample information set nodes which are then passed on to a fast perfect information solver. Kermit's suit game solver is capable of computing exact card point scores for all legal moves at a rate of approximately 68 worlds per second (at the start of the cardplay phase — 30 cards to play) on a single Intel i7 core running at 4 GHz. With forward pruning (at a ProbCut threshold of 0.5) this number increases to 237 all-move approximate evaluations per second.

The second Skat AI system we use in our experiments is XSkat. XSkat is a free software Skat program written by Gunter Gerhardt (www.xskat.de). It is essentially a rule-based system that does not perform search at all and plays very quickly, which makes it well suited for our recursive PIMC

search framework. The third and last Skat program we consider here is Bernie, which runs the UCT MCTS algorithm on information sets, choosing consistent worlds uniformly at random and using XSkat as playout module [38].

Experimental Setup

We measured the cardplay strength of the different players using a common set of 18,962 games produced by 3 Kermits bidding, discarding, and declaring. There were initially 20,000 deals, but some led to all players passing during bidding. These passed games are not included in the scores presented. All games were scored using the widely used Fabian-Seeger tournament scoring method.

When sampling consistent worlds from an information set, Kermit's inference module takes the bidding history and game declaration (but not cardplay) into account to bias worlds. Our recursive players use this inference module when sampling worlds at the top level.

Cardplay matches were run on heterogeneous hardware, with differences in speed not affecting our results. When we list timing information it is with respect to an Intel i7 core running at 4 GHz. In the interest of speed we reuse a player's move selections across matches when encountering a previously seen information set. Correctness is not affected, as inference does not consider the names of the other players. Moreover, we expect that, by correlating games in this manner, we increase the statistical significance when varying one player.

Information Set UCT vs. RecPIMC

In this subsection we compare the playing strengths of information set UCT player Bernie and IIMC player RXSkat which both use the same fast player module — XSkat.

Our recursive XSkat agent (RXSkat) samples consistent worlds (either with or without Kermit world inference), and uses XSkat completions to evaluate the effect of playing each possible move. We gave Bernie a budget of 1600 playouts and RXSkat a budget of 160 worlds (with up to 10 legal moves in each world). Thus, Bernie always uses at least as many XSkat playouts, although starting from varying depths. RXSkat is on the order of 35 times faster than Bernie, although presumably the UCT implementation could be improved. XSkat is fast enough that playing entire hands with three RXSkat players takes on the order of 1 second, using one CPU core.

Bernie's lack of informed world inference puts it at a disadvantage compared to RXSkat which uses Kermit's inference module — especially considering that the model of the bidding matches the actual process used for generating hands. Thus, we include results when RXSkat's inference module

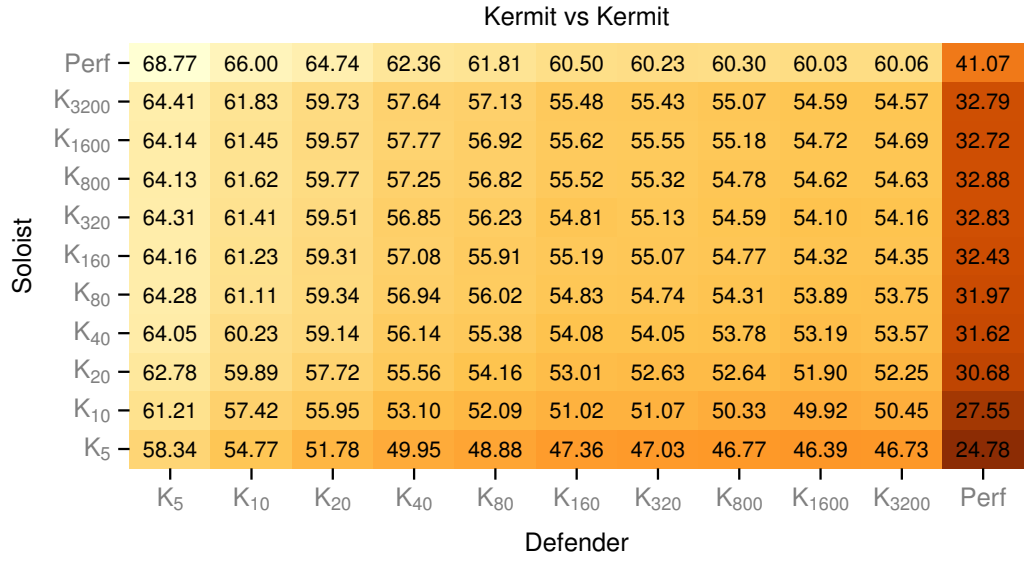

Figure 9.3: Kermit vs. Kermit games showing the diminishing effects of sampling more worlds as declarer and as defender. Perf denotes the "perfect information player" who knows the state of the game. Table entries are soloist scores. 95% confidence intervals for each entry are in the range of ± 1.2 to ± 1.6.

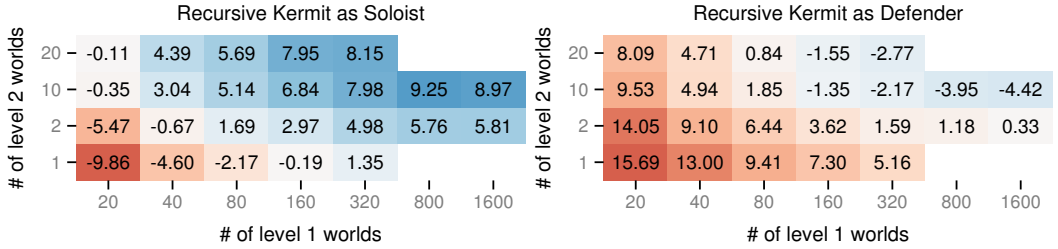

Figure 9.4: Recursive Kermit with varying numbers of first- and second-level worlds vs. Kermit using 160 worlds. Soloist scores are shown, relative to a baseline Kermit vs. Kermit score of 55.19. One standard deviation is about 0.5 points for all entries. The same 18,962 games were used for each data point. The (1600,10) entries indicate that Recursive Kermit is considerably stronger than PIMC-based Kermit at its saturation point. Lower scores are better when acting as a defender.

is disabled and replaced with uniform random as well.

To examine the difference between Bernie and RXSkat we use pair games, where copies of both agents take the roles of soloist and defender, playing each deal from both perspectives. Given Skat's three player nature, it is non-trivial to extrapolate tournament scores from pair matches.

Assuming equivalent bidding, we expect a single player to take the role of soloist one third of the time, or 12 hands in a 36-hand tournament list. Thus, an increase of 10 tournament points as soloist

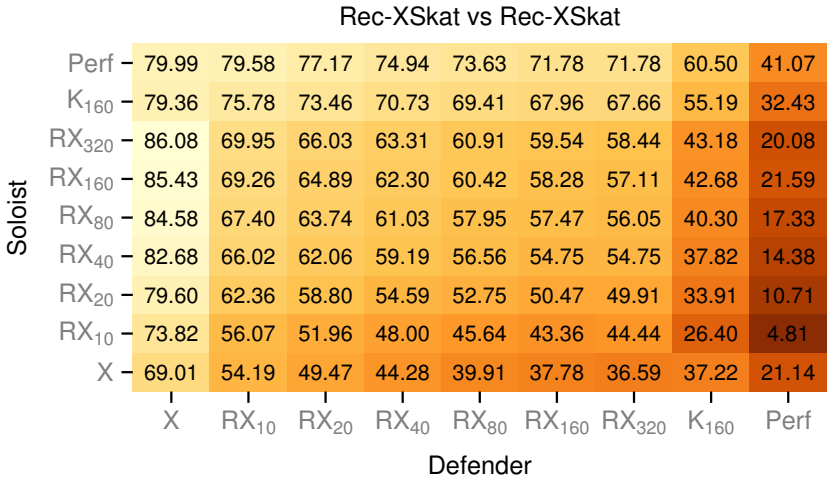

Figure 9.5: Declarer scores of pairwise matches between Recursive XSkat using Kermit inference with varying numbers of top-level worlds, XSkat, and Kermit with 160 worlds. 95% confidence intervals for each entry are in the range of ± 0.9 to ± 1.6.

Table 9.1: Recursive XSkat (RXSkat, with and without world inference) versus UCT on information sets (Bernie). RXSkat samples 160 worlds, while Bernie uses 1,600 playouts. The confidence interval is two standard deviations. The results indicate that RXSkat is considerably stronger than Bernie.

Soloist	Defenders	Soloist Score
Bernie	$\text{RXSkat}_{\text{infer}=0}$	61.34 ± 1.30
$\text{RXSkat}_{\text{infer}=0}$	Bernie	74.14 ± 1.12
Bernie	$\text{RXSkat}_{\text{infer}=1}$	53.99 ± 1.97
$\text{RXSkat}_{\text{infer}=1}$	Bernie	74.37 ± 1.58

corresponds to 120 list points. However, stronger soloist play also has the effect of decreasing the number of hands successfully defended, effectively reducing the expected score of the other players.

We show in Table 9.1 the expected declarer points for each match-up. Not shown is the change in expected defender tournament points, which goes from 7.05 to 4.81 in favour of $\text{RXSkat}_{\text{infer}=0}$ versus Bernie. Over 24 hands (as defender), this corresponds to RXSkat gaining 53.76 list points from defender wins alone. Combined with an expected gain of $12.8 \times 12 = 153.6$ declarer points, RXSkat can acquire over 200 points more than Bernie in a list, due to improved cardplay. A 200 point difference in a 36-deal list is roughly what separates World Championship calibre players from average club players. Recursive XSkat is therefore considerably stronger than Bernie, which is based on information set UCT.

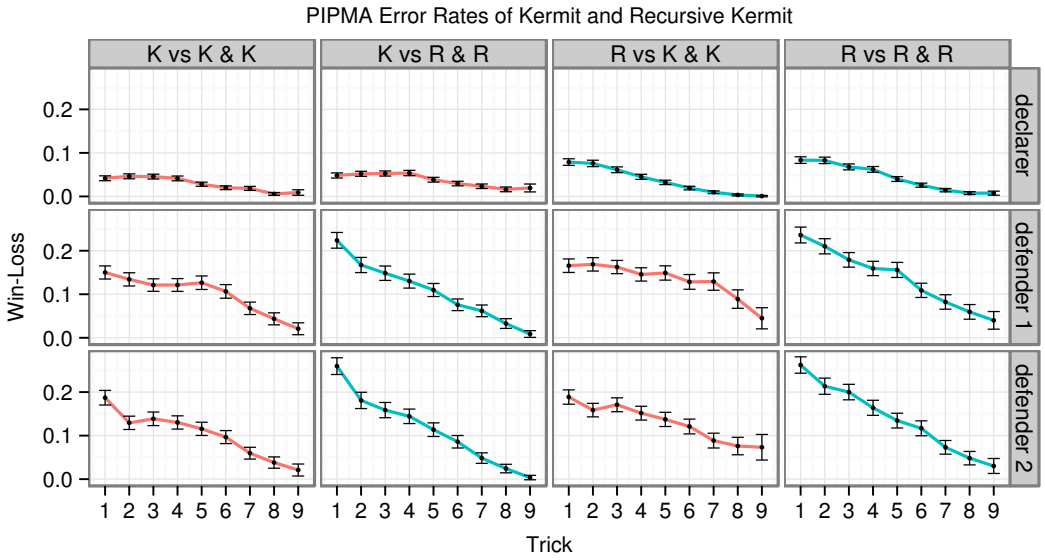

Figure 9.6: PIPMA error rates of Kermit and Recursive Kermit in all roles, measuring the frequency of perfect information game-losing mistakes. The confidence intervals indicate two standard deviations. The declarer sits in front of defender 1.

RecPIMC vs. PIMC

As seen in Figure 9.3, increasing the number of worlds sampled by PIMC player Kermit beyond 160 does not significantly improve playing strength with respect to either other Kermits nor a perfect information player (which cheats by observing the true world).

To overcome this plateau, we can observe in Figure 9.4 the effects of applying RecPIMC (a.k.a. R1) with Kermit playout modules. Playing against Kermit, both as soloist and defender, recursive Kermit achieves significantly stronger performance. For instance, $RK_{160,10}$ with 160 level-1 worlds, and 10 level-2 worlds gets about 7 points more per game than Kermit, when playing as soloist, which roughly corresponds to an 80 point gain in a list — a substantial performance increase, given that Kermit at the PIMC saturation point is currently the strongest Skat program playing at human expert level [4].

We estimate that this playing strength increase makes $RK_{160,10}$ competitive with the strongest human players, but further experiments have to be conducted with other opponents to see exactly how much of the performance gain can be attributed to having a good opponent model. Before arranging human-machine matches, which need to consist of hundreds of games to generate statistically significant results, we have to create a distributed version of $RK_{160,10}$ to offset its roughly 8-fold slowdown compared to Kermit — a PIMC-based player considering 160 worlds. Running

$RK_{160,10}$ on 32 Intel x86 cores will suffice to let it play under tournament conditions ($\approx$ 3 seconds per move on average). As indicated by the (1600,10) entries in Figure 9.4, increasing the hardware speed by another factor of 10 leads to an additional playing strength gain of roughly 2 points per game versus Kermit. This is exciting, because unlike PIMC-based Kermit, which doesn't benefit from further hardware acceleration, we anticipate being able to run $RK_{1600,10}$ under tournament conditions in the near future. As $RK_{160,10}$ is almost suitable for real-time usage, we will use it when discussing recursive Kermit unless otherwise stated.

Implicit Opponent Modelling Effects

We can see in Figure 9.5 evidence that a non-trivial portion of RXSkat's strength derives from accurately modelling its opponents. Observe that RX_{320} as soloist performs substantially better than Kermit, but only when XSkat is the defender. In the presence of stronger defense, the situation reverses, with Kermit outperforming RXSkat. However, there is a general trend that RXSkat is stronger than XSkat, for sufficiently many top-level worlds. As evidence that the gains of RK are not all due to modelling we present data (not plotted) for $RK_{160,10}$ playing against two XSkat defenders: whereas Kermit and Perfect had declarer scores of 80.7 and 80.9 respectively, RK had a score of 83.8. Since Kermit is a poor model for XSkat, we believe that the gains against XSkat were due to increased imperfect information performance, rather than exploiting a good opponent model. This effect is also visible in Figure 9.5: against two K160 defenders, RXSkat with a small number of worlds is weaker than XSkat, but then catches up and eventually outperforms XSkat when using 80 worlds or more. When defending against K160, all considered RXSkat versions are stronger than XSkat.

Perfect Information Post Mortem Analysis

Figure 9.6 shows the perfect information post mortem analysis (PIPMA, [28]) error rates for Kermit vs. Recursive Kermit tournaments. Errors are defined as moves that go from a winning open-handed position to a losing one. Positions where no errors are possible do not affect the error rates. A general trend of making fewer perfect information mistakes towards the end of the game is clearly visible. What is interesting is that defenders' error rates are higher when recursive Kermit is the soloist. This indicates that RecPIMC based players may be able to increase confusion on the defender side. Another interesting observation is that although recursive Kermit seems to have higher error rates during the first half of the game, it is still winning against Kermit overall. Whether this is an artifact

of PIPMA or indicating that recursive Kermit is trading "optimality" for creating confusion will be the subject of future work.

9.7 Conclusions and Future Work

In this chapter we have introduced imperfect information Monte Carlo search which can combine the tactical strength of perfect information move evaluation with superior modelling of imperfect information aspects. As a result IIMC search players are less prone to strategy fusion effects, compared to PIMC search, and they are also capable of increasing opponents' confusion by accurately anticipating their reaction to played moves. IIMC also appears to significantly outperform information set UCT on the domains we examined.

The results we obtained for RecPIMC, an IIMC variant using perfect information playouts, on synthetic games suggest that its performance is correlated with the game's disambiguation factor: the more information is revealed in course of the game, the better RecPIMC performs compared to PIMC. This corresponds to a broad class of games, notably trick-based card games, where we expect RecPIMC to perform well.

The results of the extensive tests we conducted suggest that in our domain — computer Skat — the playing strength gain achieved by using RecPIMC search can be substantial and that it in fact surpasses the strength of regular PIMC search at its saturation point. This is good news for future developments because with faster multi-core processors on the horizon, RecPIMC in Skat will likely become fast enough to defeat PIMC based players under tournament conditions within a few years time.

RecPIMC search has the potential to push the state of the art in trick-based card game AI even further. For example, one can imagine that playing better in Skat's cardplay phase can be compounded with more aggressive bidding to obtain even higher scores, because stronger card players can get away with weaker hands more often. Also, RecPIMC search opens the door to actively deceiving opponents and cooperating with partners beyond what simple state inference can accomplish, simply by gauging the merit of every move in an imperfect information setting. Lastly, according to Figure 9.6, non-recursive Kermit has a lower PIPMA error rate than recursive Kermit in the early game. This suggests the creation of a hybrid player that uses recursive evaluations only in the late game. However, the PIPMA rate is only a proxy for measuring how exploitable an agent is by a true best-response player — one that is constrained by imperfect information. It is possible that by playing "losing" moves, recursive Kermit is gaining important information about the hidden

distribution, and/or "confusing" the defenders by directing play to positions where they are likely to make mistakes.

We have shown that recursive Kermit is significantly stronger than Kermit against a range of opponents, but it remains to be seen how well this technique generalizes to other domains where PIMC search performs well, such as Hearts or Bridge. The promising results we obtained for synthetic games with high disambiguation factors make us confident that this will indeed be the case.

Chapter 10

Conclusion

> Arrakis teaches the attitude of the knife –
> chopping off what's incomplete and saying:
> "Now it's complete because it's ended here."
>
> *Dune*
> FRANK HERBERT

10.1 Future Work

Although we have dealt with perfect information search problems that are interesting in their own right, a large portion of this thesis has been essentially laying the groundwork for future research into imperfect information search techniques. We have barely scratched the surface of the rollout-based methods presented in Chapter 9. It remains to be seen whether the significant performance gains seen in Skat can be replicated in similar domains such as Hearts and Bridge. Results on synthetic game trees suggest they can be, but synthetic domains are inherently limited.

With perfect information evaluations becoming less of a bottleneck, opportunities arise for search-based inference techniques — asking whether a particular world is likely based on how a PIMC agent would have responded, and if that response matches the observed history.

In Skat, fast cardplay allows us to handle the problem of bidding in a more principled manner. Using self-play we can bootstrap our estimators of whether a particular initial hand can successfully bid and win a game. Because we suspect our PIMC player is much stronger than an average human, this should produce much more accurate estimates than using the current human-played games as training data. We can even consider CFR-style sampling approaches to construct mixed strategies for bidding. A related goal is to automatically adjust our bidding and cardplay to exploit weaker opponents.

Returning to the perfect information setting, we have described how LWRS (partition search) can be combined with payoff-similarity. This allows us to more broadly generalize the results of a given α-β search. It remains only to overcome certain software engineering hurdles to actually implement

such a system. LWRS produced substantial efficiency gains in Bridge; if those same gains could be had in Skat it would allow recursive PIMC to be run on current commodity hardware.

Going further, the same LWRS symmetries could potentially be used to better compress midgame lookup tables. Being able to represent portions of the 6-card tables (or even larger) could be a significant gain. Even without LWRS, it should be straightforward to make use of half rows and dynamic programming to better compress the 5-card lookup tables. This would allow computers with limited memory (e.g. cluster nodes) to load partial 5-card tables and see worthwhile performance gains.

10.2 Conclusion

Throughout this work we have used the domain of Skat to motivate research into perfect and imperfect information search techniques. Our contributions include a formal analysis of several symmetry-based search extensions and their integration with a high-performance perfect information Skat solver. We have presented methods for bounding the α-β differences between related game states, and extended those results to constructing efficient midgame lookup tables that significantly increased search performance. Finally, we investigated properties that correlated with PIMC's success in imperfect information domains, and showed that recursive rollout-based techniques could achieve superior results, in both head-to-head performance and exploitability.

Bibliography

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[2] P. M. Bethe. DTAC: A method for planning to claim in bridge. Master's thesis, New York University, United States, May 2010.

[3] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[4] M. Buro, J.R. Long, T. Furtak, and N. Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI2009)*, 2009.

[5] Tristan Cazenave. Nested Monte-Carlo search. In *Proceedings of the 21st international joint conference on Artifical intelligence*, pages 456–461. Morgan Kaufmann Publishers Inc., 2009.

[6] M-S. Chang. Building a fast double-dummy bridge solver. Technical report, New York University, New York, NY, USA, 1996.

[7] C.H. Chen. *Stochastic Simulation Optimization: An Optimal Computing Budget Allocation*. Stochastic Simulation Optimization. World Scientific Publishing Company, Incorporated, 2010.

[8] P.I. Cowling, E. Powley, and D. Whitehouse. Information set Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI for Games*, 4:120–143, 2012.

[9] P.I. Cowling, C.D. Ward, and E.J. Powley. Ensemble determinization in Monte Carlo tree search for the imperfect information card game magic: The gathering. *IEEE Transactions on Computational Intelligence and AI for Games*, 4:241–257, 2012.

[10] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[11] Ian Frank and David Basin. Search in games with incomplete information: a case study using Bridge card play. *Artificial Intelligence*, 100(1-2):87 – 123, 1998.

[12] Ian Frank, David A. Basin, and Alan Bundy. Combining knowledge and search to solve single-suit bridge. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 195–200. AAAI Press, 2000.

[13] S.H. Fuller, J.G. Gaschnik, and J.J. Gillogly. An analysis of the alpha-beta pruning algorithm. Technical report, Carnegie Mellon University, 1973.

[14] Timothy Furtak and Michael Buro. Minimum proof graphs and fastest-cut-first search heuristics. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 492–498, 2009.

[15] Timothy Furtak and Michael Buro. Using payoff-similarity to speed up search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[16] M. Ginsberg. GIB: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, pages 303–358, 2001.

[17] Matthew L. Ginsberg and Alan Jaffray. Alpha-beta pruning under partial orders. In *Games of No Chance II*, 2001.

[18] B. Haglund. Search algorithms for a bridge double dummy solver. Technical report, 2010.

[19] B. Haglund. http://privat.bahnhof.se/wb758135, 2012.

[20] Michael Johanson. Robust strategies and counter-strategies: building a champion level computer Poker player. Master's thesis, University of Alberta, Canada, 2007.

[21] H. Kaindl and A. Scheucher. Reasons for the effects of bounded look-ahead search. *IEEE Trans. Systems Man Cybernet*, 22(5):992–1007, 1992.

[22] T. Keller and S. Kupferschmid. Automatic bidding for the game of skat. In *31st Annual German Conference on AI (KI 2008)*. Springer-Verlag, 2008.

[23] D. Knuth and R. Moore. An analysis of alphabeta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[24] L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*, pages 282–293, 2006.

[25] L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*, pages 282–293, 2006.

[26] S. Kupferschmid and M. Helmert. A skat player based on Monte-Carlo simulation. In *Proceedings of the 5th International Conference on Computers and Games*, pages 135–147. Springer-Verlag, 2007.

[27] D. Levy. *The million pound bridge program*. Ellis Horwood, Asilomar, CA, 1989.

[28] Jeffrey R. Long and Michael Buro. Real-time opponent modelling in trick-taking card games. In *Proceedings of IJCAI*, pages 617–622, 2011.

[29] Jeffrey Richard Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search. In *AAAI*, 2010.

[30] D.S. Nau. An investigation of the causes of pathology in games. *Artificial Intelligence*, 19(3):257–278, 1982.

[31] M.M. Newborn. The efficiency of alpha-beta search on trees with branch-dependent terminal node scores. *Artificial Intelligence*, pages 137–153, 1977.

[32] J. Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20(4):427–453, 1983.

[33] A. Plaat. *Research, Re: search & Re-search*. PhD thesis, Rotterdam, Netherlands, 1996.

[34] A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. Exploiting graph properties of game trees. In *AAAI National Conference 1:234–239*, pages 234–239, 1996.

[35] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):255–293, 1996.

[36] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 2002.

[37] Sartaj Sahni. Computationally related problems. *SIAM J. Comput.*, 3(4):262–279, 1974.

[38] J. Schäfer. The UCT algorithm applied to games with imperfect information. Master's thesis, University of Magdeburg, Germany, October 2007.

[39] A. Scheucher and H. Kaindl. Benefits of using multivalued functions for minimaxing. *Artificial Intelligence*, pages 187–208, 1998.

[40] Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[41] N. Sturtevant. An analysis of UCT in multi-player games. In *Computers and Games*, 2008.

[42] N. Sturtevant and A.M. White. Feature construction for reinforcement learning in hearts. In *Computers and Games*, pages 122–134, 2006.

[43] Nathan R. Sturtevant and Richard E. Korf. On pruning techniques for multi-player games. In *AAAI/IAAI*, pages 201–207, 2000.

[44] B. von Stengel and D. Koller. Team-maxmin equilibria. *Games and Economic Behavior*, 21:309–321, December 1997.

[45] http://www.xskat.de, 2009.

[46] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in Neural Information Processing Systems 20 (NIPS)*, pages 905–912, 2008. A longer version is available as a University of Alberta Technical Report, TR07-14.

Appendix A

The Rules of Skat

In discussing the rules of Skat we will need to make reference to the *value* of a game/contract. The process for determining this value is somewhat involved and will be postponed until later.

A.1 Cardplay

Each hand of Skat begins with all three players being dealt 10 private cards. The two remaining cards (the *skat*) are dealt face-down. At no time are players allowed to show or talk about their hands. Unlike in bridge, there are no fixed alliances of players. Each hand consists of a *bidding* and a *cardplay* phase, with the results of the bidding determining the teams for that hand. The player who wins the bid is termed the *soloist* and competes against the other two players (the *defenders*). After bidding, the soloist must announce a game type that is worth at least as much as her bid. A bid in Skat is simply a number — however, this number should correspond to the value of a legal game, so as to avoid arbitrarily precise signaling via fractional bids.

The order of bidding is determined by seating order. The player to the dealer's left is said to be in "forehand", and will eventually begin the cardplay phase (i.e. lead the first card). The next player (clockwise) is in "middlehand", and the third player (the dealer) is in "rearhand".

Bidding begins with middlehand either passing, or announcing a bid to forehand. Forehand then either accepts the bid (says "yes") or passes. This repeats, with middlehand announcing a higher bid until either forehand passes or middlehand chooses not to bid. Once a player has passed, rearhand then bids to the remaining player in a similar manner. If both middlehand and rearhand pass (so that forehand has not yet said "yes") forehand may either pass — in which case the hand is not played — or announce a bid.

The player that won the bid may now optionally pick up the two cards in the skat, rearrange her hand, and place any two cards back down (face down). Not picking up the skat results in a *hand* game, which is worth more points. The soloist then announces a game type that is worth at least as much as her bid. The three game types are: suit, grand, and null. In suit and grand games, the soloist

is attempting to collect 61 or more *card points*. These card points are unrelated to the value of a game. Each card rank is worth a fixed number of points: Aces – 11, Tens – 10, Kings – 4, Queens – 3, and Jacks – 2. The remaining cards (9s, 8s, and 7s) are not worth any points. The total number of card points is thus 120. The two cards in the skat (after picking up and discarding) are considered to have been won by the soloist.

In suit games the named suit (♢♡♠♣) is trump, along with all the jacks. The jacks are the highest trump, and are ranked in a fixed order — clubs, spades, hearts, diamonds, with clubs being the highest. For the purposes of following suit, jacks are considered to be part of the trump suit. Tens are ranked higher than they are in the "standard" ordering — between Aces and Kings. The trump suit card rankings are shown in Figure A.1. Non-trump card rankings are shown in Figure A.2. Grand games are similar to suit games, but only the Jacks are trump.

Like most trick-taking games, players must follow suit (play the same suit as the led card) if they are able to do so. If a player cannot follow suit then she may play any card. The winner of a trick is the player who played the highest trump, or, if no trump were played, the highest-ranked card in the led suit. The winner of a trick collects those cards and gains the associated card points, and then proceeds to lead the next trick.

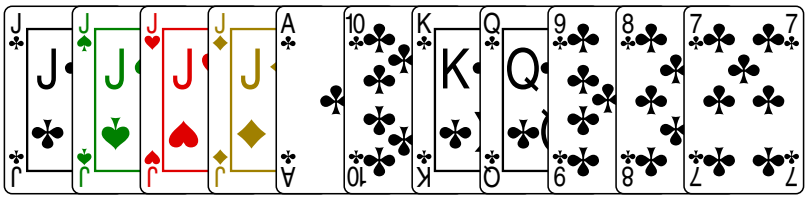

Figure A.1: Trump rankings for a clubs suit game.

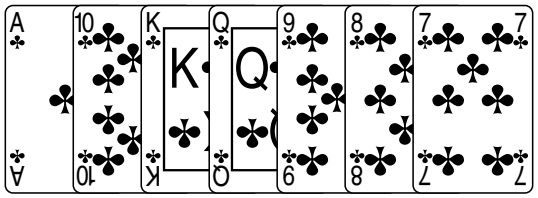

Figure A.2: Non-trump ranking for suit and grand games.

Null games are special in that there is no trump, card points are unimportant, and the ranking of each card reverts to the standard bridge/poker ordering, as in Figure A.3. The soloist's objective in a null game is to avoid taking any tricks. If the soloist wins even one trick then she loses.

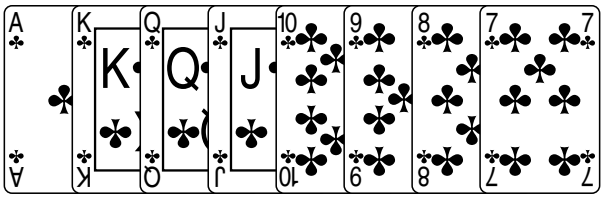

Figure A.3: Card ranking for null games.

Table A.1: Types of Skat games.

Game type	Trump	Objective	Base value
♢	Jacks + ♢	$\geq$ 61 card points	9
♡	Jacks + ♡	$\geq$ 61 card points	10
♠	Jacks + ♠	$\geq$ 61 card points	11
♣	Jacks + ♣	$\geq$ 61 card points	12
Grand	Jacks	$\geq$ 61 card points	24
Null	–	take no tricks	23
Null-hand	–	take no tricks	35
Null-ouvert	–	take no tricks	46
Null-ouvert-hand	–	take no tricks	59

Game Value

Except for null games, the value of a game is a combination of its *base value* and a *multiplier*. The base value is determined by the trump suit: ♢♡♠♣ correspond to base values of 9, 10, 11, and 12, respectively. Grand games have a base value of 24. The starting multiplier is 1, and is adjusted by counting, starting from the highest trump, how many sequential trump the soloist is *with* or *without*. For example, if the soloist has ♣J, ♠J, ♢J, then she is said to be "with 2" (for a multiplier of 3), since she is holding the two highest trump (♣J, ♠J), but missing the third-highest (♡J). Thus the lowest possible multiplier is 2, since the soloist must always be either with or without at least 1. Note that cards in the skat are always included in this counting, so one could conceivably play a suit game "with 11".

The soloist gains an additional multiplier if she acquires at least 90 card points (rather than the 61 needed to win), and in this case is said to have played the defenders "schneider". If the soloist wins all the tricks (not just all the card points), then she has played the defenders "schwarz" (in addition to schneider), for another multiplier.

Announcing a game without looking at the skat constitutes a *hand* game. If the soloist is playing a hand game then she may also announce "schneider", "schwarz", or "ouvert". Each announcement implies the previous ones, with ouvert being the most restrictive — not only must the soloist win

all of the tricks, she must reveal her hand after declaring the game. Failing to make the defenders schneider or schwarz after announcing to do so results in a loss. Each of these announcements adds an extra multiplier, in addition to the normal multiplier gain from actually achieving the stated condition. That is, playing "clubs-hand-schneider" successfully will result in 3 bonus multipliers: one for playing hand, one for announcing schneider, and one for playing the opponents schneider.

Upon winning a game, the soloist gains points equal to the value of the game — not the value of her bid — plus 50 points for winning. Losing a game results in the soloist losing 50 points and *twice* the value of the game. In the event of a loss, the defenders gain (typically) 40 points for successfully defeating the soloist. The defenders lose no points when the soloist wins.

If the soloist inadvertently plays a game that is not worth as much as her bid (say, because she played a hand game and an unfortunate jack was secretly in the skat, or because she was hoping to make the defenders schneider but failed to do so) then the multiplier is increased until the game value is at least as high as the bid. This is not a concern for null games, since the value of a null game is always known at the time of declaration.

Note that the defenders are not the only players who can be made schneider — if the soloist fails to make 31 or more card points, then not only does she lose, she loses schneider (and similarly for being made schwarz).