Solving Imperfect Information Games Using the Monte Carlo Heuristic

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Howard James Bampton

August 1994

Contents

List of Figures

Chapter 1

Introduction

1.1 Why study games?

There are a number of reasons to study games. First, there is an historical precedent of studying games in the Artificial Intelligence field. For example, Samuel's checker player [22] was one of the earliest computer players, and Deep Thought [2] is one of the best chess players, human or computer, in the world. Secondly, lessons from studying game-tree search apply to heuristic searches in general, which are a major part of the Artificial Intelligence field. Thirdly, real world problems are often simulated with a simplified set of “rules”, comparable in complexity to those found in some games. Finally, games are a source of entertainment, and commercially, computer games are a large and rapidly growing market.

1.2 Perfect and imperfect information games

One of the ways games are classified, is whether or not they are perfect or imperfect information games. A *perfect information game* is one in which all players know the true state of the game. Games such as checkers, chess, and tic-tac-toe are examples of perfect information games. In an *imperfect information game*, a player does not know everything about the game's state. Typically, the unknown information is the exact position of the opposing player(s). Card games such as poker, hearts, and bridge are examples of games with imperfect information. In poker, for example, you know the cards you hold, but you do not know what the other players have in their respective hands. A more precise definition of perfect and imperfect information games is given in chapter 2.

1.3 Solving games

When solving a perfect information zero-sum game, well known algorithms exist, such as minimax [14, 24, 29], alpha-beta [10, 26], and MAXN [12, 14, 18]. However, these algorithms do not work for imperfect information games. Previous work by Koller & Megiddo [11] and Blair & Mutchler [6] has shown that solving imperfect information games is NP-hard. (See section 2.2 for more details of this result.) In order to search such trees in practice, one has to either resort to special cases, or fall back on heuristics.

We choose to create an heuristic, which we call the *Monte-Carlo heuristic*, to deal with imperfect information games. The Monte-Carlo heuristic will be described in greater detail in section 2.4. When applied to the play of the cards in the game of bridge, the heuristic can be stated simply as follows. In bridge, each of the three active players sees their own cards, plus those of the fourth, inactive player (the so-called "dummy"). When evaluating a player's choices, the Monte-Carlo heuristic randomly deals the remaining unseen cards to the other two active players' hands. The game is then searched using minimax as if these cards were the ones actually dealt, and as if all hands were known to each player. This is done a number of times and the move that is best on average is chosen. Note that the number of iterations to do is specified by the user, not the heuristic. The heuristic's implementation for bridge is covered in section 4.4 in greater detail.

1.4 Goals of this thesis

There are four goals of this thesis: first, to show the theoretical behavior of the heuristic; second, to verify the validity of the Monte-Carlo heuristic in practice; third, to determine a set of bridge specific advice about the number of iterations to use; and last, to generalize the bridge specific advice as much as possible.

1.5 Method

1.5.1 Why bridge?

There were several reasons for choosing the card game bridge to test out the heuristic. Bridge is a good example of an imperfect information game. Additionally, the rules are simple and well known also the game is not a toy game, yet it can be reduced to a not overly large game, by varying the number of cards in the deck. Finally, there already exists a basic framework of code that did much of the lower level details, as well as a set of problems with human generated answers already available.

1.5.2 Theory

The concept of the Monte Carlo heuristic is well known in the game theory community, but is not known to the AI community at large. In addition, it is not described in a clear and concise manner anywhere that we know of.

After N iterations, the heuristic picks a move which may be right or wrong. When considering $\lim_{N\to\infty}$, the possible behaviors which the heuristic can exhibit when examining a given tree include:

- convergence on a correct answer
- convergence on an incorrect answer
- not converging on an answer while flip-flopping between two correct answers

- not converging on an answer while flip-flopping between two incorrect answers
- not converging on an answer while flip-flopping between a correct and incorrect answer

Chapter 3 presents, for each of the above cases, a tree on which said behavior occurs. Thus, all five behaviors are realizable.

1.5.3 Experiment

The experiment, which will be described in greater detail in section 5.4, is intended both to verify the validity of the heuristic in practice, and to identify factors that influence the convergence of the heuristic on a correct answer. The experiment involves examining the behavior of the heuristic when applied to the 5 choices that are made each trick (1 card per player, plus the initial choice of suit to lead), on 10 different hands of 6 or fewer cards, for a total of 300 decisions, 169 of which were non-trivial. The details of the problems used are covered in greater detail in section 5.3.

1.6 Previous work

1.6.1 Theory

Zermelo [29] developed the theory for 2-player, zero sum, perfect information games. Von Neuman [27] extended the theory to imperfect information games. Kuhn [14] extended the theory to n-player games. Classic game theory textbooks include those by Luce & Raiffa [17], Shubik [25], and Rasmusen [21].

1.6.2 Practical implementations

Shannon [24] developed the concept of a static evaluator as a substitute for exhaustively searching a game tree. He suggested that when one has searched as far as one wishes to in the tree (possibly not searching at all), the static evaluator function is called to estimate the minimax value of the underlying tree. If the node being evaluated by the static evaluator is a leaf, the true value is returned instead of an estimate. Using a static evaluator allows one to examine larger trees with fewer resources (memory and time, typically) than exhaustively searching the tree, at the cost of potentially lower accuracy.

A significant improvement in game tree search implementations, alpha-beta (in its form with "deep cutoffs"), was first described by Slagle and Dixon [26]. Knuth and Moore [10] analyze the theoretical behavior of alpha-beta and give an excellent account of its history.

For perfect information games, computers are doing quite well in many games. In chess, Deep Thought [2] is ranked as a grandmaster. A recent blitz tournament in Munich between Fritz3 [19] and 17 top grandmasters had Fritz3 actually beating the world champion.[1] In a short Backgammon game, the best computer program beat the best rated human player [5]. The second highest rated checkers player is Chinook [23]. Finally, the best Othello (Reversi) player is a computer [15].

Several attempts have been made to cope with imperfect information games. Solving one player imperfect information games has been shown to be NP-complete [6, 11]. A linear time algorithm for imperfect information games, IMP-minimax, has been developed. In single player games with a certain natural property called perfect recall, IMP-minimax computes an optimal strategy [6, 11, 28] For multi-player imperfect information games, it is NP-complete to determine whether or not an n-player imperfect information game with perfect recall has what is called a pure-strategy equilibrium point [7]. Koller & Megiddo [11] examined the complexity of finding min-max strategies for 2-player, zero-sum games both with and without perfect recall. Brown [8] used the concept of using multiple iterations of playing a game to develop a strategy, although his work was aimed towards games in the so-called "normal-form".

The Monte Carlo heuristic, the subject of this thesis, is well-known to the game-playing commu-

[1]While blitz games favor computers, this match is notable because it involved a (modified) commercial chess program running on a pentium-based personal computer.

nity, although we have no specific references to cite. Its essence, as applied to the card game bridge, is to deal the unknown cards randomly and then to solve the resulting perfect information game using minimax (or alpha-beta), finally picking the move that is best on average. Levy [16] suggested doing something similar for bridge with his "Distributor":

> The 1,000 distributions should themselves be distributed in a manner which reflects all the probabilistic data available at the time of their creation. ... The 1000 hands generated by the Distributor represent a kind of probability distribution of the way that the concealed cards lie.

Our work differs from Levy's in several aspects:

- Our work is not specific to bridge.
- Levy confuses hand and distribution.
- In averaging, Levy does not weigh each hand by its correct probability.

Barr [4] and later Owens [20] used the aforementioned concept of distributing the unknown cards randomly, and then applying a heuristic (or one of a suite of heuristics in Owens' case) to the resulting hands to determine a move. In our experiments, we eliminated this second level of heuristic in favor of exhaustively searching the resulting tree.

1.7 Contributions of this thesis

The contributions of this thesis are:

- the development of a heuristic that is suitable for imperfect information games
- the examination of the theoretical behavior of the heuristic
- an experiment that:

  - further validates the usefulness of the heuristic in practice
  - identifies and examines the influences that bridge specific factors have on determining the rate of convergence of the heuristic on a correct answer
  - suggests some game independent as well as game specific performance characteristics that can be used in future work

- identification and implementation of adaptations of common algorithms, such as alpha-beta, that are needed to apply them to games such as bridge, where there is not a strict alternation of players.

Chapter 2

The Monte Carlo Heuristic

This chapter provides definitions which are needed to understand the heuristic, some background on previous work in this area, and a formal statement of the heuristic.

2.1 What is imperfect information?

In order to describe the heuristic, several terms need to be defined first. The remainder of this section is taken almost verbatim from Blair & Mutchler [7].

By a tree, we mean a rooted tree with an associated parent function. With regard to trees, we use without explanation terms like *node*, *edge*, *path*, *depth*, *root*, *leaf*, *interior node*, *parent*, *child*, *ancestor*, and *descendant*. (A node is both an ancestor and descendant of itself.) See any standard text on data structures (for example, [1]) for definitions of these terms. We follow with minor variation the standard game-theory terminology and notation [17, 21, 25] introduced in [14].

An n-player game ? consists of:

- A finite tree $\mathcal{K}$ called the game tree. The edges below any interior node x in $\mathcal{K}$ are the alternatives from x.

- A partition of the interior nodes in $\mathcal{K}$ into $n+1$ classes: the chance nodes and the player-k nodes, for k from 1 to n.
- For each chance node x in $\mathcal{K}$, a probability distribution on the alternatives from x.
- For each k, $1 \leq k \leq n$, a partition of the player-k nodes in $\mathcal{K}$ into information sets such that for any nodes x and y in the same information set:
  - The number of children below x equals the number of children below y.
  - If $x \neq y$, then neither node is an ancestor of the other.
- For each k, $1 \leq k \leq n$, a payoff function h_k which maps for player k from the leaves of $\mathcal{K}$ to the real numbers.

Throughout we reserve the symbols ?, $\mathcal{K}$ and h to refer to the game, game tree and payoff function, respectively, under current consideration.

To see that the above definition captures our informal notion of an n-player game, think of the root of $\mathcal{K}$ as the initial position of the game. To play the game means to follow a root-to-leaf path in the tree, with each edge on the path corresponding to a single move in the game. If a chance node x is encountered during the play, then "nature" will determine the edge below x in the root-to-leaf path, at random and according to the probability distribution associated with x. If a player-k node is encountered, then player k will choose the edge (next move). The outcome of the game for player k is the k^{th} component of the payoff vector $h(w)$ at the leaf w reached by the play.

We have yet to comment on the interpretation of information sets in the above definition. First we need to define what we mean by a "strategy."[1] A strategy π_k for player k on $\mathcal{K}$ is a function on the player-k nodes in $\mathcal{K}$, such that for any player-k nodes x and y in $\mathcal{K}$:

[1]Game theorists will recognize our definition of "strategy" as what they would call a "pure strategy," which we focus on here.

- $\pi_k(x)$ is a child of x.
- If x and y are in the same information set, $\pi_k(x)$ and $\pi_k(y)$ are the same alternative (i.e., if $\pi_k(x)$ is the j^{th} child of x, then $\pi_k(y)$ is the j^{th} child of y).

A strategy π in an n-player game ? is an n-element vector whose k^{th} component, π_k, is a strategy for player k.

"What a player knows" is reflected in the strategies available to the player, which are determined by the information sets. If two nodes x and y are in the same information set, then the player "cannot tell them apart," because by definition the player's strategy must be the same (choose the jth child, for some fixed j) on the two nodes. Thus, when there exists an information set with more than one node in it, the game is said to exhibit imperfect information.

Chess is a game of perfect information: the state of the game is described by the positions of the pieces and whose turn it is, and this information is available to both players. Backgammon is also a game of perfect information, but includes chance nodes: at certain positions, the next position in the game is selected by rolling the dice. The card game bridge is a game of imperfect information. The first move of the game is to deal the cards at random. Each player knows the contents of the player's own hand, but the contents of the other players' hands are revealed only gradually, as cards are played one by one.

The quality of a strategy is measured by its expected payoff, which, in turn, depends on the probability of reaching leaf nodes. Given strategy π on game tree $\mathcal{K}$, the probability of node x under π, denoted $p_\pi(x)$, is defined to be the product of the probabilities of the arcs on the path from the root to x, with each arc below a non-chance node granted probability 1 or 0 depending on whether or not π selects that arc. The expected payoff under strategy π for player k, denoted $H_k(\pi)$, is defined to be $\sum p_\pi(w)\, h_k(w)$, where the sum is over all leaves w in $\mathcal{K}$.

For example, the expected payoff for the strategy indicated by thick lines in Figure 2.1 is $\frac{34}{6}$ +

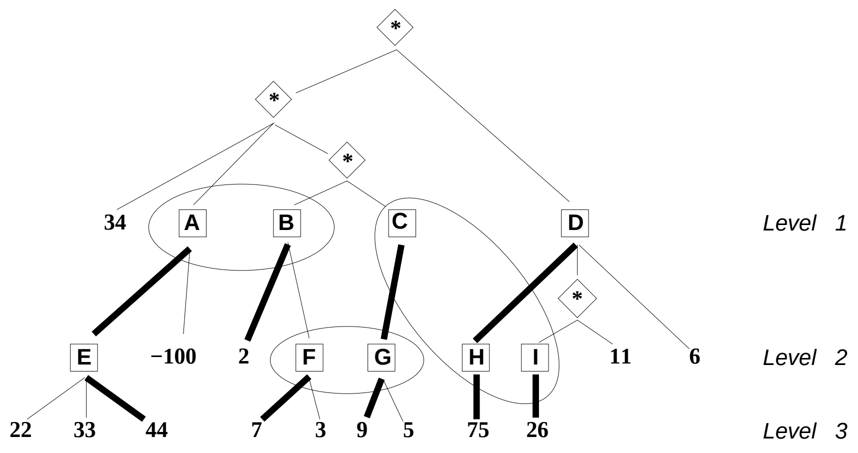

Figure 2.1: A one-player game tree. For any chance node (labeled *), the arcs directly below it are equally likely; player nodes are labeled with letters; and leaves are labeled with their respective payoff values. Ellipses are used to show information sets that contain more than one node. The thick lines indicate the strategy selected by `IMP-minimax`.

$\frac{44}{6} + \frac{2}{12} + \frac{9}{12} + \frac{75}{2}$.

A player-k strategy π_k is <u>optimal for strategy π</u> if for every player-k strategy α_k, we have $H_k(\pi) \geq H_k(\alpha)$, where α is the same as π except that the k^{th} component of π is π_k while the k^{th} component of α is α_k. A strategy π is an <u>equilibrium point</u> if each component π_k of π is optimal for π. Thus, in an equilibrium point, no player can strictly improve her expected payoff by a unilateral change in strategy. A <u>solution</u>[2] to an n-player game ? is an equilibrium point for $\mathcal{K}$. Clearly, for one-player games a solution is a strategy that maximizes (over all strategies) the expected payoff to the single player.

[2]Game theorists will recognize our definition of "equilibrium point" as what they would call a "pure strategy equilibrium point." While one-player games always have a pure strategy equilibrium point, a solution for multi-player imperfect information games typically requires using a "mixed-strategy." The Monte-Carlo heuristic does not require mixed-strategies and our examples are all single player game trees, so mixed-strategies are not described herein.

2.2 Previous methods for imperfect information games

Brown [8] discusses games in the "normal form," the definition of which is beyond the scope of this thesis. Games represented this way are typically impractical to examine (or are toy problems). He suggested using an iterative algorithm for finding the solution to imperfect information games in normal form.

In the previous section, we discussed games in the extensive (tree) form. For such games, there are three types of strategies. "Pure strategies" are defined in section 2.1. There exist two others—"behavior strategies" and "mixed strategies." Koller & Megiddo [11] developed a polynomial time algorithm for finding optimal behavior strategies in 2-player, zero-sum, imperfect information games with perfect recall. However, the algorithm is impractical to use because the exponent of the polynomial is huge.

Two previous attempts at heuristics should be mentioned. IMP-minimax has been shown to find in linear time an optimal solution for 1-player games with the property of perfect recall [6, 11, 28]. For other games, it is a linear time heuristic of unknown utility. Levy [16] proposes writing a bridge playing program which generates hands in proportion to their distribution frequencies, with extensive use of bridge specific knowledge. The program would then do an alpha-beta search of the game trees resulting from these hands. The move that is best on average would be the one it selects.

2.3 Definitions

For each node x in a game tree, we define two functions, $q(x)$ and $p(x)$. Function $q(x)$ is given by: $q(x) =$ the product of the probabilities of the arcs between the root of the tree and the node x. If the arc is from a chance node, use the probability associated with the arc; if the arc is from a player node, use the uniform distribution. Function $p(x)$ is given by:

$$p(x) = \frac{q(x)}{\sum q(y)}$$

where the sum is over all nodes in the information set containing x. Note that for each information set I, function p provides a probability distribution on the nodes in I.

To clarify the above, let us examine 3 examples:

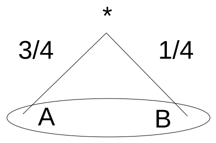

Figure 2.2: A simple tree with one chance node

In figure 2.2 there is a single chance node at the root of the tree. In this figure, $q(A) = \frac{3}{4}$, and

$$p(A) = \frac{q(A)}{q(A)+q(B)} = \frac{\frac{3}{4}}{\frac{3}{4}+\frac{1}{4}}$$

.

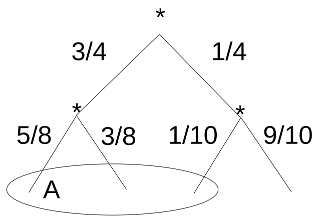

Figure 2.3: A simple tree with multiple chance nodes

In figure 2.3 the three interior nodes are all chance nodes. The probability of generating node A would be:

$$\frac{\frac{3}{4}*\frac{5}{8}}{\frac{3}{4}*\frac{5}{8} + \frac{3}{4}*\frac{3}{8} + \frac{1}{4}*\frac{1}{10}}$$

In figure 2.4 there is a single, top level chance node, and two player nodes. The probability of generating node A would be:

$$\frac{\frac{3}{4}*\frac{1}{2}}{\frac{3}{4}*\frac{1}{2} + \frac{1}{4}*\frac{1}{3} + \frac{1}{4}*\frac{1}{3} + \frac{1}{4}*\frac{1}{3}}$$

.

A final definition, that of chance-minimax, is needed. This definition is copied from Blair & Mutchler [7], and goes as follows:

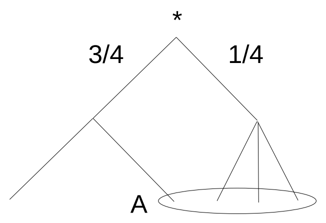

Figure 2.4: A simple tree with player nodes

For two-player games with perfect information, even with chance nodes, a simple modification of minimax computes the value of the game tree [3]:

$$\texttt{chance-mm}\,(x) \quad = \quad \begin{cases} \text{payoff}\,(x) & \text{if } x \text{ is a leaf} \\ \\ \text{F}\,\{\,\texttt{chance-mm}\,(y) \mid y \text{ is a child of } x\,\} & \text{otherwise} \end{cases}$$

where function F is a *max* operator at nodes where Player 1 moves, *min* at nodes where Player 2 moves, and *average* at chance nodes. (See [13, 14, 18] for the extension to more than two players.)

2.4 The Monte Carlo heuristic

With the above definitions, we are now ready to define the Monte Carlo heuristic.

Given an information set I belonging to player k which has *alt* alternatives, and a number of iterations *iter*, create a vector *best* of size *alt* and initialize its elements to 0.

For each iteration, randomly select in accordance with the probability distribution $p(x)$ in section 2.3, a node n from I. With n, do an exhaustive chance-minimax search of the resulting tree as if it were a perfect information game tree. For each alternative m of n, increment the corresponding entry in *best* by the backed up chance-minimax value of m.

The heuristic's suggestion for the best move is the alternative corresponding to the highest entry in *best*. Any ties are broken randomly.

See section 4.4 for how this algorithm is applied to bridge.

Several variations of the Monte Carlo heuristic are possible. A chance version of alpha-beta

[3, 10, 26] can be substituted for minimax, since alpha-beta computes the same value as minimax and is faster. One could use a Monte Carlo version of chance-minimax, and have a single node under a chance node picked randomly (according to the probability distribution associated with that chance node) instead of searching under all nodes under a chance node. This would be faster, but increase the chances of an incorrect result. Finally, one could modify the definition of the function q to assign a probability of 1 for non-chance nodes without invalidating the heuristic.

Chapter 3

Theoretical Behavior of the Monte Carlo Heuristic

The concept of the Monte Carlo heuristic is well known in the game theory community, but is not known to the AI community at large. In addition, it is not described in a clear and concise manner anywhere that we know of. The previous chapter defined the heuristic. This chapter examines its theoretical behavior.

After N iterations, the heuristic picks a move which may be correct or incorrect. When considering $\lim_{N\to\infty}$, the possible behaviors which the heuristic can exhibit when examining a given tree include:

- convergence on a correct answer
- convergence on an incorrect answer
- not converging on an answer while flip-flopping between two correct answers
- not converging on an answer while flip-flopping between two incorrect answers
- not converging on an answer while flip-flopping between a correct and incorrect answer.

This chapter presents, for each of the above cases, a tree on which said behavior occurs. Thus, all five behaviors are can be realized.

Figure 3.1 is a preliminary example to illustrate the ideas of the forthcoming sections.

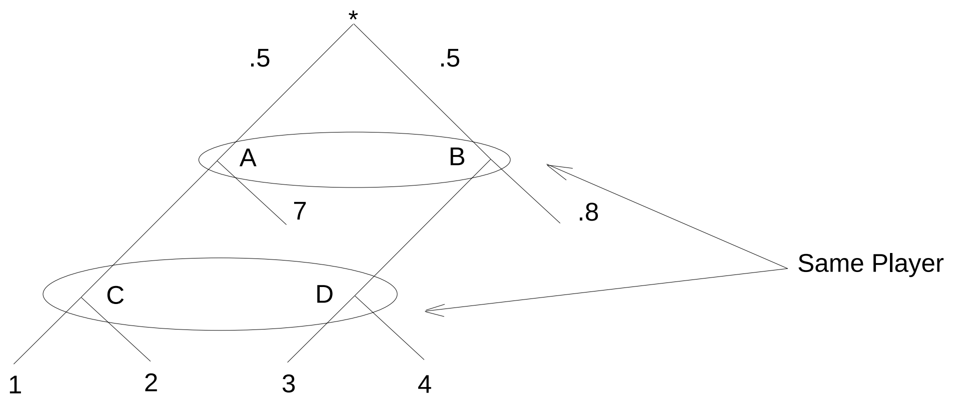

Figure 3.1: A one-player tree with imperfect information

In this tree, as well as the ones later in this chapter, the top node of the tree (marked with a "*") is both the root of the tree, and a chance node. For simplicity, that is the only chance node in any of our examples, although the heuristic can deal with them elsewhere in the tree (see section 2.3 for examples of trees with multiple chance nodes). The numbers along an arc represent the probability of traversing that arc. Ellipses around nodes represent information sets, and numbers at leaves are payoffs.

In this tree, there are four strategies[1]— left from the top information set and then left at the second, left at the top information set and then right at the second, right at the top information set and then left at the second, and right at the top information set and then right at the second. (Obviously, the latter two strategies have the same expected payoff.) Following the left/left strategy would have a payoff of 1 or 3 depending upon whether the actual node that we are at is A or B. The average (expected) payoff for this strategy is the sum over all nodes of the probability of the

[1]At first glance, it would seem that there are in fact only 3, however a strategy must include the choice one would make at all information sets, even if an information set is unreachable under a given strategy.

node times its payoff, thus the left/left strategy's payoff is $.5 * 1 + .5 * 3 = 2$. Similarly, the left/right strategy has payoffs of 2 and 4, for an average of 3, and the right/left and right/right strategies have payoffs of 7 and .8 respectively, for an average of 3.9. The best strategy is any which has the highest average, namely the right/left and right/right strategies. Thus the best decision from the top information set, assuming optimal decisions elsewhere, is to move right from nodes A and B.

To calculate the minimax value of a node, in a one player game one returns either: the value of the node if it is a leaf node, otherwise the largest value of all children of that node (this will result in recursion). Applying this to the above tree, the value of node A is the maximum of the value of node C, and 7. Node C's value is the maximum of 1 and 2. To simplify things, we will write the values of the children of a node as a vector, thus C's would be written as (1, 2). Backing up the value of C to A, A's children would have values of (2, 7). Through a similar process, B's value can be calculated as (4, .8).

The Monte Carlo heuristic uses these minimax vectors when making its choice. After generating a node from the top level information set, it evaluates the underlying subtree as if it were a perfect information tree. Each element of the *best* vector is incremented by the minimax value of the respective child of the node. For example, assume that the heuristic chose nodes A, B, and A for the first three iterations. The *best* vector would then look like (2, 7) after the first iteration, (6, 7.8) after the second, and (8, 14.8) after the third iteration. It would then chose the move corresponding to the highest value, namely, to go right.

Note the distinction between the best decision, which is the one which maximizes the expected payoff when assuming optimal decisions elsewhere in the tree, and the Monte Carlo heuristic's decision, which is based on minimax values.

3.1 Convergence on a correct answer

When given a tree such as in Figure 3.2, the heuristic applied to the top information set will converge

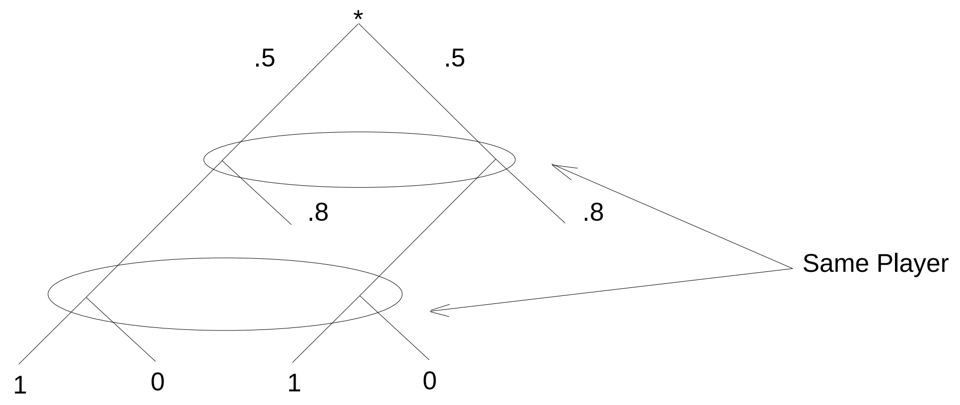

Figure 3.2: Convergence on a correct answer

on the correct value after a single iteration. To see this, note that the *best* vector will have values of (1, 0.8) after one iteration, (2, 1.6) after two, and so on, regardless of which of the two nodes in the top information set is selected by the Monte Carlo heuristic each iteration. Thus the Monte Carlo heuristic converges immediately to the decision "move left from the top information set." This decision is optimal because regardless of at which node in the top information set we actually happen to be, the same strategy, namely left/left, is optimal. Its expected payoff is 1. The expected payoff for the left/right strategy is 0, for the right/left strategy is .8, and for the right/right strategy, .8.

3.2 Convergence on a incorrect answer

In the tree depicted in Figure 3.3, the left/left and left/right strategies have expected payoffs of .5, while the right/left and right/right strategies have expected payoffs of .8. Thus the correct decision from the top information set in the above tree is to go right, getting a payoff of 0.8. However, the heuristic will incorrectly decide to go left. This happens because of the heuristic's choice of a node from the top information set. If it chooses the left node, when it searches its left subtree, it will back up a value of 1. Similarly, when it picks the right node in the information set, the search of its left subtree will also return 1. The heuristic will therefore choose to move left from the top information

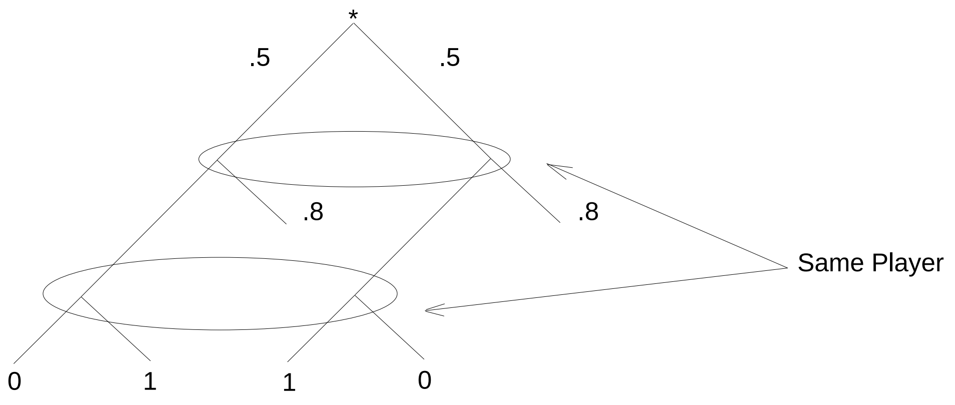

Figure 3.3: Convergence on a incorrect answer

set, because the *best* vector's value, as in the previous example, is incremented by (1, 0.8) after each iteration.

3.3 Non-converging, two correct answers

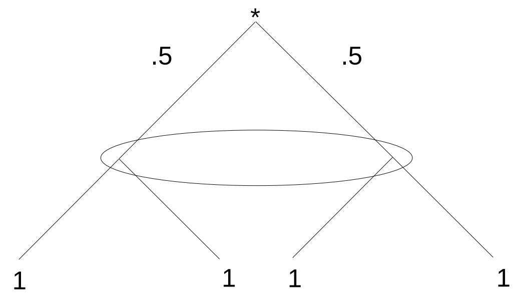

Figure 3.4: Trivial non-converging between two correct answers

Figure 3.4 is an example of a trivial case. The *best* vector is incremented by (1, 1) after each iteration, so the heuristic will choose one of the two strategies (left and right) randomly. Both decisions are correct, of course. Similar trees can be generated for the other non-converging cases; however, we will use the interesting ones instead.

The tree in Figure 3.5 has two strategies, with expected payoffs of .5. Thus the optimal strategy

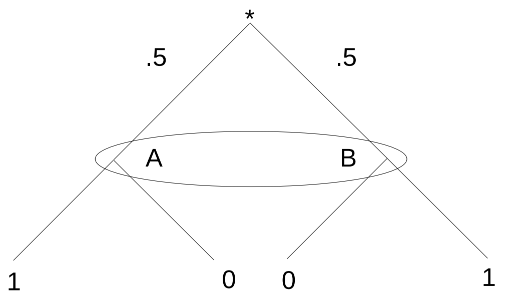

Figure 3.5: Nonconvergence between two correct answers

from the information set is to pick either one.

To see how the Monte Carlo heuristic operates on this tree, let us consider an example. Assume that 5 iterations are done, and the nodes which are generated from the information set are A, B, B, A, and A. After 1 iteration, the *best* vector will have a value of (1, 0) and the heuristic will decide that moving left is best. By the third iteration, the *best* vector will have a value of (1, 2), and the "best" move will now be to move right. By the fifth iteration, things will have flip-flopped again, and moving left will be better according to the heuristic, since *best* will be (3,2). Whichever node has been randomly selected more often will determine which move is better, even though the moves are both equally good (or bad).

If we takes the difference between the value of the two elements of the *best* vector, and examines it over time (each iteration), we are left with an example of a *random walk*: A particle that starts at the origin and at each time step, either goes up one unit (with probability p) or down 1 unit (with probability $1 - p$). The random walk for the example described above has $p = 0.5$, and would be graphed as in Figure 3.6:

Thus the random walk shows the behavior of the Monte Carlo heuristic— if the random walk is above the x-axis, the Monte Carlo heuristic chooses to move left from the information set, while if the random walk is below the x-axis, the heuristic chooses to move right.

Convergence of the Monte Carlo heuristic to a stable decision can also be stated in terms of the

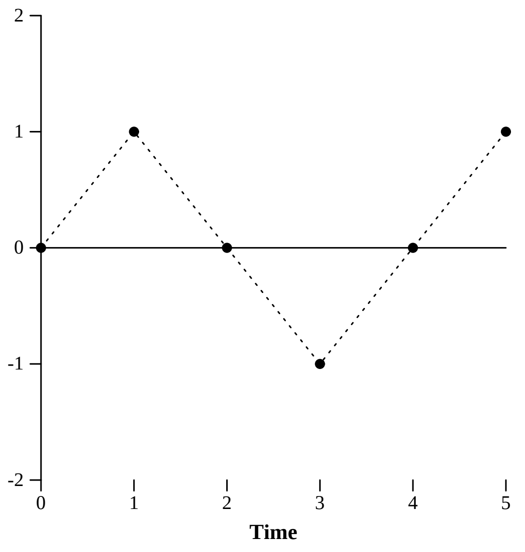

Figure 3.6: A random walk

graph of the random walk corresponding to the difference between the best and second best choice. If at time t we are on one side of the x-axis, and for all times greater than t, we stay on that side of the x-axis (without changing the best choice), one can say that the heuristic has converged on the choice corresponding to that side of the x-axis. Results on page 348 of Feller [9] state that with probability equal to 1, a random walk with $p = 0.5$ crosses the x-axis infinitely many times. Therefore, since the random walk never stays on the same side of the x-axis, the Monte Carlo heuristic never converges.

3.4 Non-converging, two incorrect answers

In the tree depicted in Figure 3.7, there are twelve strategies— left/left/anything, left/right/anything, center/anything/anything, right/anything/left, and right/anything/right.[2] Their expected payoffs are: 2.5, 2, 3, 2, and 2.5 respectively. Therefore, the optimal decision from the top information set

[2]A strategy must list a choice from each information set, even if the information set can never be reached under that strategy. For simplicity, we will list the unreachable components of a strategy together— i.e. rather than listing left/left/left and left/left/right, we will list them both as left/left/anything. Here the first component of each of these triples refers to the top information set, the second component refers to the bottom left information set, and the third component refers to the bottom right information set

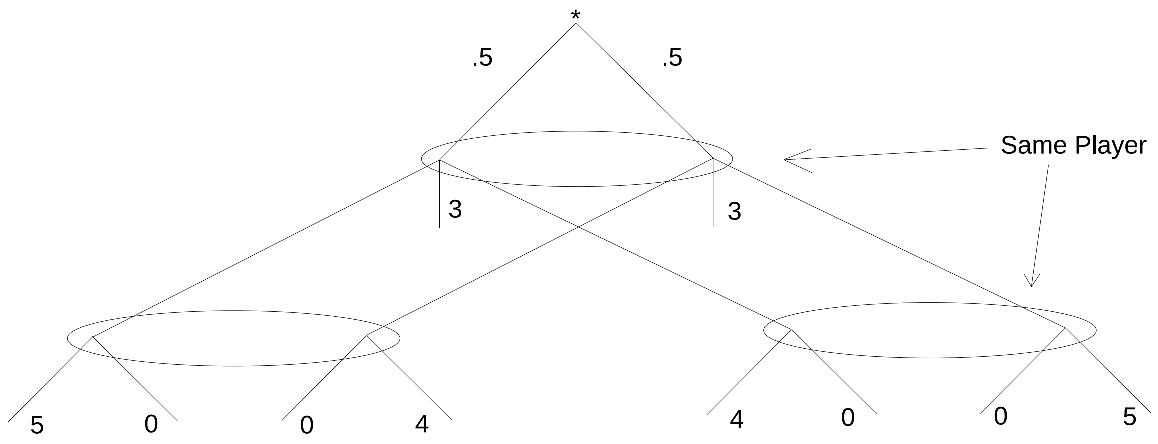

Figure 3.7: Nonconvergence between two incorrect answers

is to choose the central alternative, yielding the expected payoff 3.

The Monte Carlo heuristic will compute the *best* vector with the left and right elements being incremented by 4 or 5 each time, while the center one is incremented by 3. This will result in it picking either the left or right choice, rather than the correct choice of making the center move. If one examines the difference between the first and third values of the *best* vector over time, one again sees an example of a random walk. As before, random walk theory shows that the heuristic will fail to converge, flip-flopping between the left and right choices.

3.5 Non-converging, correct and incorrect answers

For the tree in Figure 3.8, there are 4 strategies— left/left, left/right, right/left, and right/right, with expected payoffs of 1, .5, 1.5, and 1.5 respectively. Thus the correct decision from the top information set is to move right, yielding expected payoff 1.5.

The heuristic, however, will compute a *best* vector which will be incremented by (2, 1) or (1, 2). Depending upon whether the left or right node in the information set is generated more often, the heuristic will alternate between moving left (incorrect) and right (correct). Again, examining the difference between the two elements of the *best* vector over time is an example of a random walk.

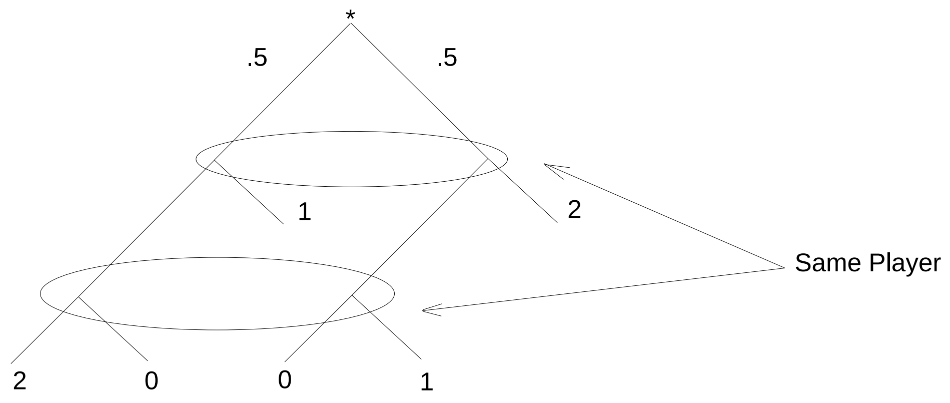

Figure 3.8: Nonconvergence between a correct and incorrect answer

As before, random walk theory shows that the heuristic will fail to converge, flip-flopping between the left and right choices.

Chapter 4

Applying the Monte Carlo Heuristic to Bridge

4.1 Why bridge?

There were several reasons for choosing the card game bridge to test out the heuristic. Bridge is a good example of an imperfect information game. Additionally, the rules are simple and well known also the game is not a toy game, yet it can be reduced to a not overly large game, by varying the number of cards in the deck. Finally, there already exists a basic framework of code that did much of the lower level details, as well as a set of problems with human generated answers already available.

4.2 Rules of bridge

The following information, describing how bridge is played, is taken verbatim from the user manual of a commercial bridge-playing program called BridgePal but with any information about the program itself omitted.

A hand of bridge consists of 4 different activities:

- Shuffling and dealing
- Bidding
- Playing the hand
- Scoring

Bridge is played with a 52-card deck composed of 4 suits of 13 cards each. Cards are ranked from Ace (highest), King, Queen, Jack, and 10 to 2 (lowest). All cards are dealt to the 4 players, so each player has 13 cards in his "hand."

Bidding is the way you describe your hand to your partner, and make a guess about the strength of your combined hands. The highest bid becomes the "contract," or goal of the offensive team. If you can fulfill your contract, your team scores points. If you fail to make the contract, your opponents will score.[1]

You may bid a suit or "no-trump." If the highest bid is a suit, that suit becomes the "trump" suit. Cards in the trump suit are something like "wild cards." If "no-trump" is the final bid, then there are no "wild cards."

Suits are ranked from lowest to highest: clubs, diamonds, hearts, spades and no-trump. So a bid of "1 diamond" is higher than a bid of "1 club." The bidding begins with the dealer and continues clockwise until 3 players in a row say "pass" (pass means "no bid"). Each bid must be higher than the previous one: for example, 1 club, 1 heart, 1 No-trump, 2 diamonds, etc. Each bid is supposed to give your partner more information about the strength of your hand.

The team that makes the highest bid wins the contract. The suit from the highest bid becomes the trump suit (unless it was no-trump). The player on the high bidding team who first mentioned

[1]There are other aspects associated with bidding, including psychological effects, which can maximize the benefit for your side and can help place your opponent in unfavorable situations.

the trump suit (or No-trump) is called Declarer. The high bidding team is on the offense and the Declarer has to "play" the hand by himself.

After the bidding is finished, the person to the left of Declarer plays the first card (opening lead). The Declarer is on offense and he plays both his own and his partner's cards. Declarer's partner is called the Dummy. The Dummy puts all his cards face-up on the table, immediately after the opening lead.[2]

Declarer then decides which card to play from Dummy. The second defensive team member plays next, and finally Declarer plays from his own hand. The highest card played in the same suit as the opening lead wins the "trick." A trick is a round of 4 cards, one from each player in a clockwise direction. If someone plays a card from the trump suit (a "wildcard"), then the highest trump card wins. But you must follow suit if possible (you may play the trump suit only if you don't have any cards in the suit that was led–the suit of the first card played on each trick).

The hand that wins the trick has the lead and plays the first card on the next trick. Play proceeds in this manner until all cards have been played and the 13 tricks are divided between the two teams.

For brevity, the information about scoring has been omitted.

Both bidding and scoring are beyond the scope of this heuristic.

4.3 Bridge specific definitions

4.3.1 State

A state consists of the following information:

- current player
- current trick

[2] If all cards are face-up, it is called Double Dummy.

- number of tricks left
- number of tricks won by the North/South team
- the highest card played in the current trick and who played it
- the cards played in the hand (but not who played them)
- the cards held by the current player and the dummy
- the cards held by the remaining two players (only if during an iteration of the heuristic).

4.3.2 N-th hand

- 0th hand- the initial choice of suit to lead in a trick
- 1st hand- the choice of cards to lead, given the suit
- 2nd hand- the second player's choice of cards
- 3rd hand- the third player's choice of cards
- 4th hand- the fourth player's choice of cards

4.3.3 Alternative

An alternative is either a suit (for 0th hand states), or a suit and card.

4.4 The Monte Carlo heuristic applied to bridge

4.4.1 Applying the definition

Below we describe how the heuristic is applied to bridge. This definition is a straight adaptation of section 2.4.

Given an information set I (which consists of a state as described in section 4.3.1 without the cards held by the two other active players filled in) belonging to player k which has alt alternatives, and a number of iterations $iter$ (200 in our experiments), create a vector $best$ of size alt and initialize its elements to 0.

Let the function P be the probability distribution corresponding to a fair deal of the cards in question.

For each iteration, first randomly deal the unknown cards into the two unknown hands using P to acquire a state S. Then do an exhaustive minimax search of the game tree rooted at state S as if it were a perfect information game tree. For each alternative m from state S, increment the corresponding entry in $best$ by the backed up minimax value of m.

When all iterations are complete, the "best" move is the one corresponding to the highest valued entry in $best$. Ties were broken in favor of the highest ranked card.[3]

4.4.2 Refinements

Two refinements were made to the heuristic when it was applied to bridge. Instead of minimax, negamax alpha-beta [10] (which computes the same value as minimax, but possibly faster) was used. Due to the fact that bridge does not have a strict alternation of players, and the introduction of 0th hand moves, the alpha-beta code was further modified to change the alpha and beta cutoffs appropriately.

A second modification was to introduce bucketing of alternatives. In card games like bridge, when a player has two cards such as the 3 and the 2, it makes no difference to the minimax value of the game which card is played— they are both "equal". As cards are played, additional cards become equal- for example, once the K has been played, the Q and A are equal. In order to reduce

[3]In some situations, the highest card of a set of equally good cards is the correct play— such as deciding between the A and K. In other situations, such as discards, the lower choice is typically the correct one.

the size of the game tree searched, all alternatives were bucketed using a table lookup on the union of the cards held by the player in that suit, and the cards played in that suit. The highest exemplar held by the player from each bucket was then used to represent all cards in that bucket in the alpha-beta search.

Chapter 5

Experiment

5.1 Motivation

We decided to perform an experiment to check the validity as well as the practicality of the Monte Carlo heuristic, rather than to attempt a proof of its average-case performance. If the heuristic were inaccurate on average, this would rapidly become evident.

The experiment has two goals— to verify the validity of the Monte Carlo heuristic in practice, and then to identify factors that influence the convergence of the heuristic on a correct answer.

5.2 Environment

The problems were run on 31 Sun IPX's with 16 Megabytes of physical memory, and 90 Megabytes of swap space running SunOS 4.1.3. The code was written in Austin Kyoto Common Lisp version 1.615. The Lisp code was compiled via gcc 2.3.3, and the default compile options. Adding optimization and other optimizing flags to the compile options produced mixed results with regards to speedup; apparently the Lisp to C translation of AKCL does not produce code that gcc can optimize to any great extent. Increased garbage collection may have also masked any speedups as a result of more efficiently compiled code.

5.3 Game setup

The problems generated for the experiment had the following characteristics.

The deck of cards was dealt to all four players so that the North-South team's hands were given the advantage. In order to have an advantage, North-South team's combined high-card points (Ace = 4, King = 3, Queen = 2, Jack = 1) had to be at least 23. In order to have hands similar to those commonly published which might also have produced a no-trump bid, the North-South team could not have more than eight cards in the two hands together of either Spades or Hearts.

From each deal, there are 65 problems— 5 decisions (0th hand/suit, and 1st through 4th hand choice of card) per trick times 13 tricks. In order to create the problems, an answer for each decision was made by Dr. Mutchler, a tournament-class bridge player. When there was more than one "correct" answer, all of them were recorded, and it was assumed that the first choice listed was the one actually done for purposes of later play. (See section 6.5 for further discussion of what is "correct.")

A total of 10 bridge hands were dealt, resulting in a total of 650 problems.

North was always the dummy. A player knew only his cards and dummy's, or North's and South's cards if he were dummy. The cards played since the deal, what hand it is, who played the highest card in the current trick, what suit was led in the current trick (if it was not 0th hand), and the number of tricks remaining are known to the player. Who played any card (other than the highest one in the current trick) is not known, because it is ignored by the Monte Carlo heuristic. The contract that is being met and the sequence of bids that lead to it is also not known, again because it is not relevant to the Monte Carlo heuristic.

5.4 Experiment design

The experiment involved running the heuristic on hands with 6 or fewer cards per player— 300 of the original 650 problems. The machines were used during generally light access times. Each problem was run 3 times sequentially from within the same AKCL session. For each run, if it degenerated into a single choice after bucketing or was already degenerate, the problem was recorded as such; if it wasn't, 200 iterations of the sampling heuristic were done. After each iteration, the cumulative CPU time, the choice with the highest cumulative running total, and the vector of cumulative running totals for all the choices was recorded. The default random seed was used to start the first run.

Of the original 300 problems, 131 were degenerate, and 3 were not completed due to extreme CPU time requirements— roughly 40 days of CPU time for each. The remaining 166 took a total of 146 days of CPU time to complete. The individual problems took anywhere from 0.06 seconds to 1766 seconds per iteration (36 seconds to 12.26 days for 3 runs of 200 iterations).

The purpose of this experiment is to verify the validity of the heuristic, and then to identify factors that influence its convergence on a correct answer.

One should keep in mind that actual minimax values are being computed, rather than using a static evaluator, as was done in Barr [4] and Owens [20]. This explains why there are such large variations in run times and why problems with more than 6 cards per player were impractical. Since there is no static evaluator being used, the only heuristic being used is the Monte Carlo heuristic itself.

Chapter 6

Experimental Results

This chapter presents the results of the experiment. The purpose of this experiment is first to verify the validity of the heuristic, and then to identify factors that influence its convergence on a correct answer.

6.1 Validity

There were initially 300 problems on which the heuristic was tested. A total of 131 of them (81 of which involved hands with more than one card per player) had a single choice left after bucketing (or only had one before bucketing was done). Three problems were not completed because it was estimated that they would require more than a month of CPU time each to finish, leaving 166 non-degenerate, completed problems. Each of these 166 problems was run with 3 sets of random numbers, for a total of 498 problem instances.

6.1.1 Degenerate problems

When there is only a single choice, whether or not it is a result of bucketing, you can predict the Monte Carlo heuristic's choice, so there is no need to run it. However, it is not true that all choices that have been bucketed together are correct— if nothing else, you must randomize choices to prevent the opposition from guessing your cards when you consistently make the same choice from a bucket. For example, if you have the Q and J in a suit, bucketing will merge the two for the game tree search, even thought the other players don't know you have both. If this bucket is the best one to play, you should not always pick the highest (or lowest) one. If you always play the highest card from a bucket, when you play the Q, they don't know if you have the J. If you only had the J, then when you play the J, they can deduce that you don't have the Q.

For all 131 times where there was only a single choice after bucketing, the Monty Carlo heuristic agreed with the human expert.

6.1.2 Non-degenerate problems

There were 166 non-degenerate, completed problems. Here we consider the effectiveness of the Monte Carlo heuristic at the end of all 200 iterations. The remainder of the chapter considers its effectiveness and convergence behavior during the 200 iterations.

Seventeen of the remaining problems were solved incorrectly after 200 iterations on or more of their three runs. Thirteen problems were not solved correctly after 200 iterations on any of their three runs, one was solved right on two of its runs,[1] and three were solved correctly on one of their three runs for a total of 44 instances where it got an incorrect answer. Of particular note is that nearly a third of the incorrect answers were when choosing the suit. Eight different problems were never correctly solved i.e. the heuristic's choice all 200 iterations was incorrect. Three of these problems were 0th hand decisions, two were 2nd hand decisions, and three were 4th hand decisions. Three of them were not solved on any run, 3 were solved right on one run, and two were solved right on two runs. (See Figures 6.1, 6.2, 6.3, and 6.4 for additional details.)

[1]See section 6.7 for further discussion of this problem.

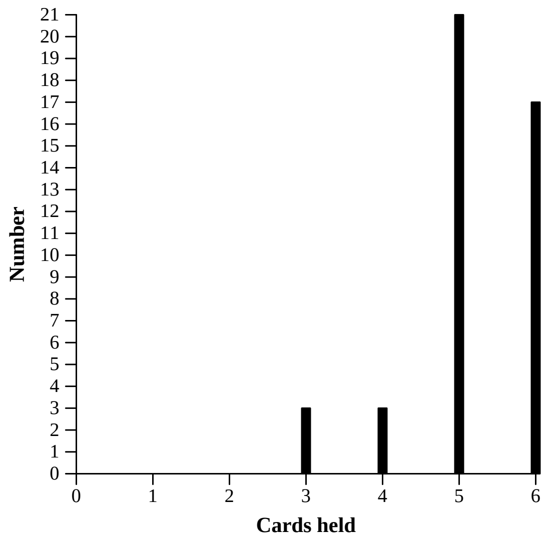

Figure 6.1: Number of problems incorrectly solved after 200 iterations, by cards held

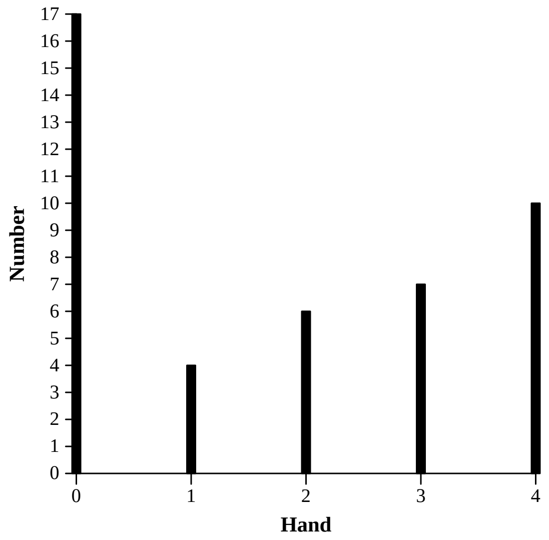

Figure 6.2: Number of problems incorrectly solved after 200 iterations, by hand

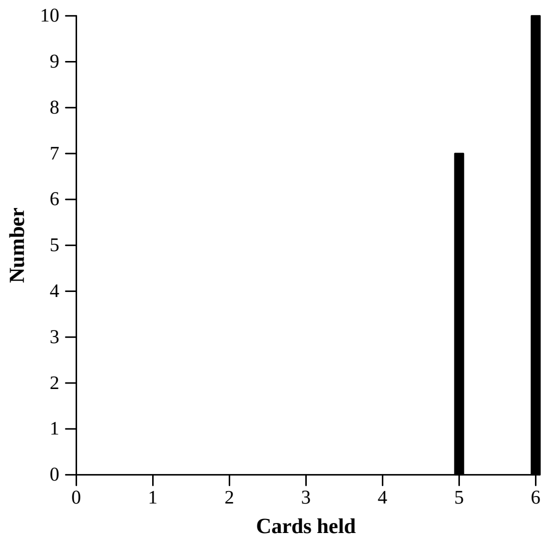

Figure 6.3: Number of problems incorrectly solved all iterations, by cards held

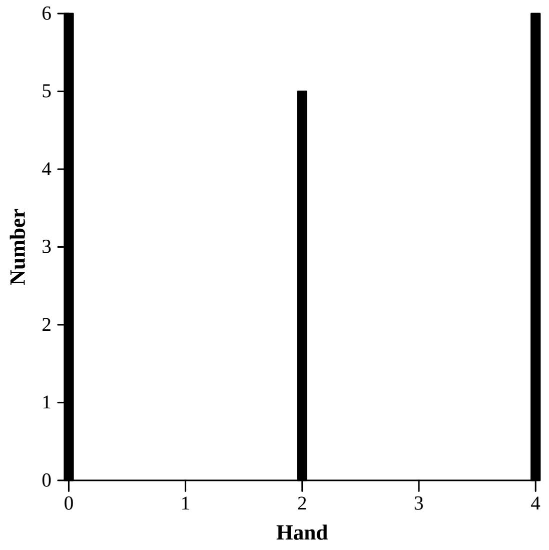

Figure 6.4: Number of problems incorrectly solved all iterations, by hand

6.2 Factors that influence convergence

A main goal of this thesis is to advise the user of the Monte Carlo heuristic how to judge how many iterations to run, before making a decision. Here we present that advice. In each subsection, we:

- State advice we conjectured in terms not specific to bridge (as far as possible).
- Explain how the results supported or denied the advice.

There are two classes of factors that influence convergence: static and dynamic. Static factors are things such as the number of cards, or how strong your team's hand is. They can be calculated before you does any iterations of the heuristic. Dynamic factors are calculated between iterations, and are things such as quiescence or the difference between the two best choices. Static factors will be examined in greater depth in section 6.3. Dynamic factors will be examined in section 6.4.

6.3 Static Factors

To understand the results of the experiment, the results were looked at from several perspectives. For each subclass of problems considered, we graphed the percentage correct versus the number of iterations. For example, Figure 6.5 shows the results over all 498 non-degenerate, completed problem instances. (Throughout this and the following section, only problem instances which were completed but are not degenerate are included in the statistics.) For each curve, 2 points were sought: the point at which the curve first slowed its rate of increase and the point at which the curve greatly flattened out. Both of these points were identified by visual inspection of the graphed data. A point was generated on each graph at each iteration, and then the points were connected with straight lines.

For example, in Figure 6.6 (a blowup of Figure 6.5), the curve rises rapidly for 8 iterations and levels out at about 90% after 13 iterations.

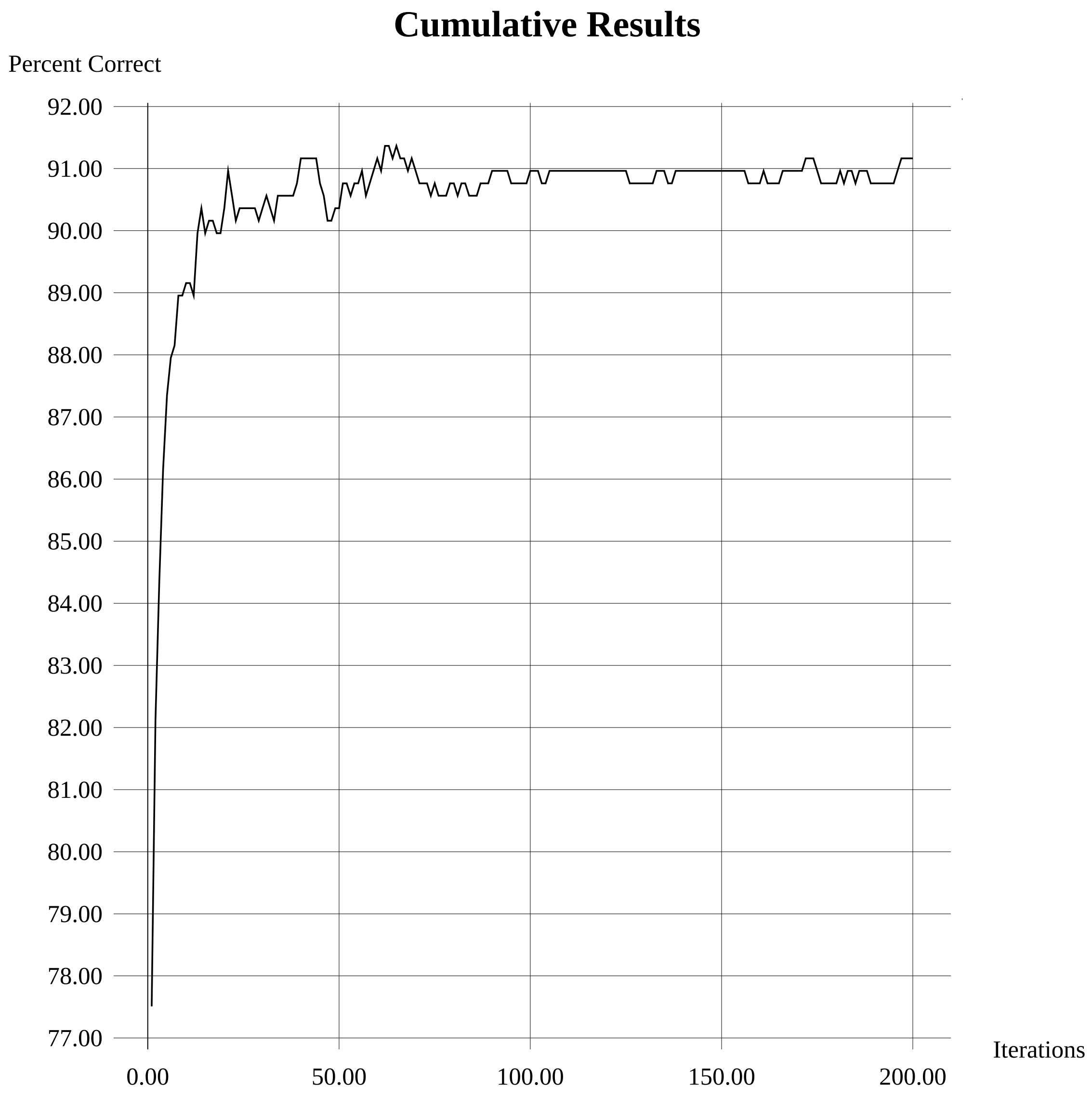

Figure 6.5: Cumulative results over 200 iterations

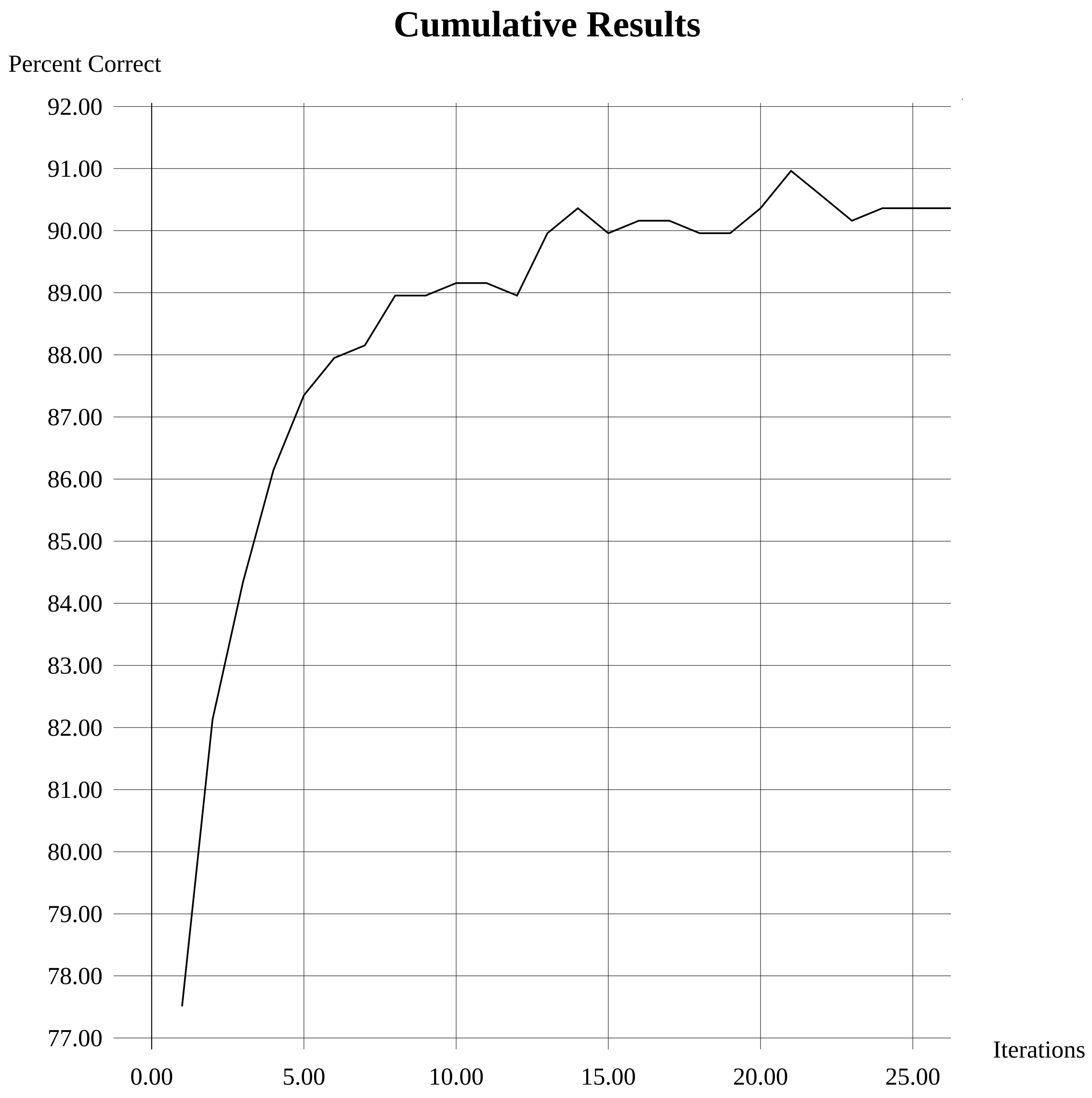

Figure 6.6: Cumulative results over 25 iterations

6.3.1 Simple

You would expect that the information gathered from a single iteration would give substantially better performance than random choice of an alternative would. Similarly, doing a small constant number of iterations should give good results in the general case.

Experimental evidence supports the conclusion that a single iteration can give passable performance, and that a small number of iterations will produce good results overall.

After 1 iteration, roughly 77.5% of the problems are correctly answered. (See Figure 6.5 and Figure 6.6.) Additional iterations will continue to add more than a percent per iteration, until 8 iterations, when the curve hits its first peak around 89%. Beyond 13 iterations, where it passes 90%, the curve all but flattens out.

In summary, for many games, a simple "do x iterations" heuristic may be sufficient. Depending on the influence that the unknown information has on the outcome and the desired performance, even a single iteration may be sufficient.

6.3.2 Moves remaining

You would expect that the farther away from the end of the game a position is, the more iterations will be required to determine the correct choice.

This conclusion also is justified by experimental evidence. Bridge's 13 cards per player and playing one card per turn makes determining the distance to the end of the game easy; you simply count the number of cards in your hand. The problems which involved more cards started with lower accuracies, and generally required more iterations to reach each plateau. This data is shown in Figures 6.7 and 6.8.

For 6 card problems, the percent gain in accuracy per iteration climbs steeply for the first 8 iterations to 76%, and then levels off, eventually reaching 81%. Five card problems rise rapidly for 4 iterations to 83%, and level off around 85% after 8 iteration. For a large number of iterations,

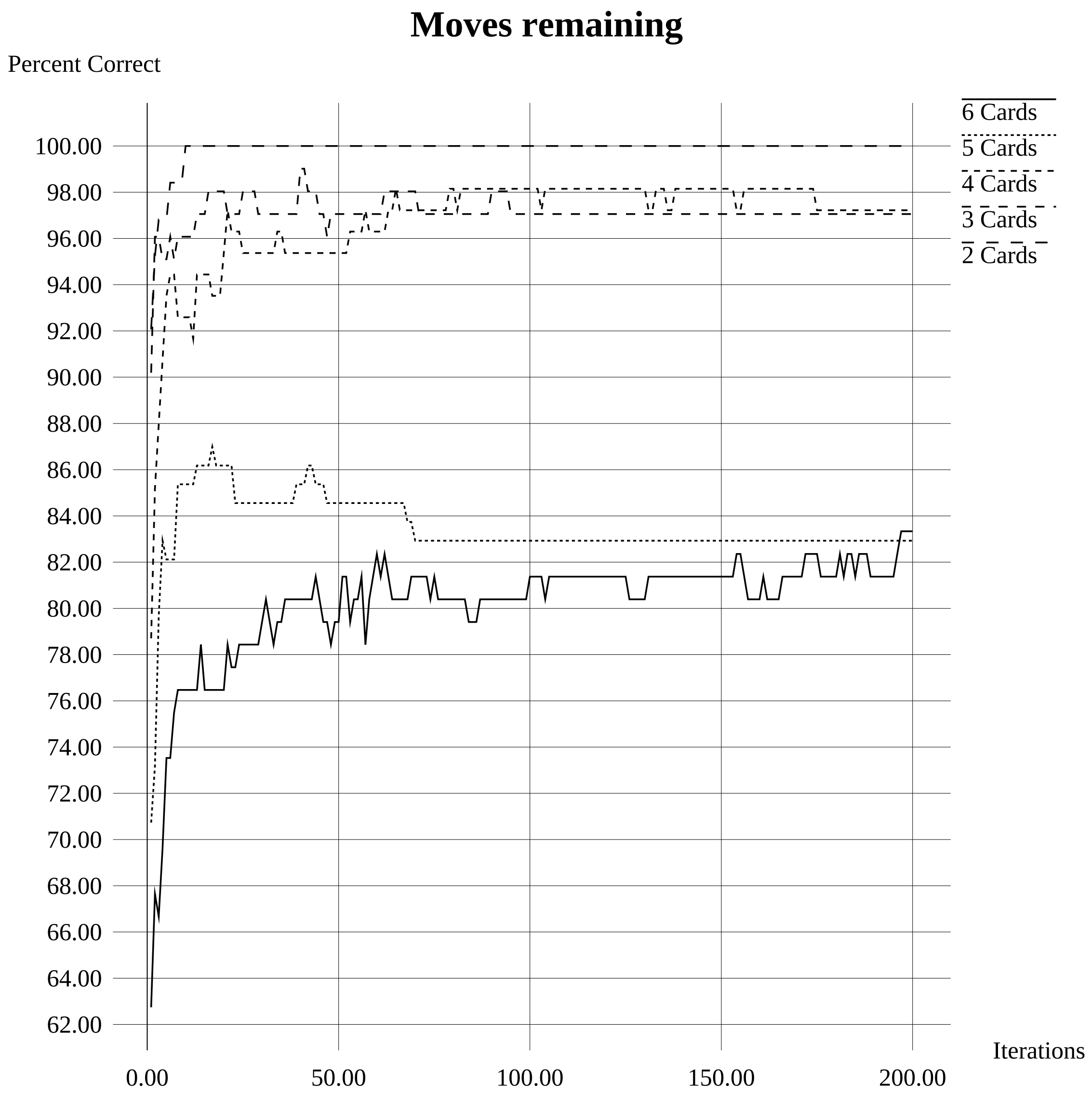

Figure 6.7: Moves remaining over 200 iterations

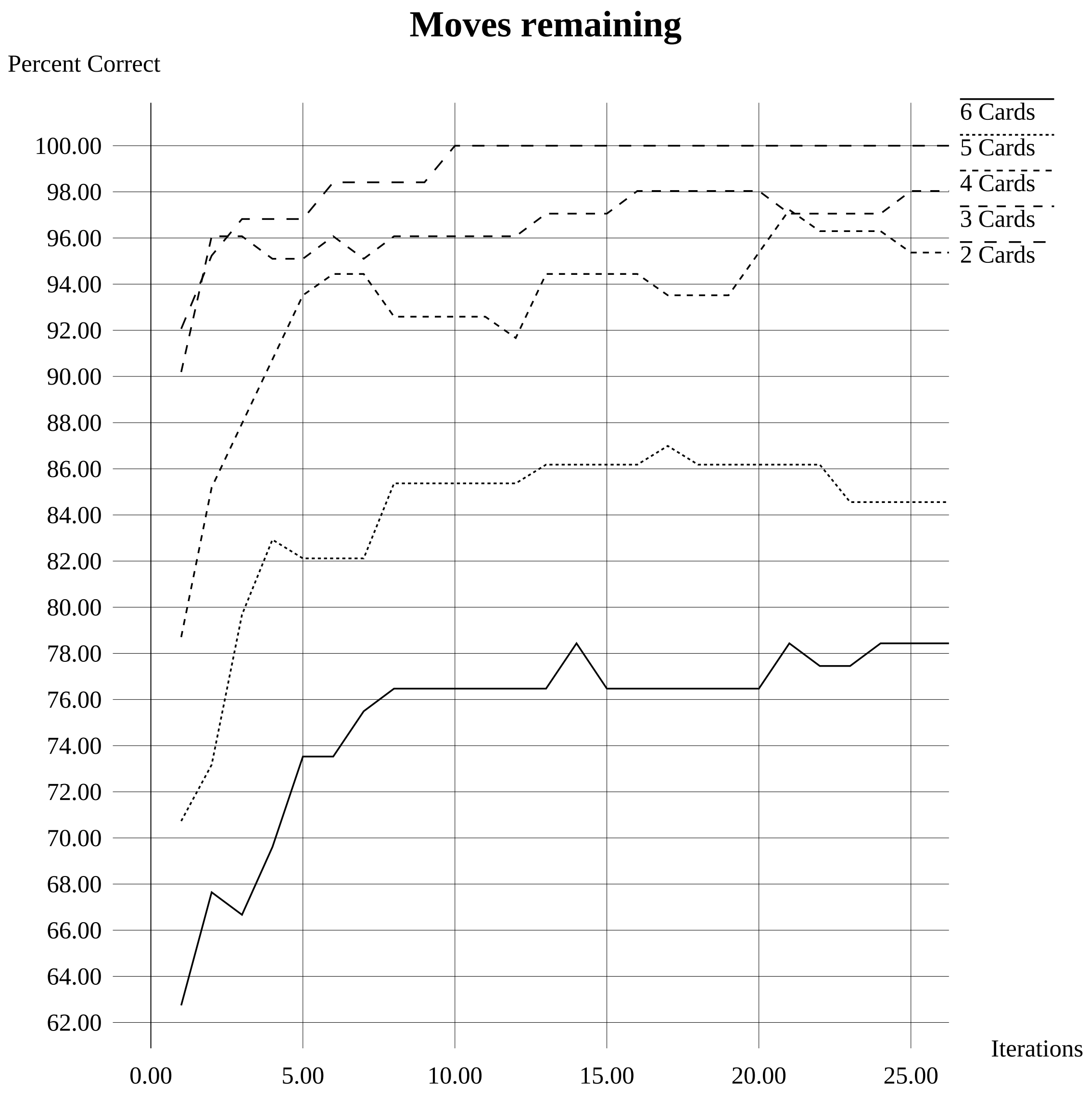

Figure 6.8: Moves remaining over 25 iterations

the curve drops down to 83%. With 4 cards, 5 iterations is the end of the most rapid increase in the curve, reaching just over 93% correct. After 6 iterations the curve levels off and eventually reaches 98%. Given 3 cards, both points of interest occur after 2 iterations, where the curve reaches 96%. For 2 cards, the curve climbs rapidly for 3 iterations to 97%, and levels out at 100% after 10 iterations.

In summary, positions further away from the end of the game usually require more iterations to determine a best choice.

6.3.3 Heuristics for positions

When a strong game specific heuristic exists for a class of problems, you would expect that the algorithm would identify problems for which the heuristic is accurate and would make a choice rapidly. Weaker heuristics require more iterations to make a choice.

This behavior was seen in the experiment to a limited extent. Bridge's "Win or lose as cheaply as possible" for 4th hand decisions and "2nd hand low" are both fairly simple heuristics which are usually correct [David Mutchler, private correspondence]. When there is no reliable heuristic, such as the choice of card to lead, more iterations are required to get significant changes from the heuristic's choice after 1 iteration. Probability-based moves such as finesses[2] can be identified, but require many more iterations to determine their likelihood of success. All these phenomena are demonstrated by the algorithm's performance. There were some surprises, however. You would expect 2nd and 4th hand decisions to be more accurate, and settle on the correct answer quicker than the others. In fact, after 60 iterations, the 1st, 2nd, and 3rd hand curves all lie very close to each other, while the 4th hand one is significantly lower.[3] A second surprise was that the accuracy of 0th hand decisions

[2] See section 6.7 for an example of a finesse.

[3] We believe that 4th hand decisions behave poorly because, as implemented, the heuristic always picked the highest ranked card in ties. There were presumably several occasions when a higher ranked card than the correct one being played earlier than needed has no affect on the outcome of play, given that all the cards are seen.

actually declines significantly after 40 iterations. (See Figure 6.9 and Figure 6.10.)

When choosing a suit (0th hand), the curve starts low and tied with 4th hand, then rises very rapidly for 6 iterations, to 88%. It then levels off around 85.5%. First hand decisions start off the highest, rise rapidly for 7 iterations, and level off there at 93%, eventually reaching 95%. Second hand decisions start off poorly, rise rapidly for 6 iterations to 91%, and level off around 93% after 14 iterations. Third hand decisions are the most unstable ones. They rise rapidly for 7 iterations, reaching 89% accuracy. The curve continues to jump about considerably until 90 iterations, where it levels off near 94%. Fourth hand decisions start off very poorly, climb rapidly for 4 iterations to 84%, and level off at 89% after 14 iterations.

To summarize, game specific heuristics will allow you to reduce the number of iterations needed in a manner related to their accuracy and generality. However, the lack of a heuristic does not necessarily indicate that more iterations are needed.

6.3.4 Strength of the position

Given a very weak or very strong position, you would expect that it would be much easier to determine the best move: for weak positions, because there is little you can do to improve it greatly, and for strong ones because of what the weak opposition can or cannot do.

This was not borne out by the experiments. In order to determine the strength of the position, a simple static evaluator that summed the value of all cards a player had, assigning 13 points for Aces, 12 points for Kings, on down to 1 point for a deuce was used. The total for the player and his partner was then divided by the total for all players.[4] The relative strengths were then divided up into 2 and 4 roughly equally sized groups. The teams strengths run from 36% to 47%, 47% to 50%, 51% to 53%, and 53% to 68%. (See Figures 6.11, 6.12, 6.13 and 6.14.)

[4]This heuristic leaves something to be desired, in that it values an A and 2 in a suit differently when that card is the only one remaining in that suit.

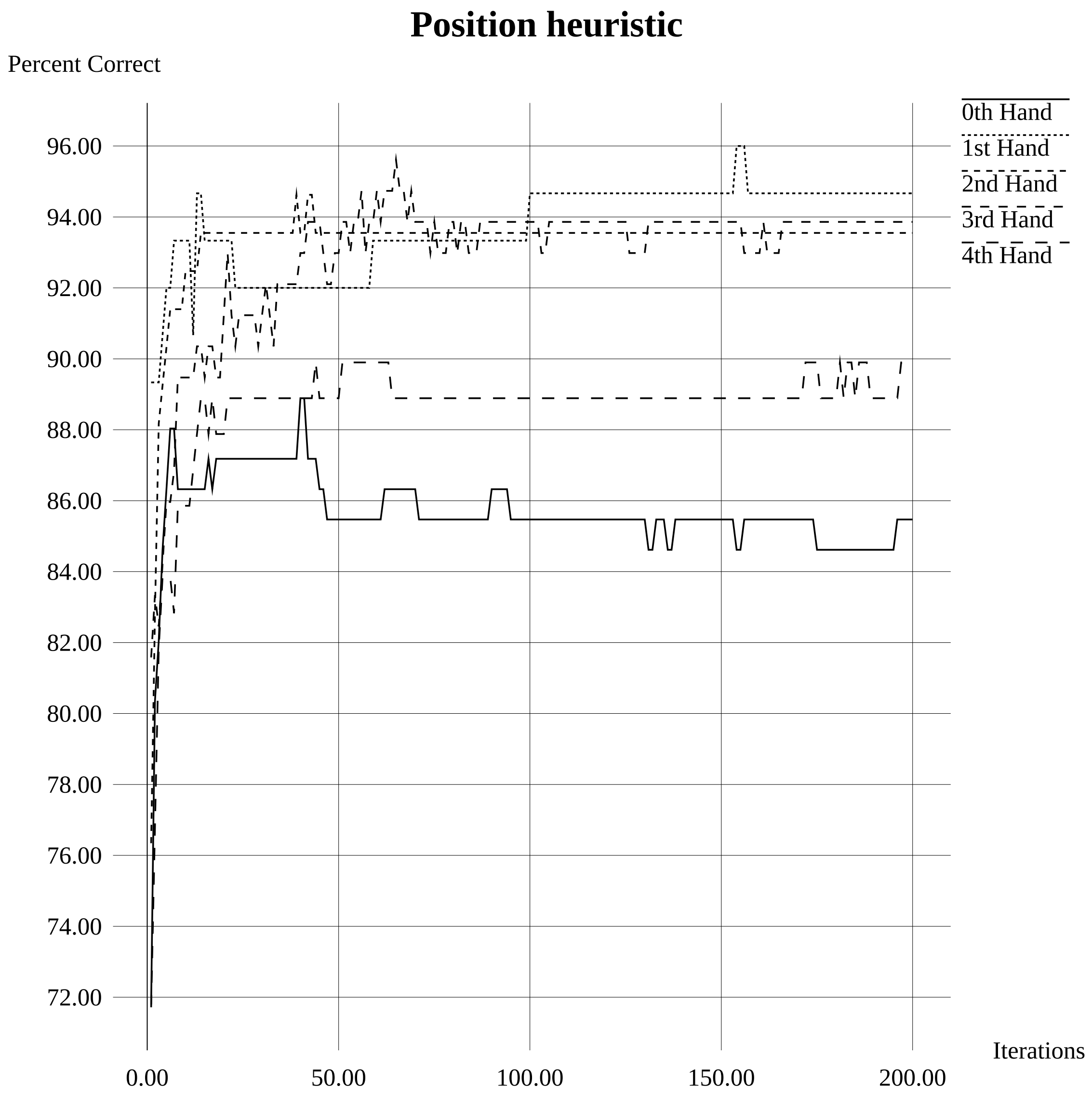

Figure 6.9: Position specific results over 200 iterations

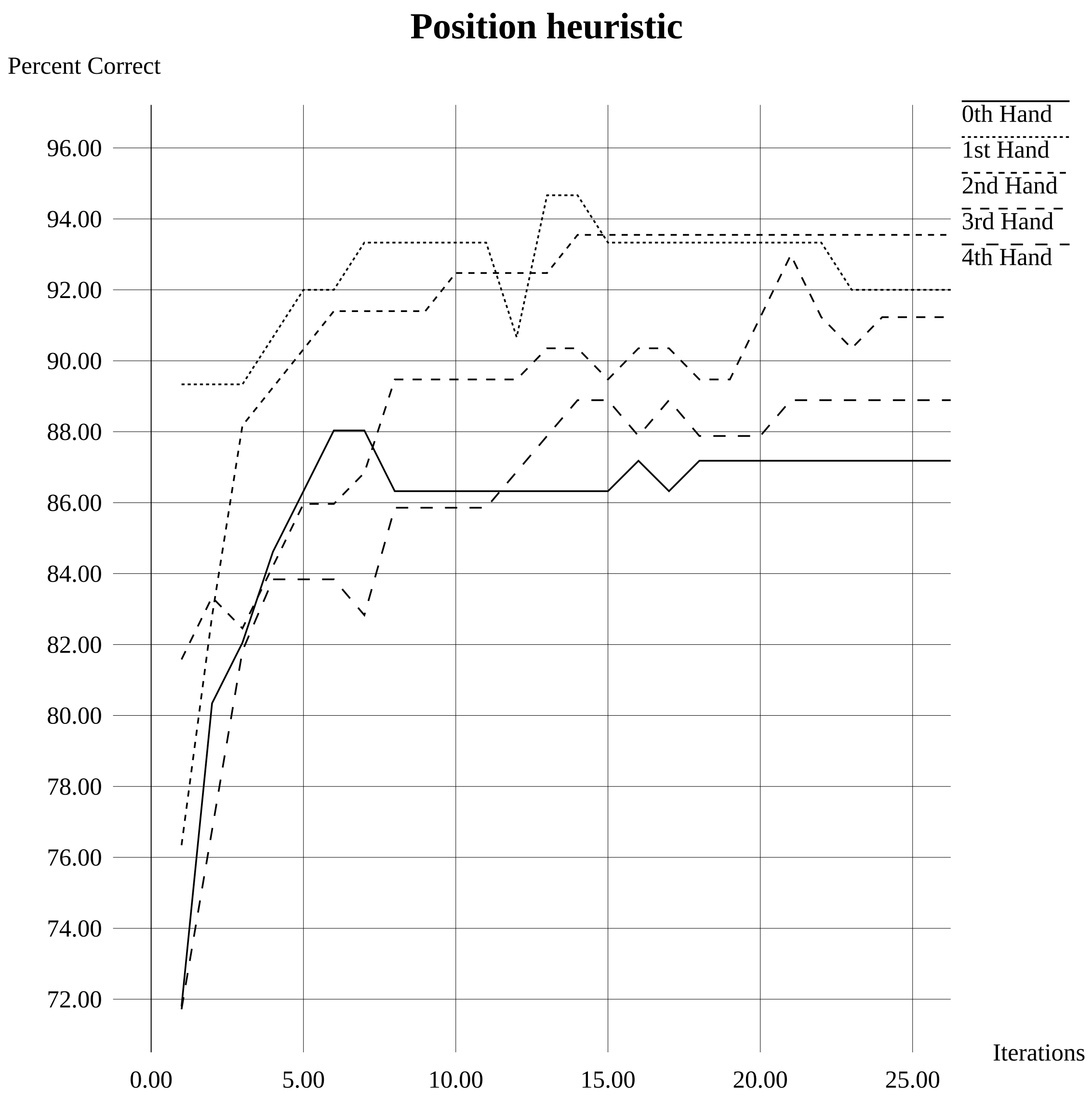

Figure 6.10: Position specific results over 25 iterations

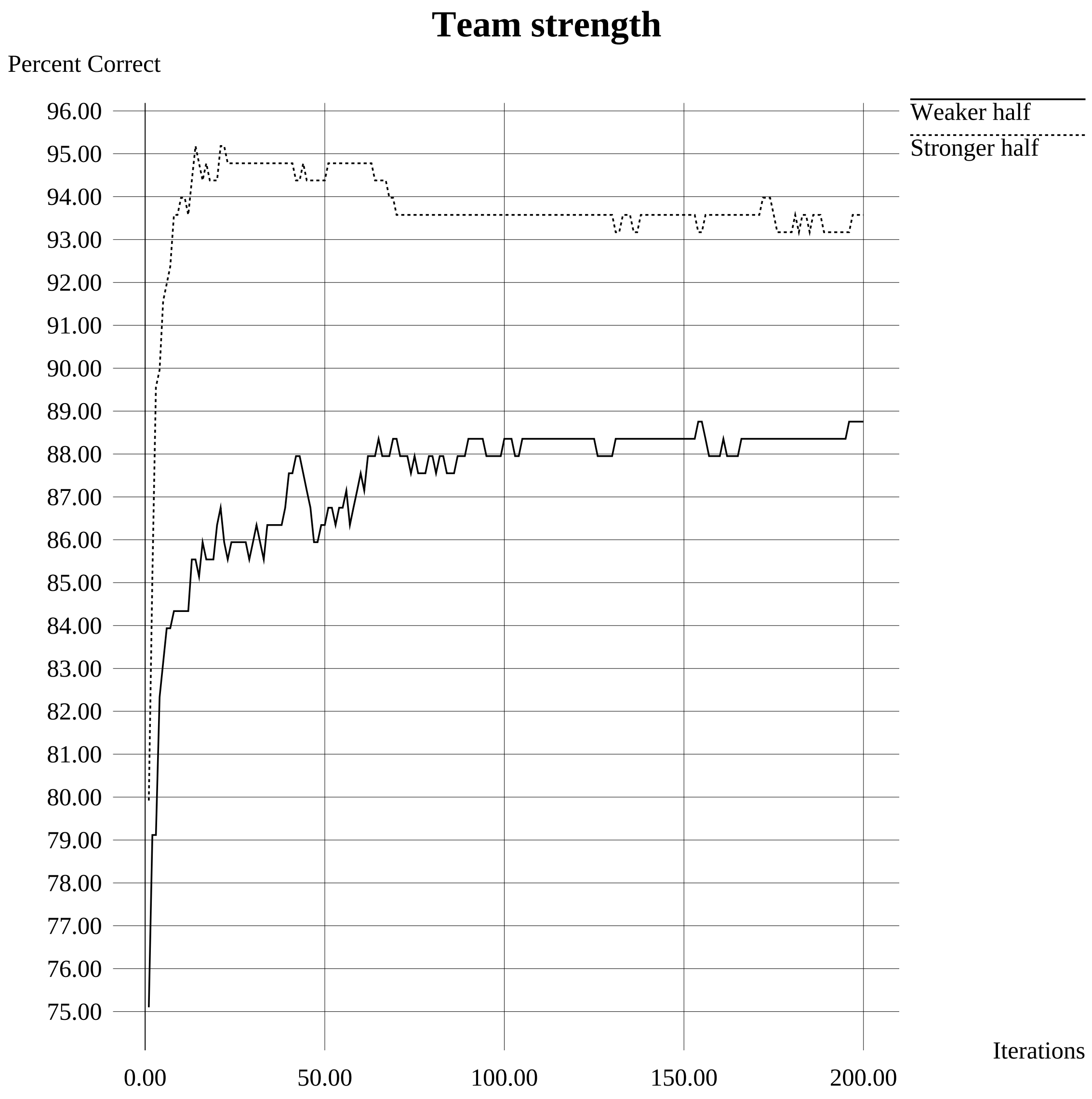

Figure 6.11: Strength of the position over 200 iterations, 2 groups

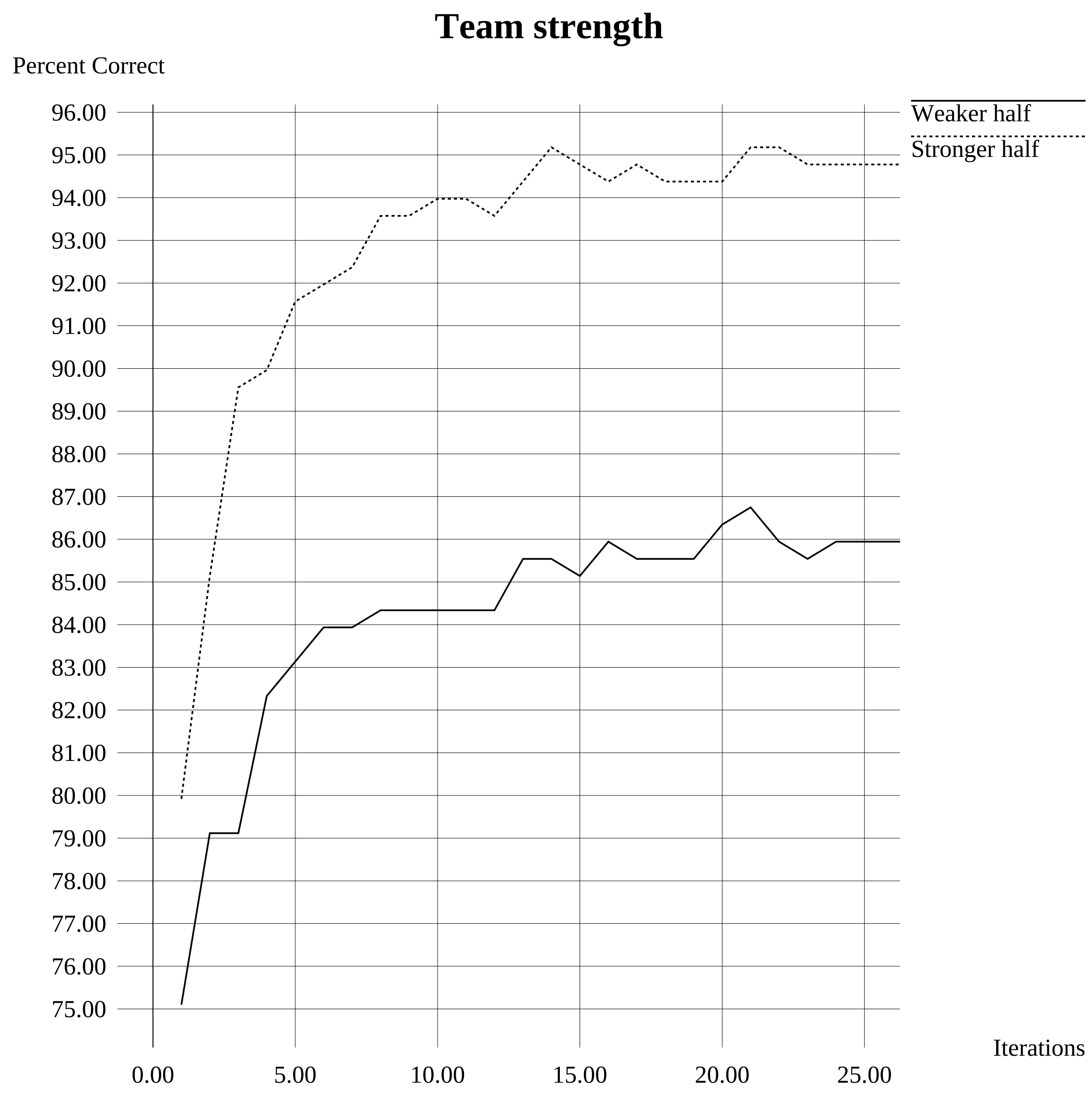

Figure 6.12: Strength of the position over 25 iterations, 2 groups

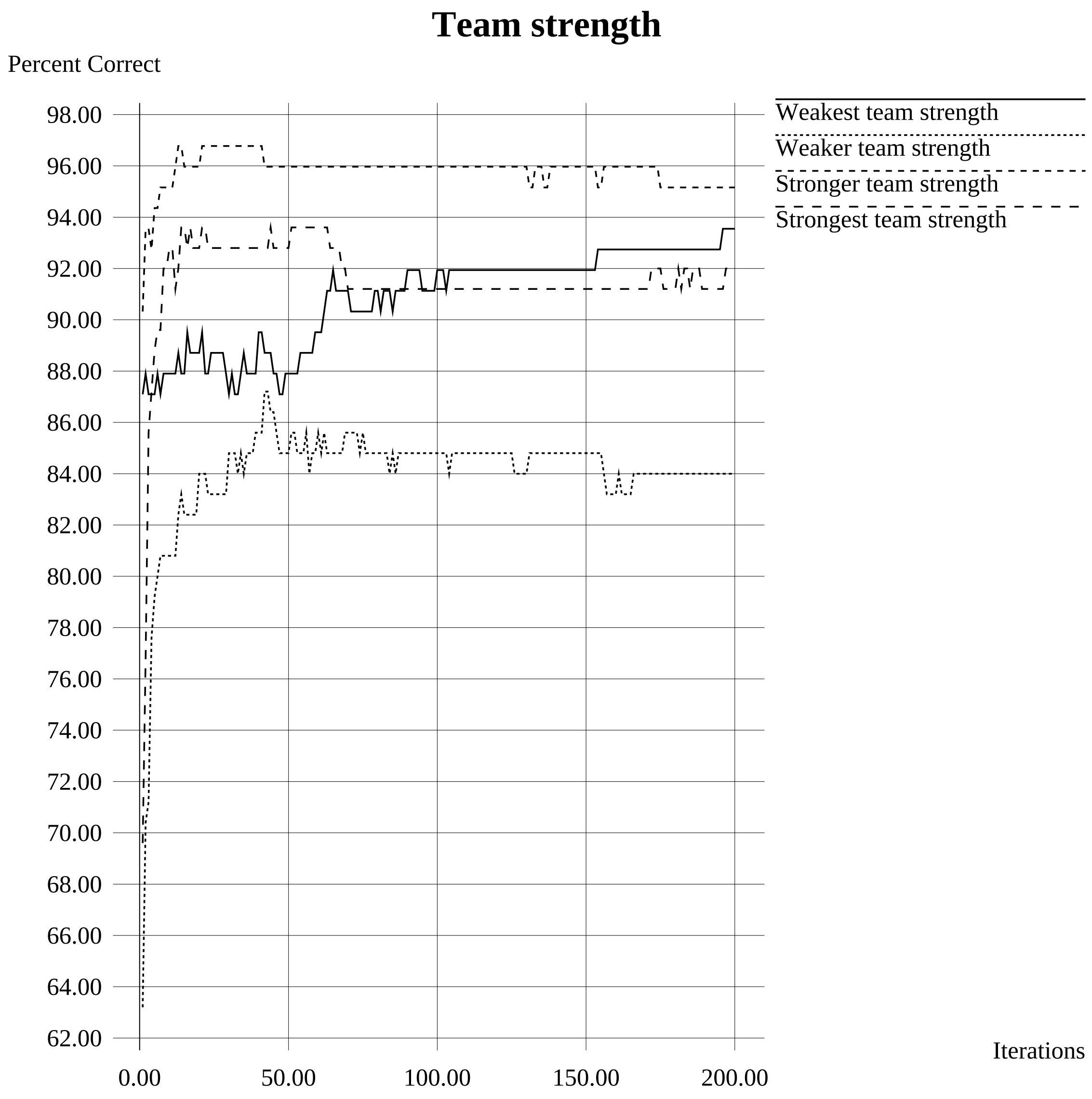

Figure 6.13: Strength of the position over 200 iterations, 4 groups

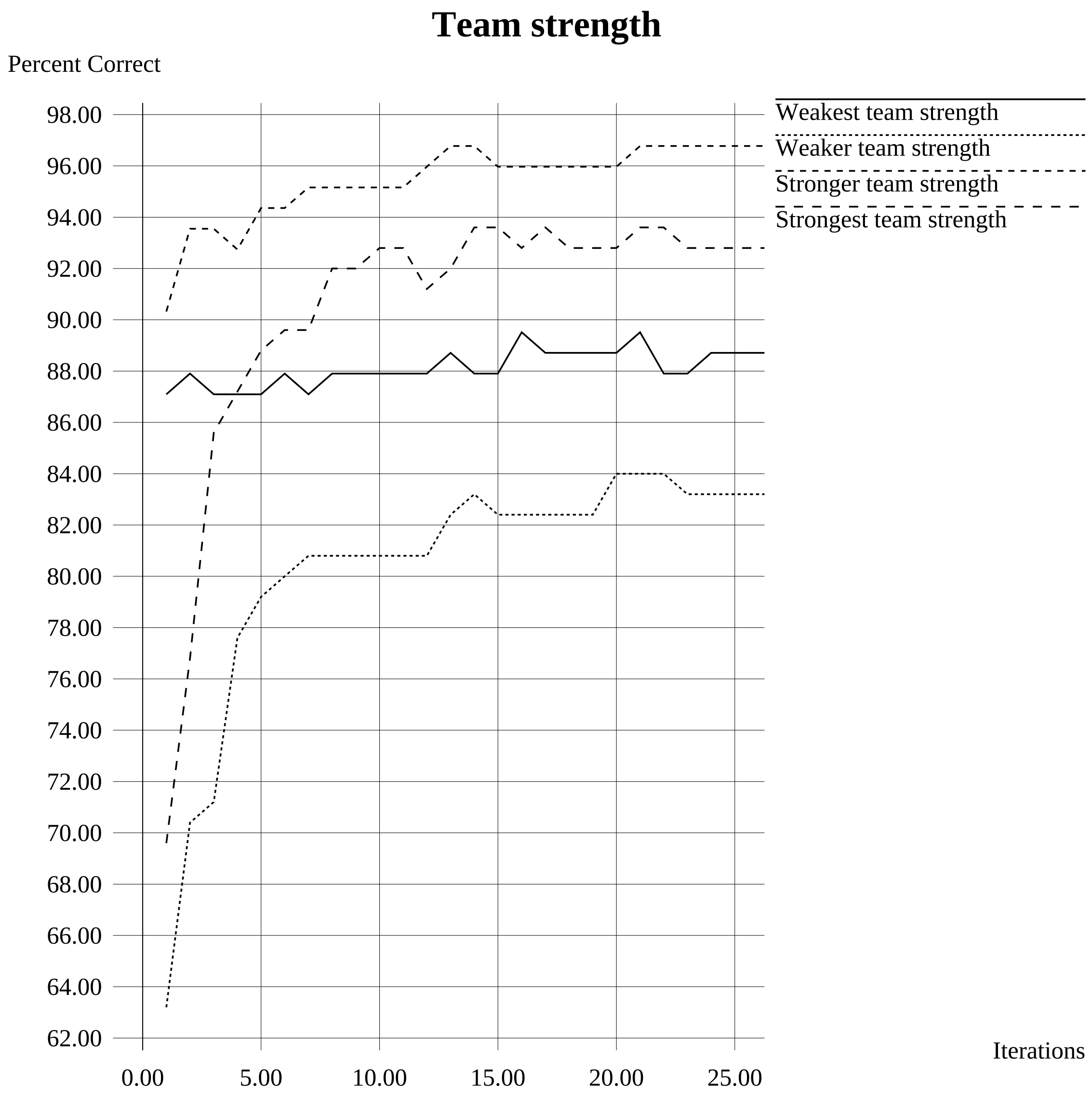

Figure 6.14: Strength of the position over 25 iterations, 4 groups

Both groupings showed a tendency towards stronger hands being easier to settle on a correct choice. When broken into 4 groups, some interesting events took place. The strongest and weakest problems stayed between the two middle groups. The strongest positions started out poorly, had their curve rise rapidly for 8 iterations around 92%, and flatten out at 93% after 10 iterations, and then eventually decline to 91%. The weaker than average team strength positions had similar characteristics— it started out very low, rises rapidly for 7 iterations, and flattens out around 85% after 30 iterations. Stronger than average hands started out very high, rise rapidly for 2 iterations to 93%, and level off around 96% after 12 iterations. The weakest hands start out between 87% and 88%, and stay close to there for 60 iterations. Past 60 iterations, the curve stays around 92%.

In general, it seems that very strong positions make great gains when given a few iterations, while weak hands start out the best, and gain very little thereafter, with most of the gain coming after a large number of iterations. Positions with a slight advantage make moderate gains after a very high starting accuracy, while ones where the team is at a slight disadvantage start out poorly and make rapid gains.

6.3.5 Strength of visible positions

When the visible information for the various players indicates significantly weaker or stronger positions than the unknown positions could have, you would expect that it would be easier to determine the correct move.

When broken down into 4 roughly equally sized categories, the above conjecture was observed. As with position strength (section 6.3.4), the strengths were calculated by assigning 1 to 13 points for Deuces to Aces respectively. The strengths of the two hands that the player could see were compared to the total strength for all 4 hands. The visible strength runs from 25% to 45.9% for the first group, 45.9% to 50% for the second, 50% to 54% for the third, and over 54% for the last group. (See Figure 6.15 and Figure 6.16.)

As can be seen from the graphs, positions where the visible position strengths are either very

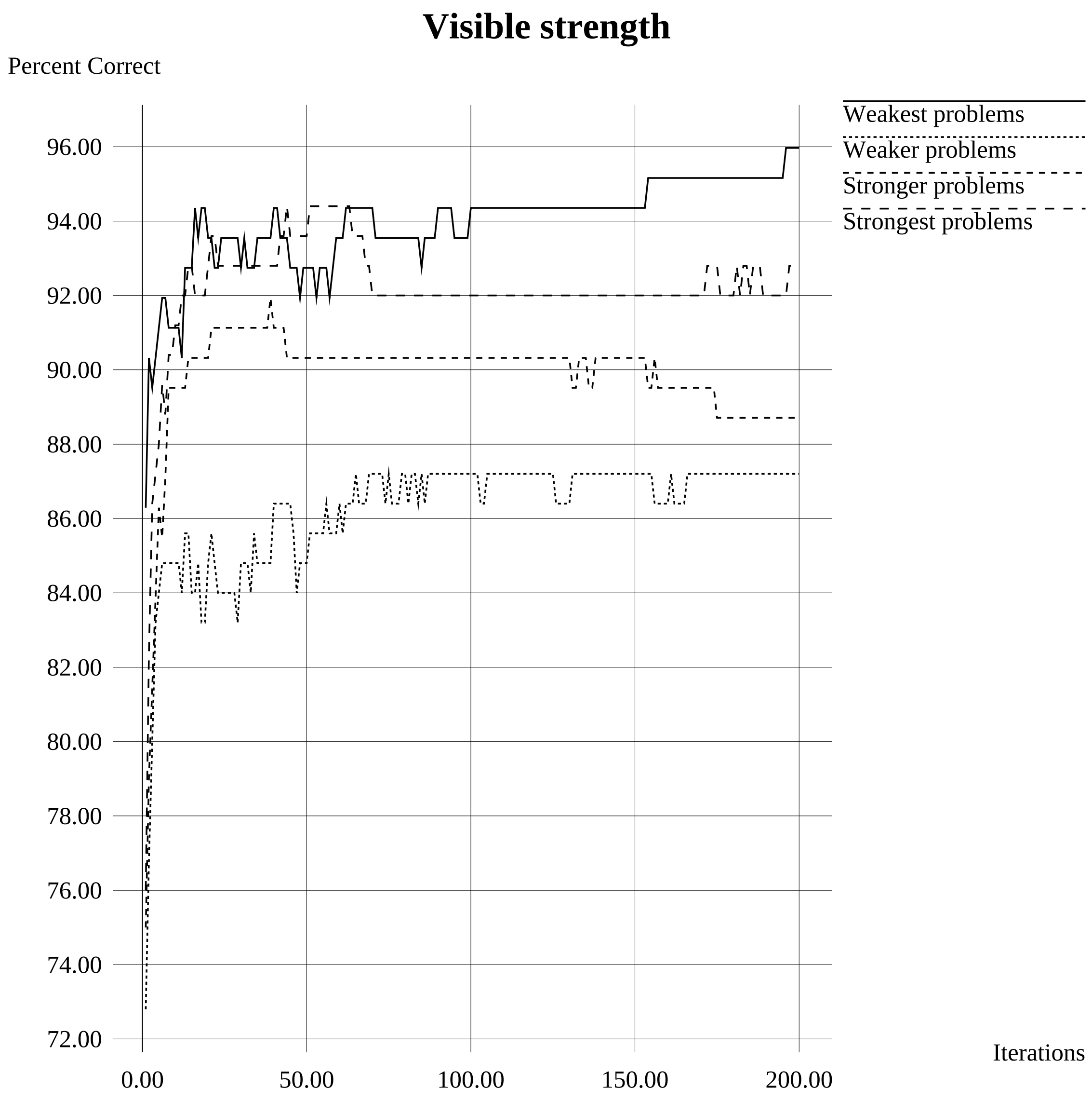

Figure 6.15: Visible strength results over 200 iterations

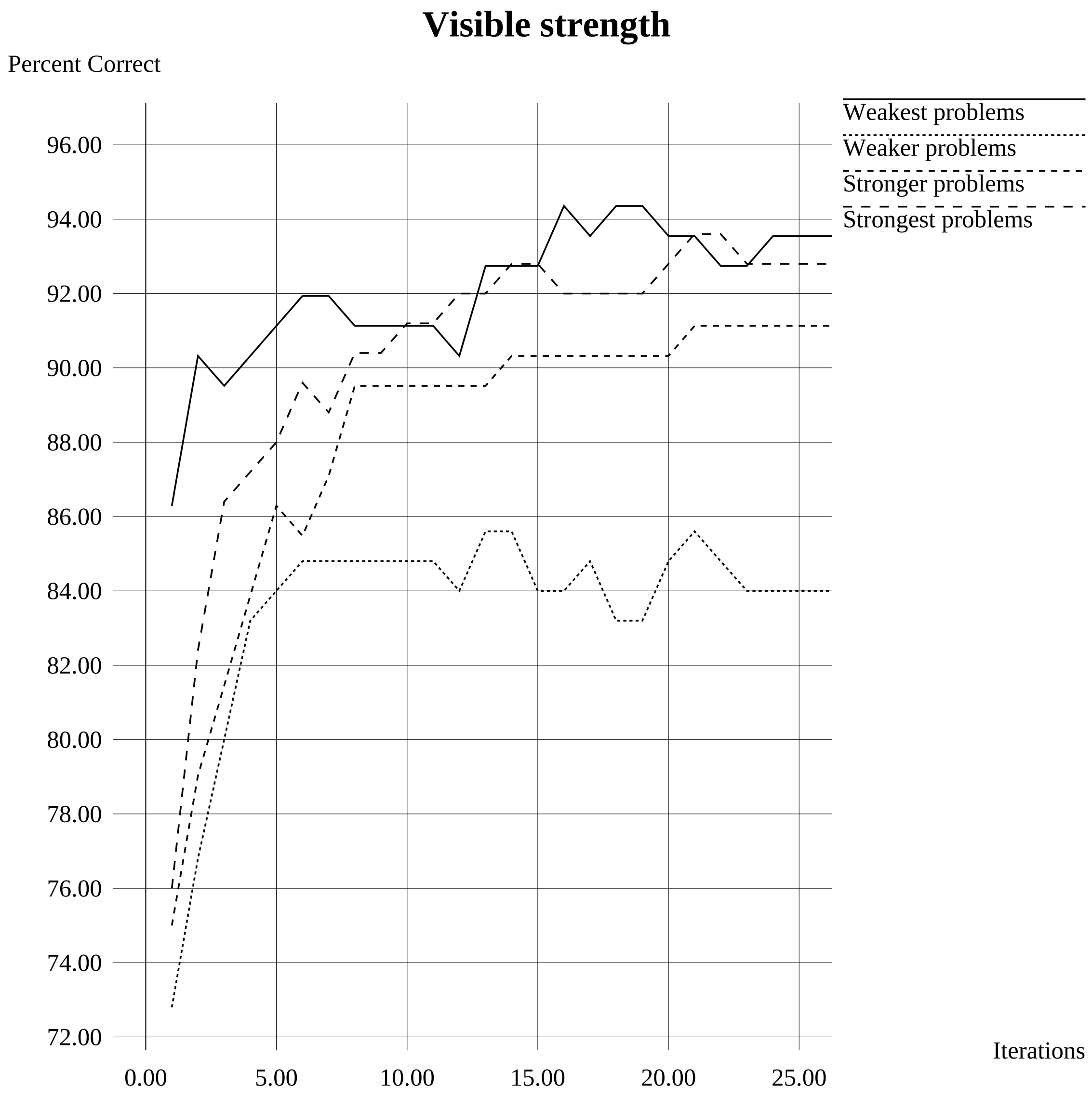

Figure 6.16: Visible strength results over 25 iterations

strong or very weak, consistently outperform ones where the visible versus not visible strengths are close to even. In particular, when the visible strength is lowest, the graph shows a high starting accuracy, followed by a rapid rise for 2 iterations, followed by a leveling off around 93% after 13 iterations. The curve does eventually reach 96%. When there is a very strong position showing, the curve starts out poorly, but rapidly rises to 89% after 6 iterations. At 14 iterations, the curve levels out at 92%. Given a strong position, the curve again starts out low, and rises rapidly for 8 iterations to 89%. The curve levels out at just over 90% after 14 iterations, and eventually declines to 89%. The heuristic performs the worst on weak positions, where it starts low, rises rapidly for 6 iterations and levels out at 85%. (A second plateau occurs after 70 iterations around 87%.)

In conclusion, when the strength of the visible to not visible positions is fairly close, more iterations are typically needed. Given greatly imbalanced ones, fewer iterations will get you higher accuracies.

6.3.6 CPU time

Problems that are more expensive to compute require more iterations to settle on a best choice.

While this trend is supported by the data, several caveats apply. Since cards are distributed randomly in the two unknown hands, the time that an iteration takes can vary widely— for example, if player A gets a few cards (or better yet, a single card) in the suit lead, then the size of the tree to search is considerably smaller than a deal in which A gets none and all cards in his hand need to be considered as discards. Noise due to garbage collection further distorts the time that iterations took. (See Figures 6.17, 6.18, and 6.19.) It should be noted that we chose to look at this factor as if it were static, since you can predict the top level branching factor, number of alternatives and cards which have the biggest influence on it. You could just as easily have considered it a dynamic one.

The quarter of problems that took the least time rise quickly for 3 iterations, and level out at 99% after 10 iterations. The faster than average group of problems rise rapidly for 6 iterations, and

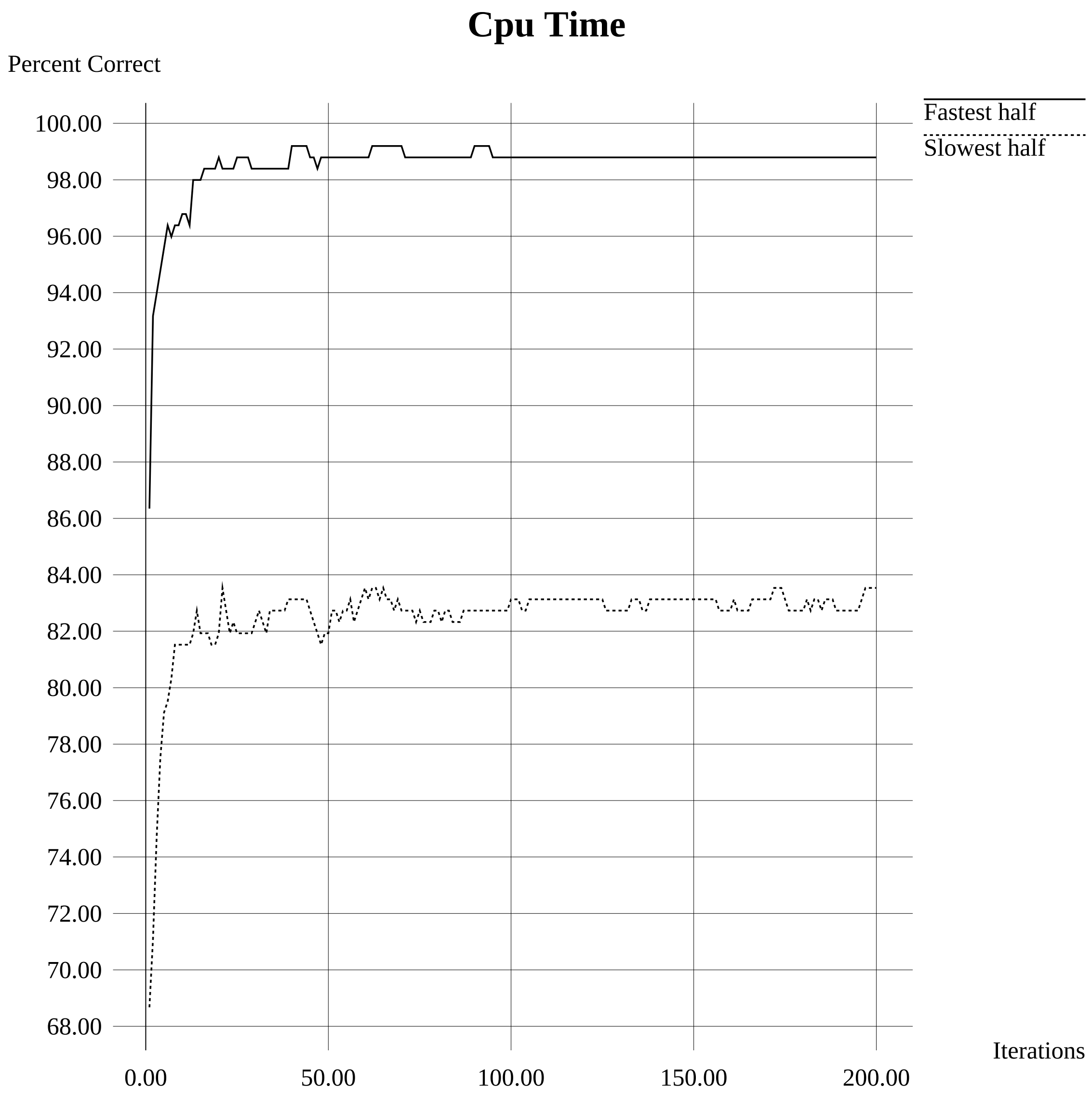

Figure 6.17: CPU time results over 200 iterations in 2 groups

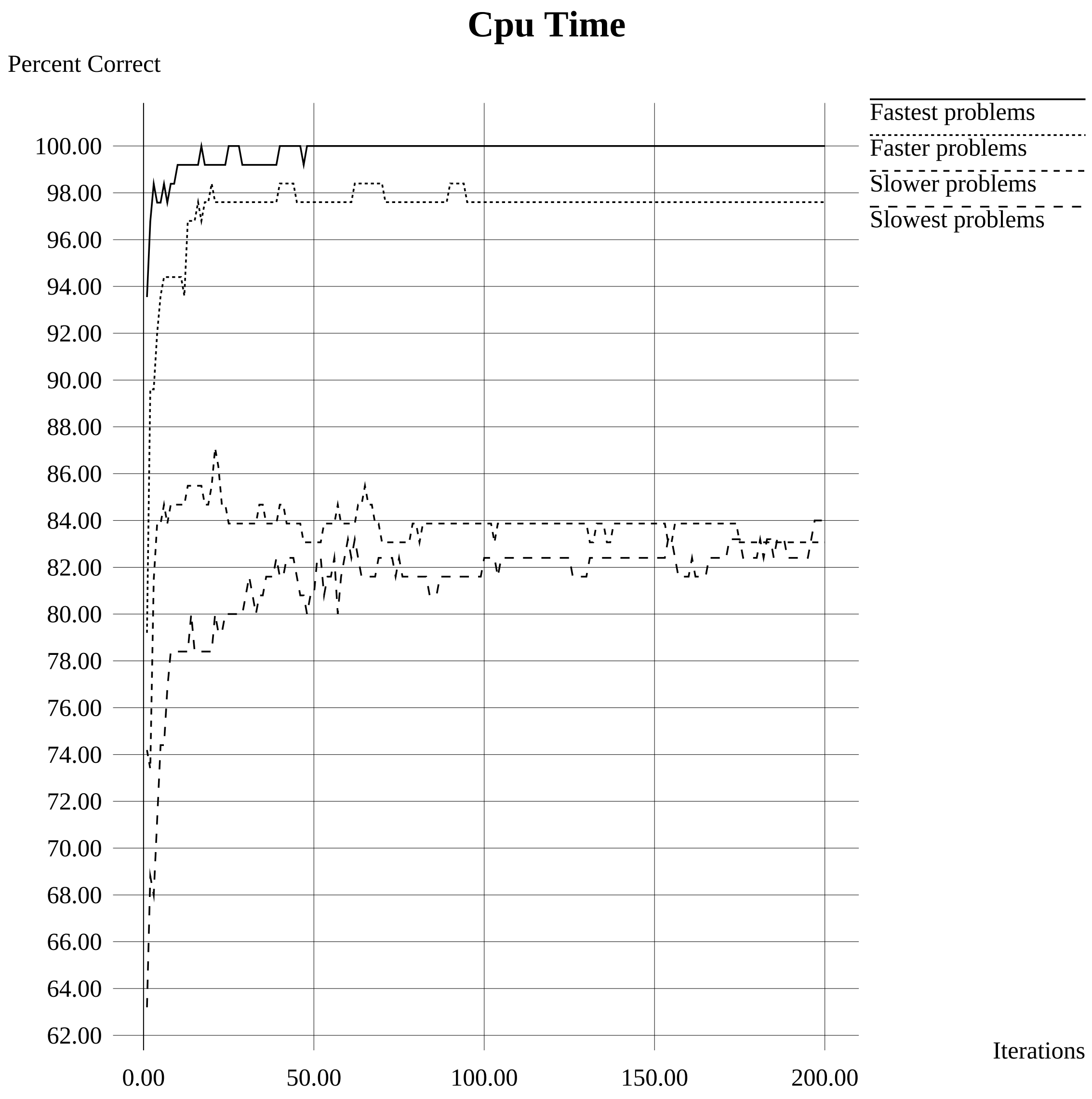

Figure 6.18: CPU time results over 200 iterations

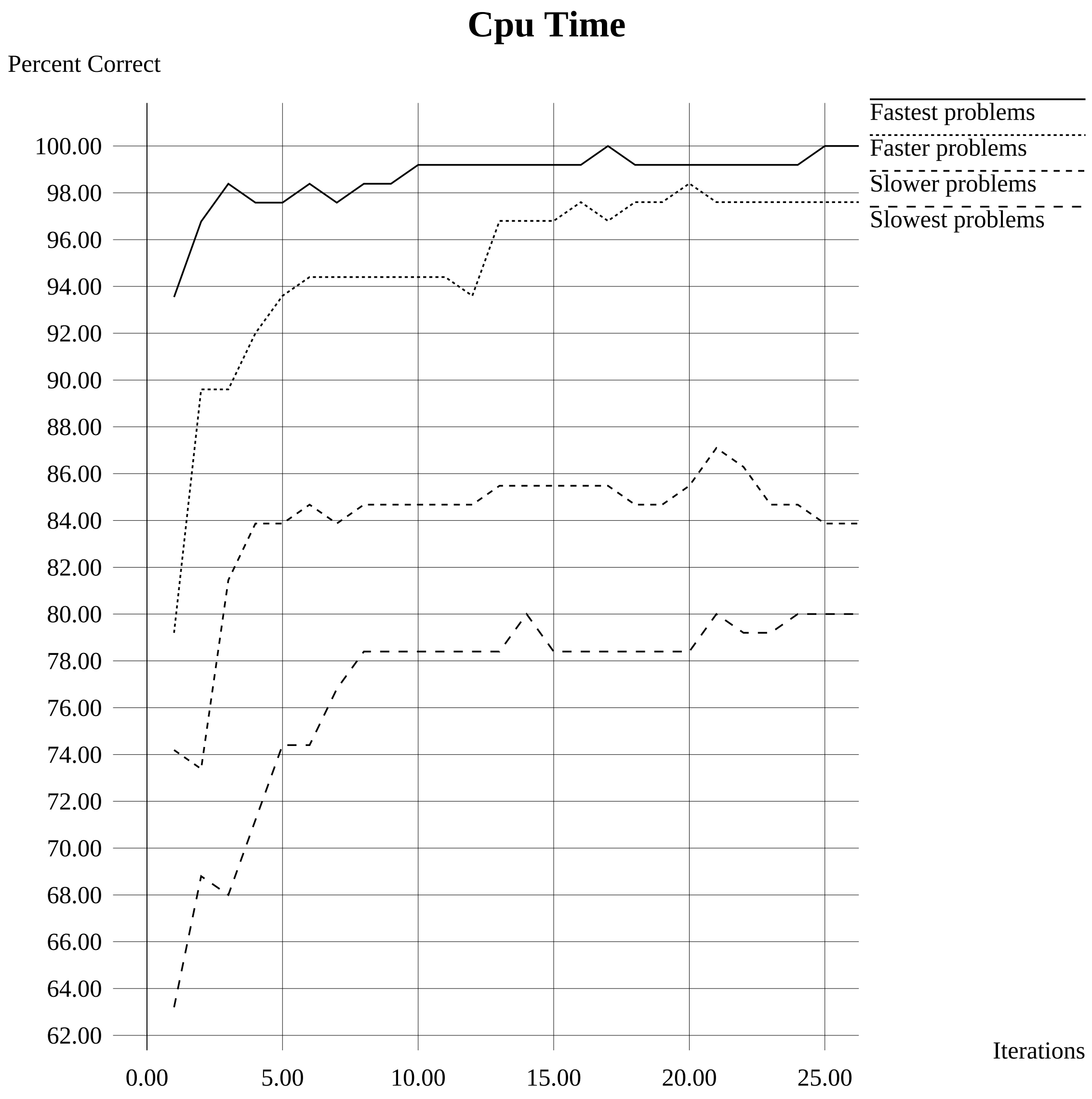

Figure 6.19: CPU time results over 25 iterations

levels out at 97% after 13 iterations. The slower than average problems rise rapidly for 4 iterations to 84% and level out there. The slowest problems rise rapidly for 8 iterations, and level out there at 78%.

In summary, the longer a problem takes to compute a single iteration, the more iterations needed. Several other factor's influence may cause sufficient noise in the run time to make extremely accurate calculations impossible. Taking an average of several runs may be sufficient to reduce the noise, at the cost of complicating the code.

6.3.7 Number of alternatives

When faced with more alternatives to choose from, you would expect that more iterations would be needed.

This expectation was not conclusively borne out however. (See Figures 6.20 and 6.21.) While the 2 and 6 alternative sets were the easiest and hardest to solve as expected, the 3 and 4 alternative problems did not behave as anticipated.

With 2 alternatives, the curve rises rapidly for 3 iterations, and levels out above 96% after 7 iterations. Given 3 alternatives, the curve climbs steeply for 5 iterations, and levels off around 81%after 16 iterations; When given 4 alternatives, the curve rises rapidly for 6 iterations, and levels out at 100% after 75 iterations. The 5 and 6 alternatives hands' sample sizes are too small for any good advice.

In summary, problems with more alternatives do not automatically need more iterations to determine a choice. Other factors appear to overwhelm any influence this may have on the number of iterations that a problem should receive to be correctly solved.

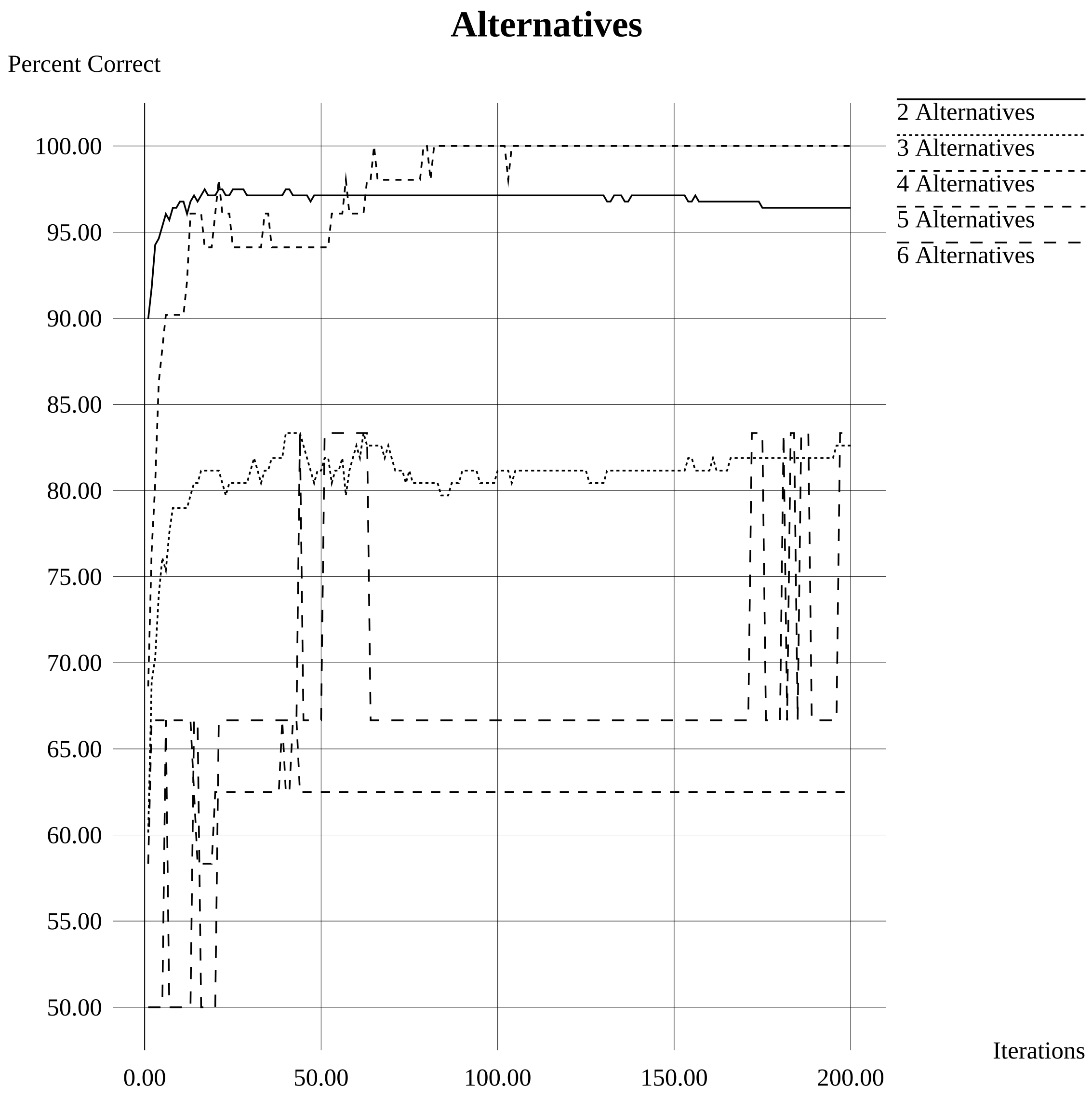

Figure 6.20: Effects of the number of alternatives over 200 iterations

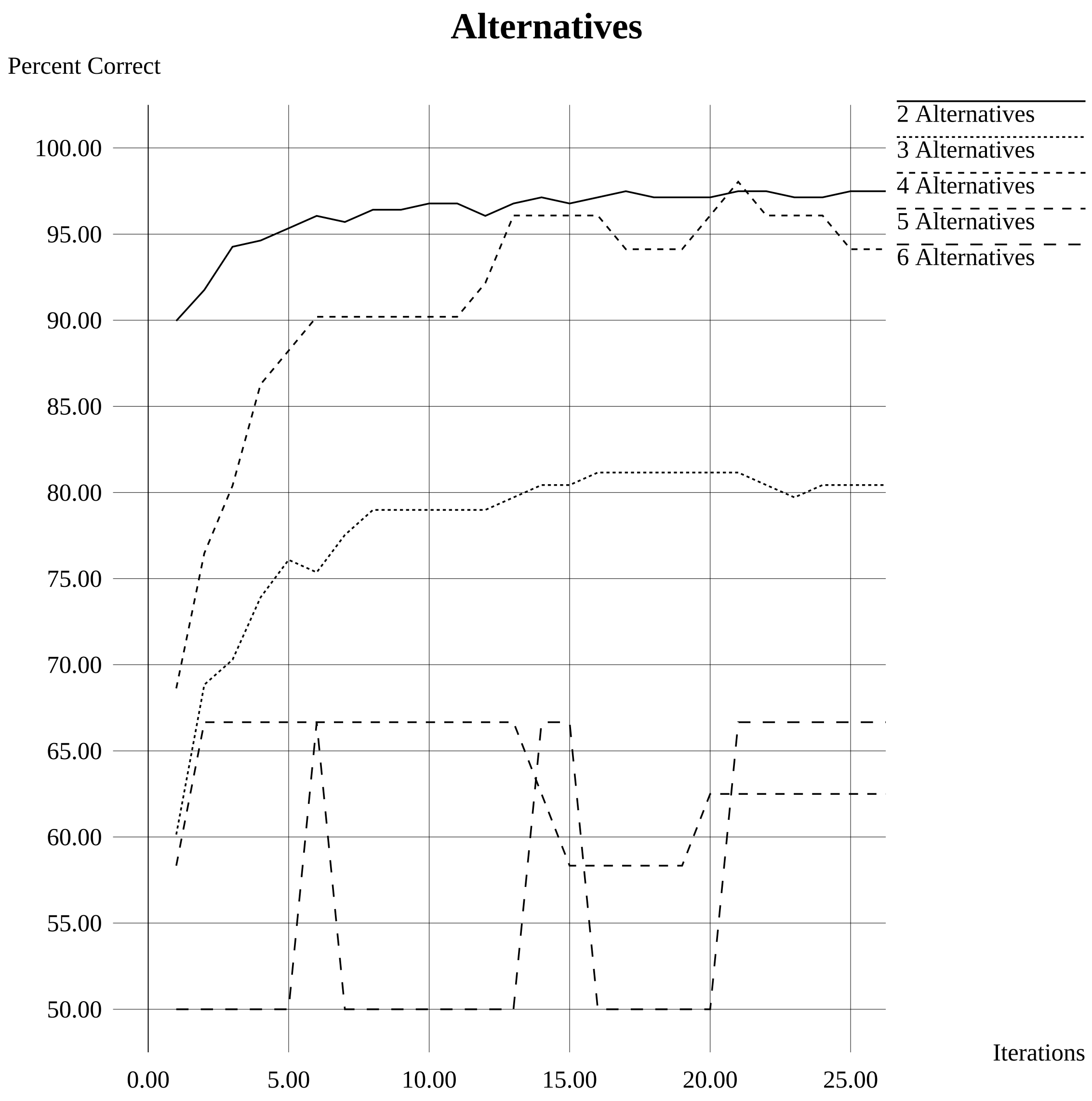

Figure 6.21: Effects of the number of alternatives over 25 iterations

6.3.8 Discard versus non-discard moves

With the greater number of alternatives, and less obvious effect on the game's play, you would expect that discard positions would be more difficult to determine a correct choice for.

This conclusion is not universally true, however. Initially, discards score lower than non-discards; however after 50 iterations they score better than non-discards. (See Figures 6.22 and 6.23.)

Non-discard positions rise rapidly for the first 5 iterations, and then level off at 90% for a while at 8 iterations. Discard positions rise rapidly for 12 iterations, where they level off, eventually reaching 92%.

There are several probable causes for the discards being easier over the long run to solve. First, there are more larger hands in the non-discard hand set, than in the discard hand set. These problems have been shown in section 6.3.2 to be more difficult to solve. A second reason is because discard positions have no first hand decisions, which are more difficult to solve in section 6.3.3; while also having more 2nd and 4th hand decisions, which are the easiest to solve.

In summary, for games which have moves analogous to bridge's discard, the number of iterations needed should be lower than for non-discard positions, except when you are trying to get the last few percent of the problems correct, when discards are slightly easier to solve.

6.3.9 Player

You would hope that there is no bias in bridge which makes any of the 4 player positions (N, S, E, W) any easier to play.

When broken down by player, there is some indication that there is a bias. As can be seen from Figures 6.24 and 6.25, with the exception of North, all players have similar points where the curve flattens out. However, North's curve is considerably higher than the others, and West's curve is noticeably lower.

Further investigation is probably needed to determine if this is a result of other factors, or if it

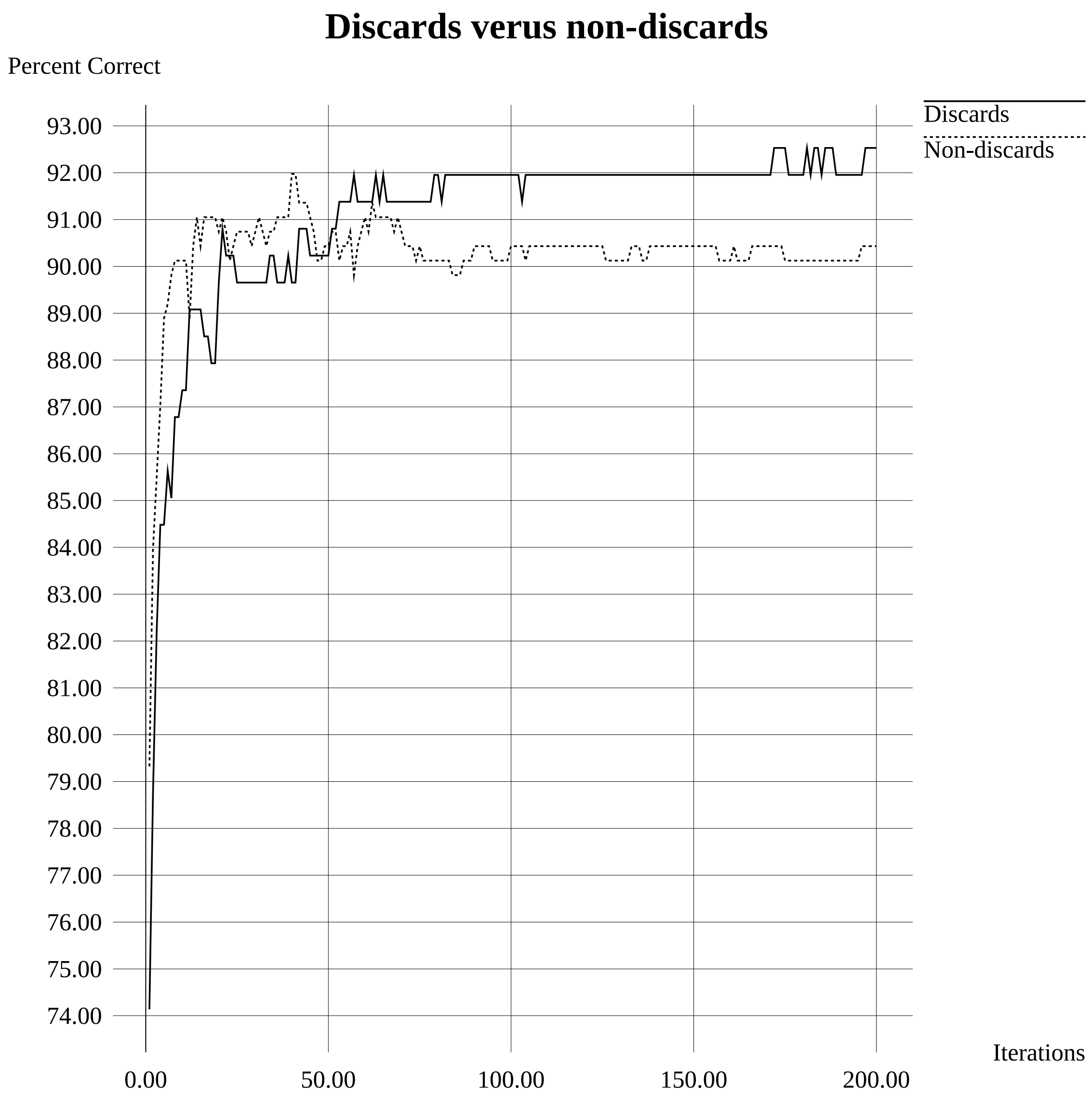

Figure 6.22: Discard versus Non-discard results over 200 iterations

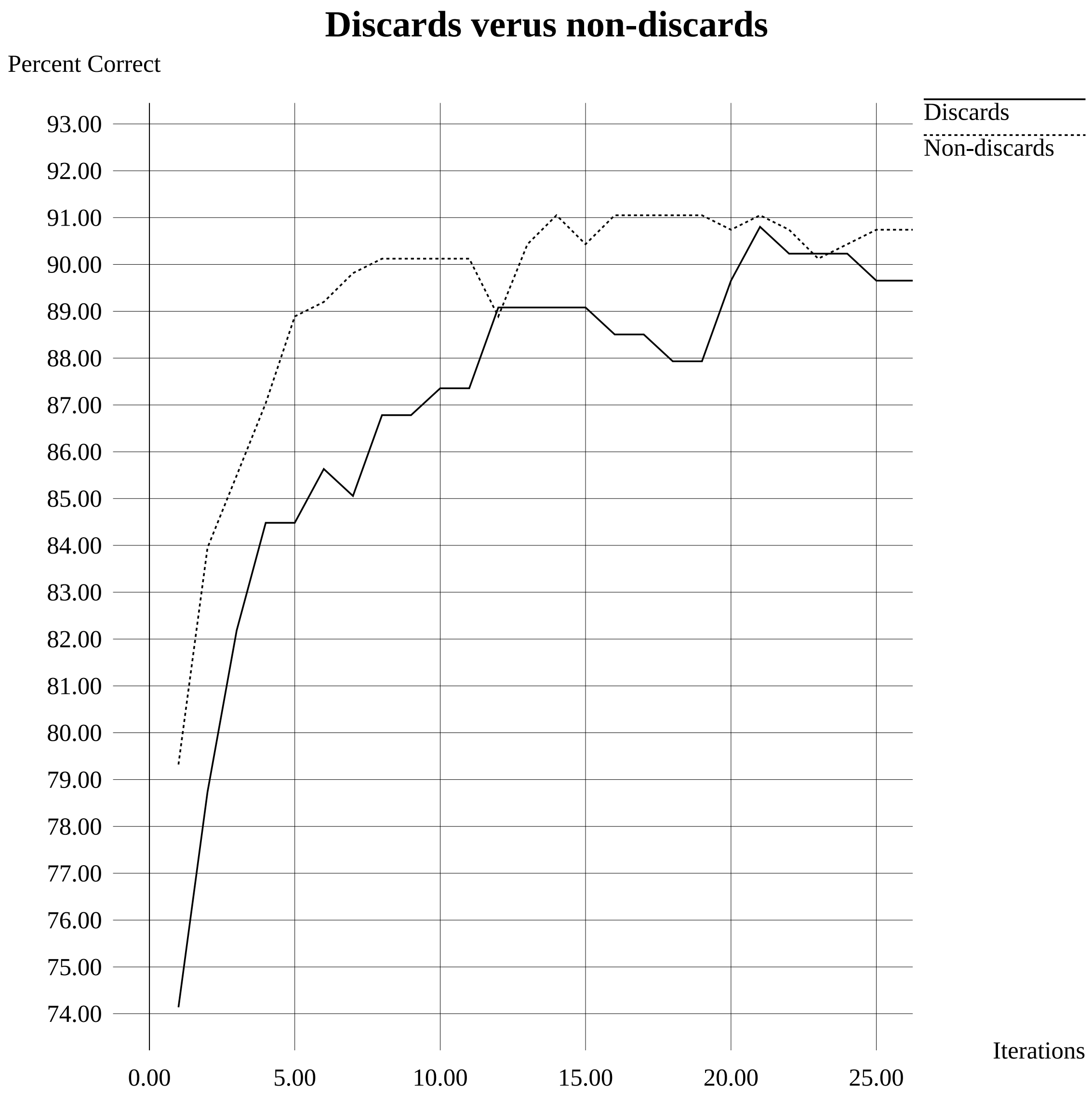

Figure 6.23: Discard versus Non-discard results over 25 iterations

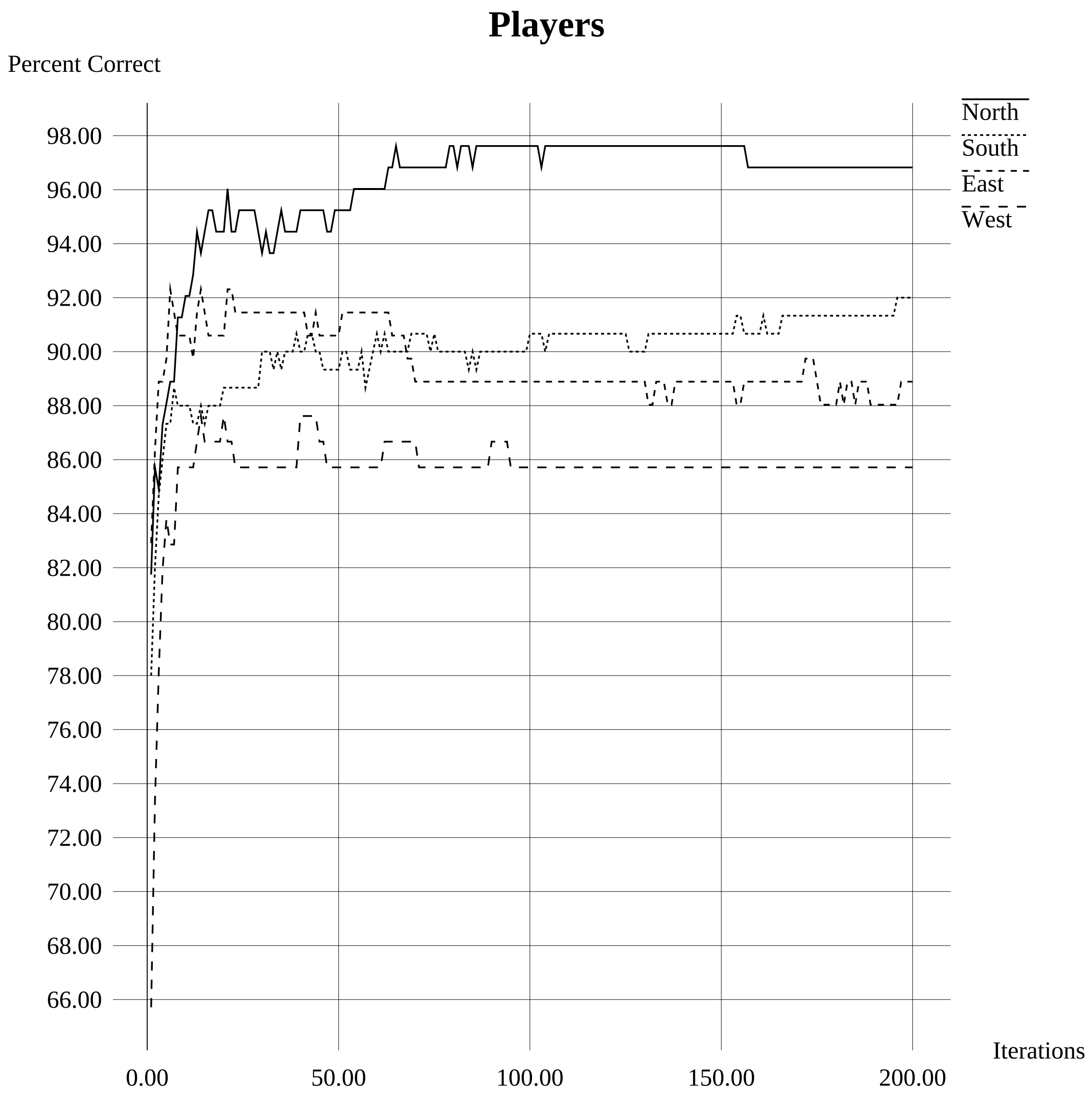

Figure 6.24: Player results over 200 iterations

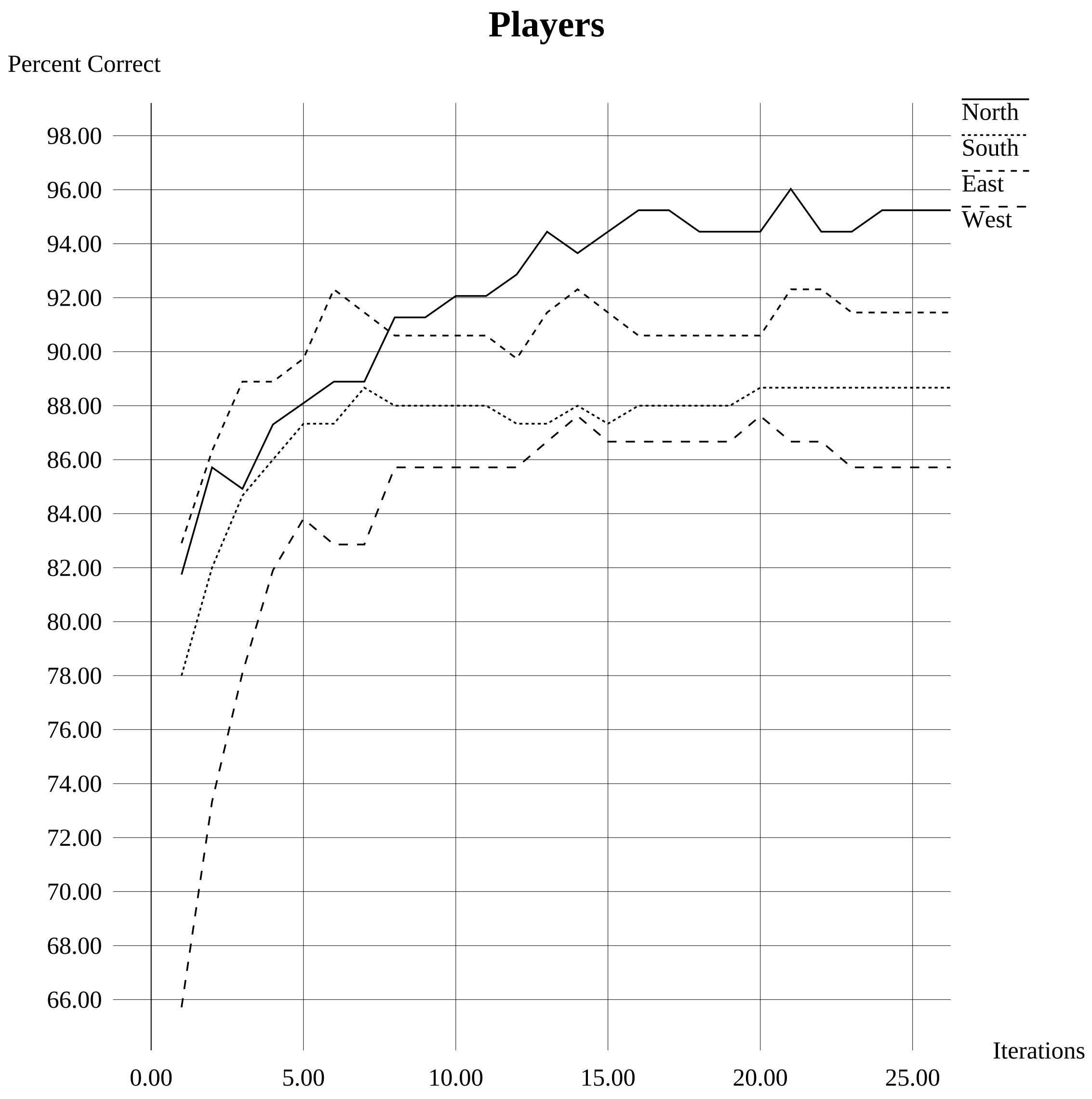

Figure 6.25: Player results over 25 iterations

is really easier to play North's hands, and harder to play West's.

6.4 Dynamic factors

For each type of dynamic factor, we will describe the factor, state our conjecture about its influence, and then examine the factor's actual influence. Dynamic factors are somewhat more difficult than static ones to graph. For each factor we identify, we graph two things. First, we graphed the accuracy against different values of that factor. Second, we graphed the average number of iterations needed versus that factor. As with the static factors, we visually identified the point where the curve stops making rapid gains, and the point where it levels out.

For example, in Figure 6.27, all the curves rise rapidly for 6 iterations, and flatten out at 15.

6.4.1 Quiescence

Quiescence is a measure of how many consecutive iterations a given choice is the one chosen by a heuristic or algorithm. You often search until some quiescence value q, or until an overall cutoff S is reached. For this section, we sample all values of q between 1 and 200, and picked S values of 5, 10, 25, 50, 100, and 200.

> You would expect that the heuristic will, with a small q settle on a correct choice after a small number of iterations, and stick with that choice thereafter.

This behavior was seen quite strongly in the results. As can be seen from Figures 6.26, 6.27, and 6.28 there is a strong correlation between quiescence and accuracy. As can also be seen, for small values of q the choice of S makes little difference.

The curves all rise rapidly for values of q up to 6. Once q reaches 15, there is little if any additional gain. (If we are searching for a q of 15, this means we will keep doing additional iterations until we have gotten the same choice 15 times, or we have reached our overall cutoff, S.)

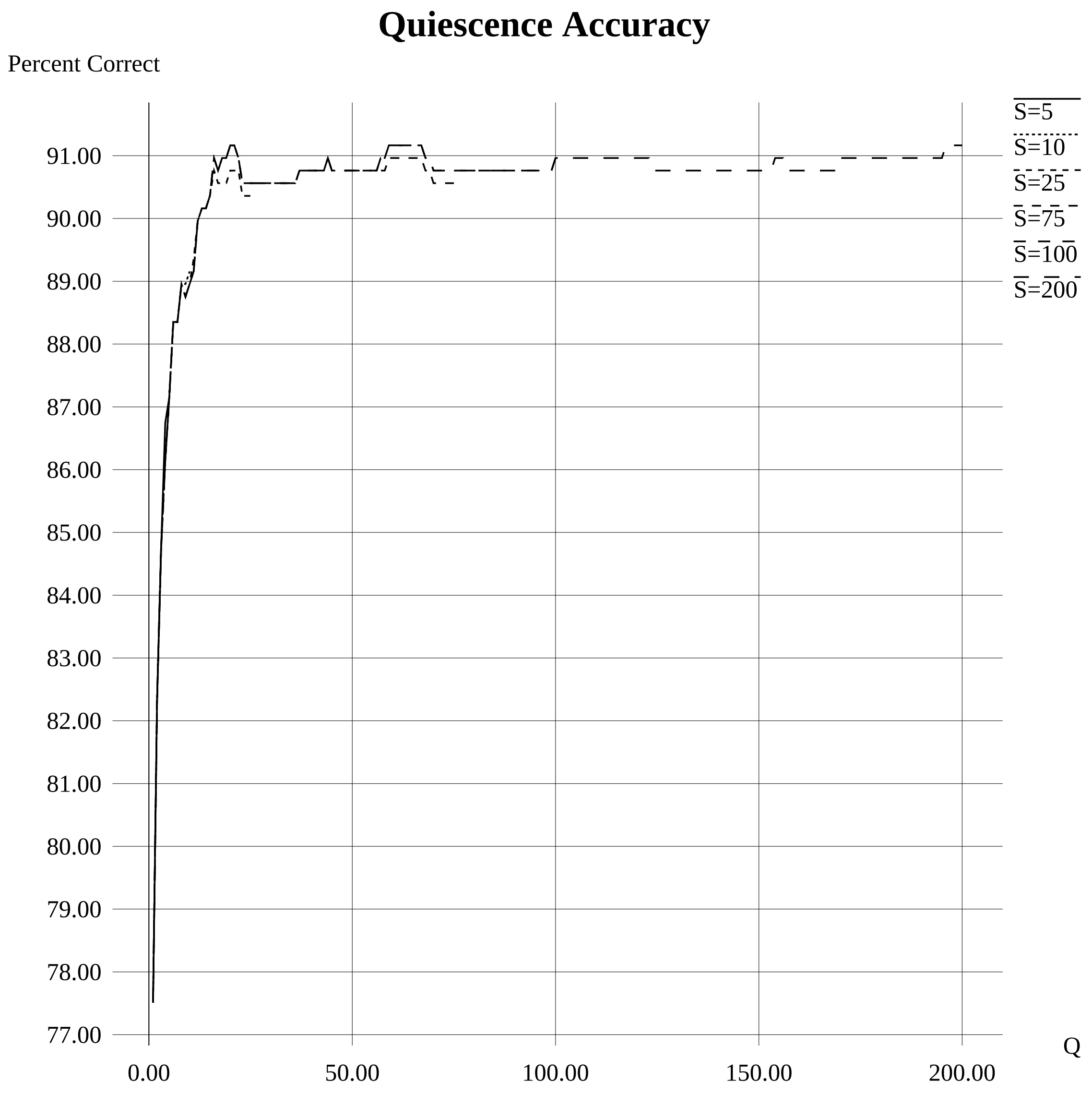

Figure 6.26: Quiescence accuracy for various S values

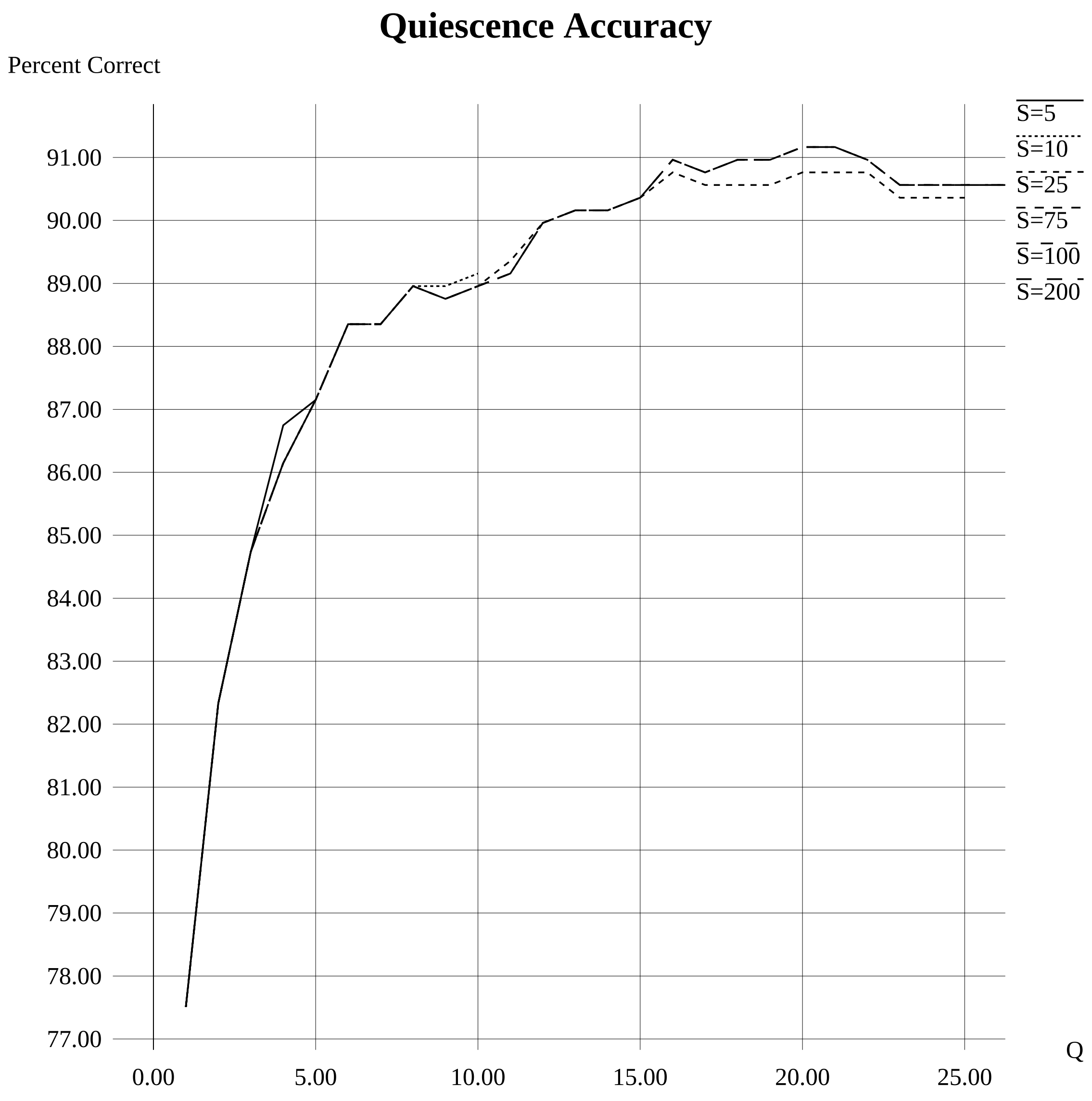

Figure 6.27: Quiescence accuracy for various S values (closeup)

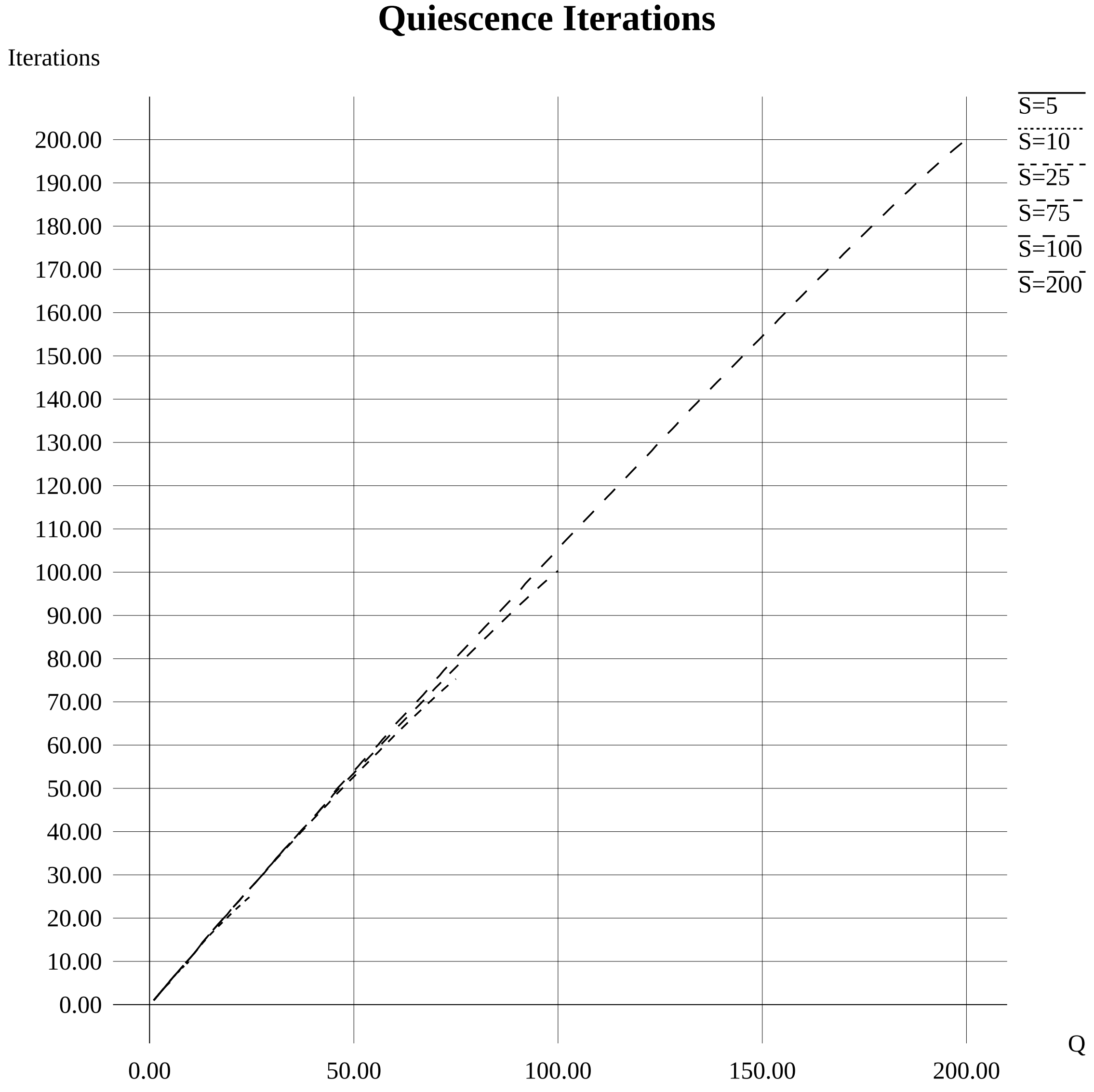

Figure 6.28: Iterations until quiescence

When compared to doing a set number of iterations, as examined in section 6.3.1, a q of 15 is right roughly 90% of the time and requires 16 to 17 iterations. This is the same accuracy as you would get from doing 16 or 17 iterations all the time. This means that quiescence is little better than doing a constant number of iterations.

In summary, quiescence is a strong predictor of accuracy. Moderate to low values of q will get reasonable accuracy. The choice of S appears unimportant. Unfortunately, quiescence is not significantly better than doing a constance number of iterations.

6.4.2 Relative strength of choices

Rather than looking at how long it takes to settle on an answer, you can also look at the difference in value between the top two choices. We will examine two versions of this. In the first variation we will look at the difference between the top two choices. The second variation to be examined is the difference between the two top choices, divided by the number of iterations.

For both variations, we define two parameters. An overall cutoff S, and a difference delta (either absolute, or relative to the number of iterations).

When given two alternatives, there are two possible relationships between them— the one is stronger, or they are equal. Given an infinite number of iterations, the distinction between these cases is possible. With a smaller number of iterations, you can not be certain if the choices are equal, or just very close to equal. You would expect that once there is a difference between the choices, the difference will continue to widen with additional iterations.

There is a reasonable correlation between a non-zero delta value and accuracy, as can be seen in Figures 6.29, 6.30, and 6.31. Values of S smaller than 10 appear to make significant impact on the accuracy of the heuristic. For all values of S, the rises rapidly for deltas up to 2, and the flattens out.

For Figures 6.32, 6.33, and 6.34, we looked at the relative differences between the top two choices

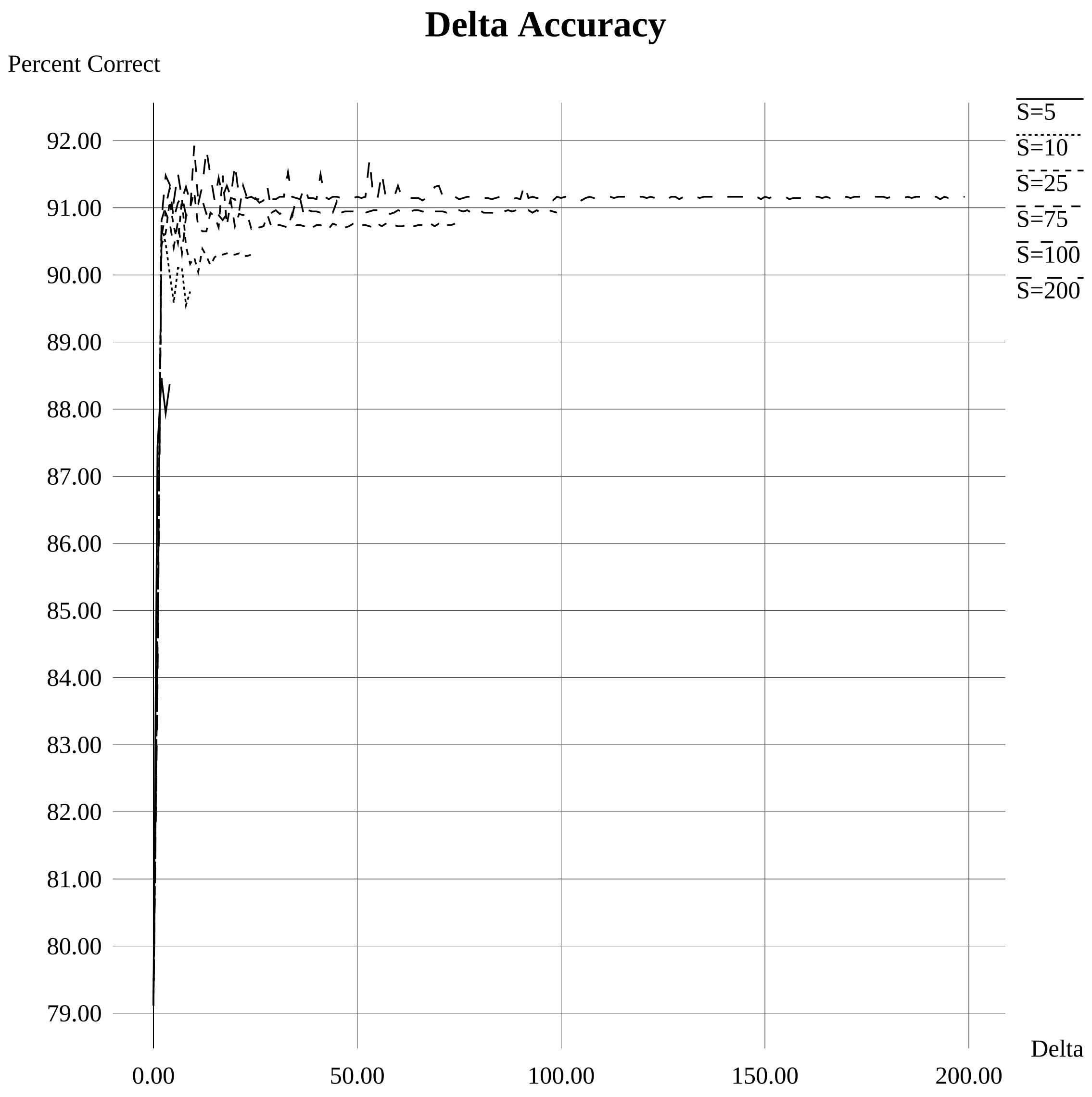

Figure 6.29: Delta accuracy for various S values

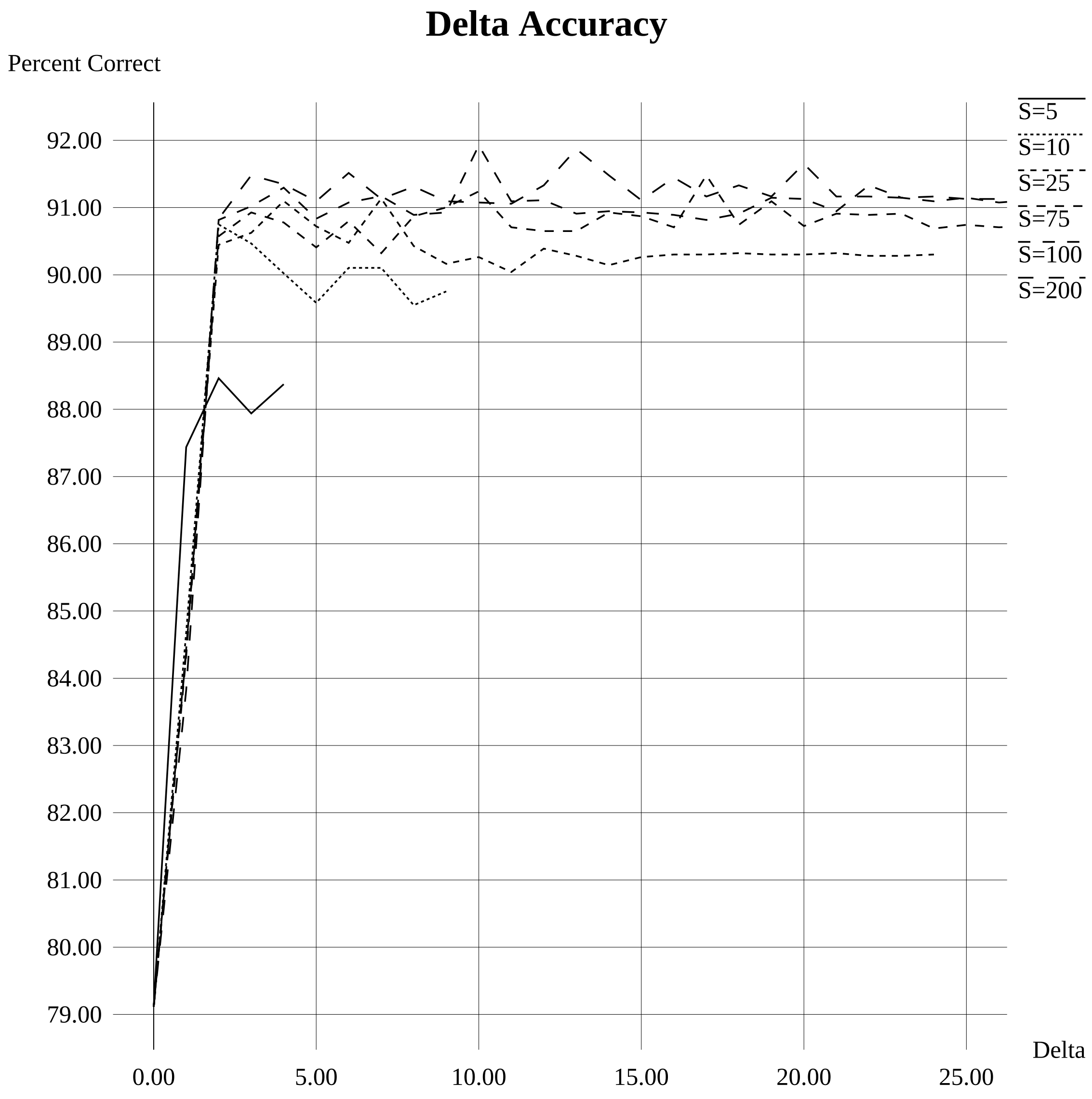

Figure 6.30: Delta accuracy for various S values (closeup)

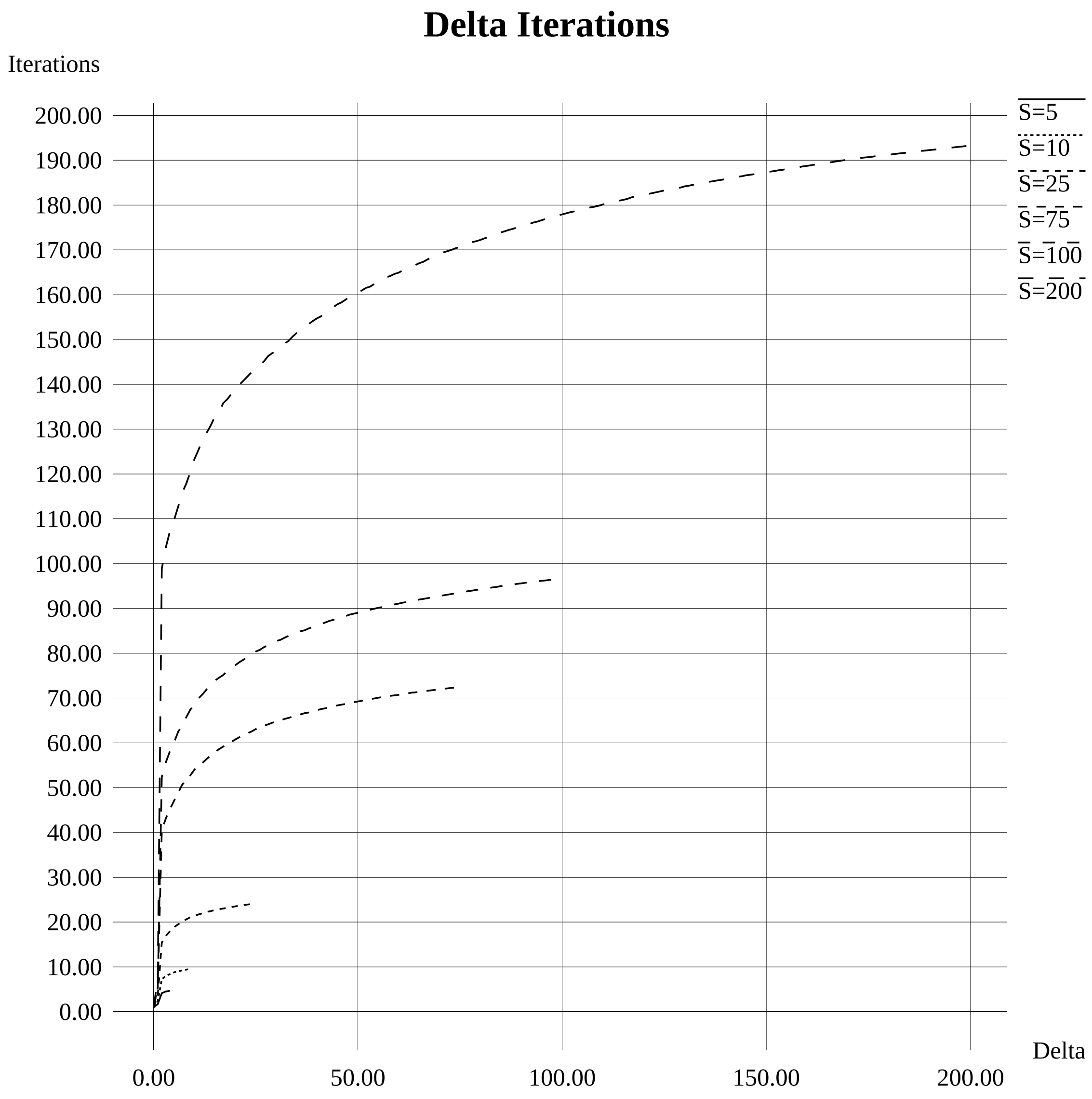

Figure 6.31: Iterations until reaching delta

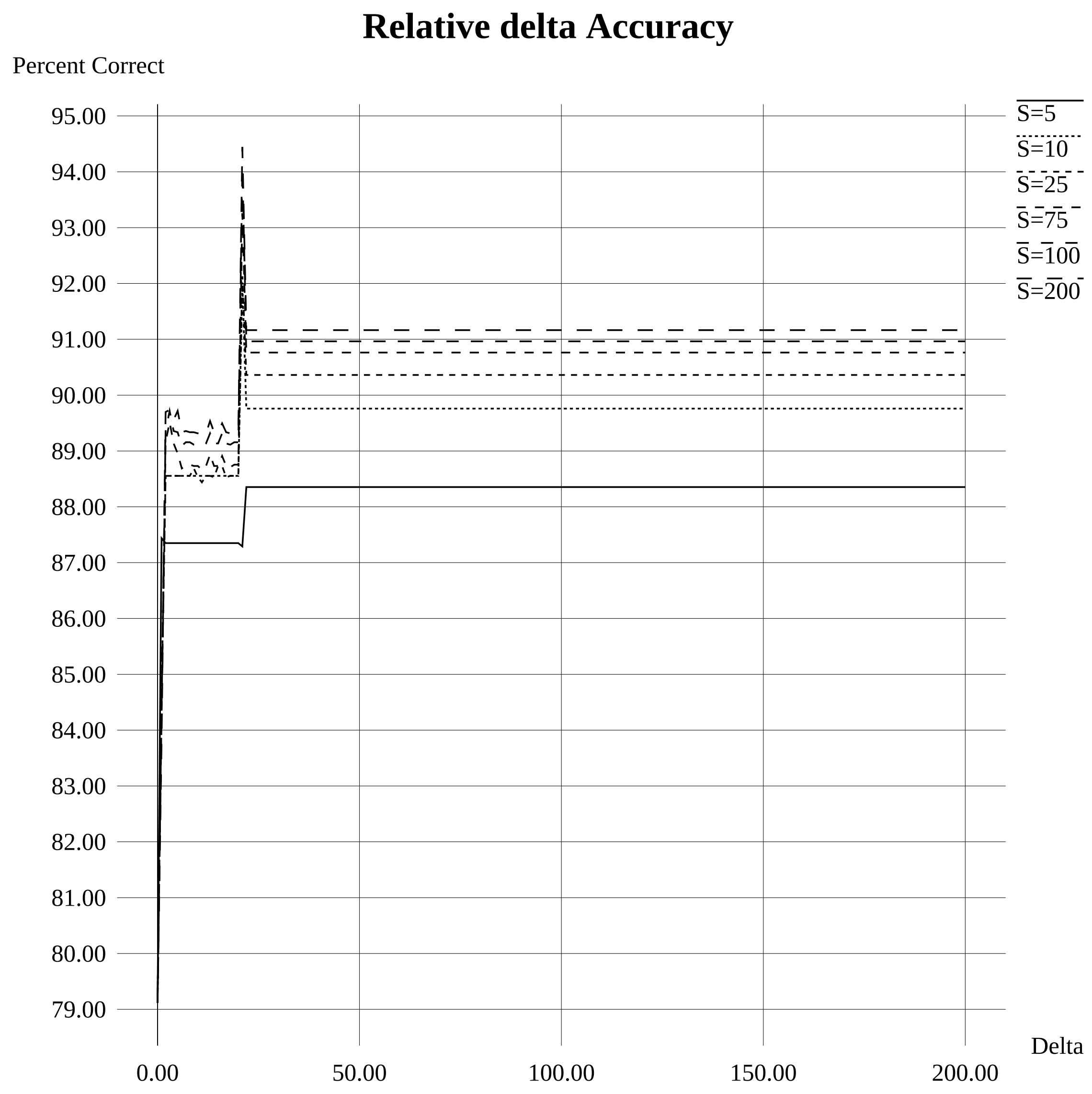

Figure 6.32: Relative delta accuracy for various S values

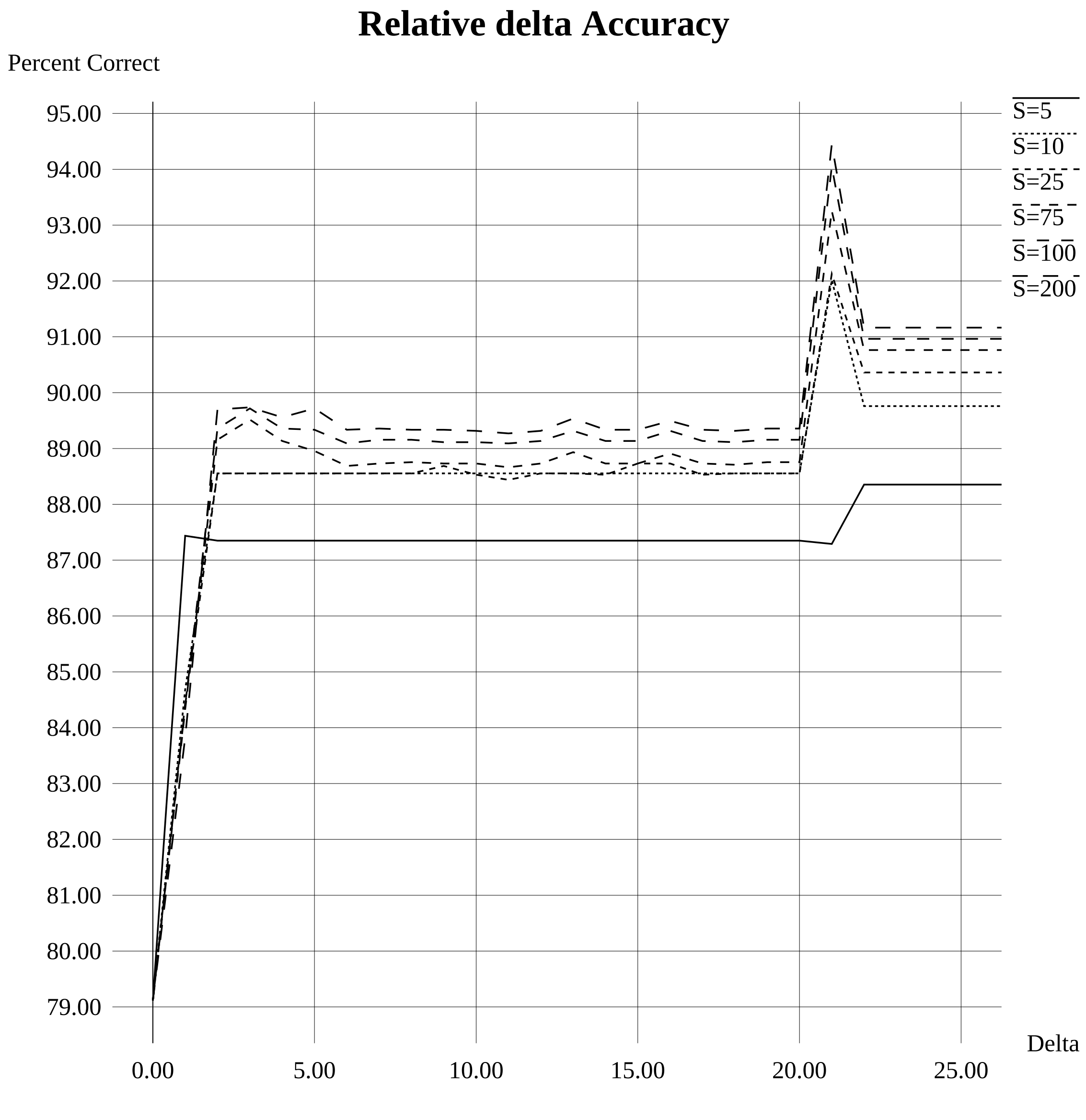

Figure 6.33: Relative delta accuracy for various S values (closeup)

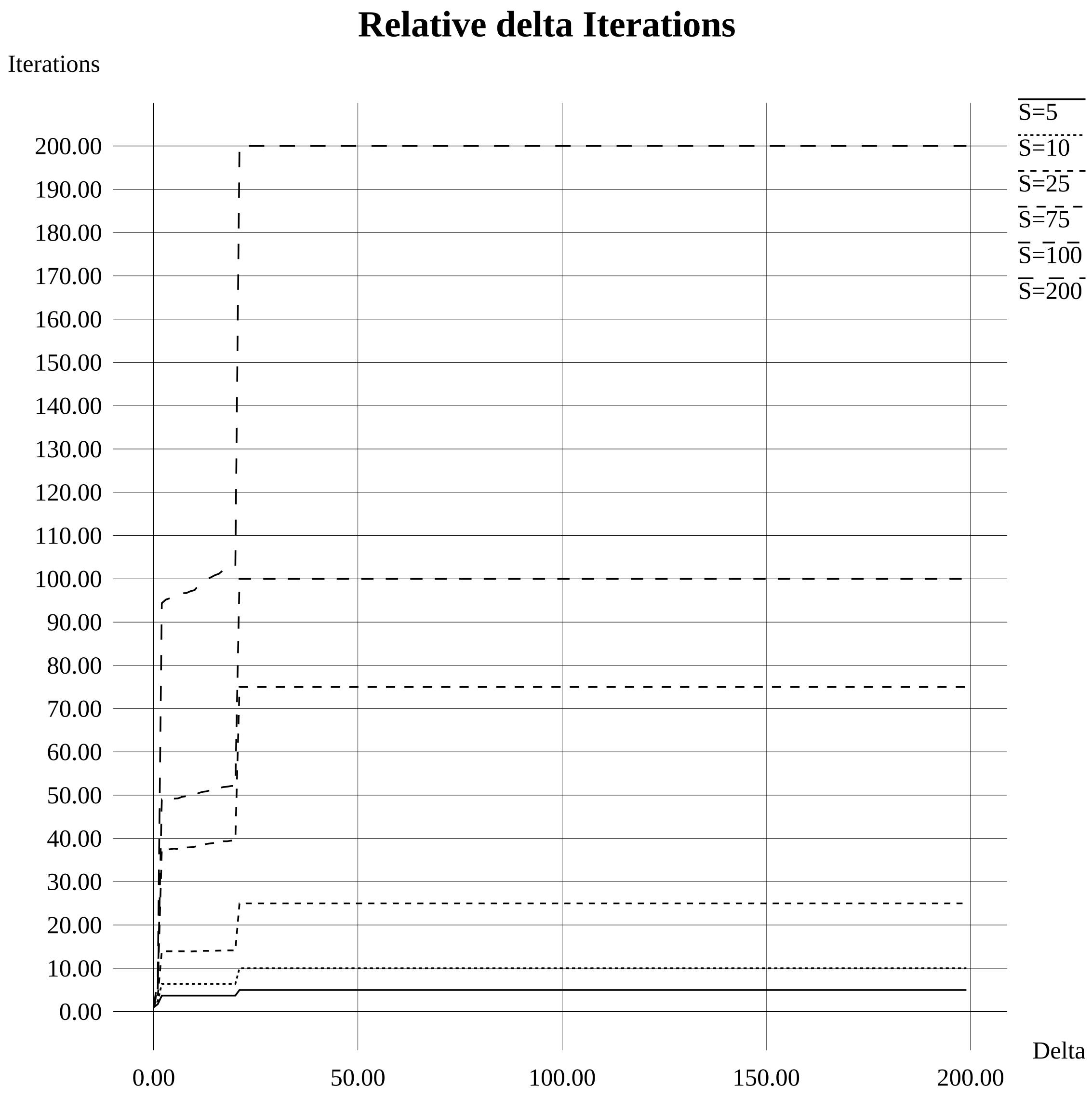

Figure 6.34: Iterations until reaching a relative delta

at each iteration, and divided that value by the number of iterations. We then grouped these values into 200 buckets- 0 to 0.099, 0.01 to 0.199, ..., 1.80 to 1.89, and 1.90 to 1.99. As with quiescence and the absolute delta analysis, we also passed in a value S, for the maximum number of iterations to do in any case. On the graph, the point $x = 0$ corresponds to the first bucket, the point $x = 1$ corresponds to the second bucket, and so on.

As has been seen before, the choice of S has little influence on the accuracy of the curve. Picking relative delta values of 0.02 (the third bucket), or 0.23 (the 24th bucket) appear to give the best accuracy.

6.5 Corrections to human expert answers

After the experiment was run, the 25 problems where the heuristic and human expert disagreed were examined. The human expert decided after looking at them, that 8 of the 25 problems had additional answers which are correct. Three of these problems involved adding an additional valid choice of suit. One problem involved adding an additional card to follow suit with. The remaining four problems involved adding additional discards. Most of these corrections are a result of the code's choice of picking the highest card when there are ties, versus the human's natural inclination to pick the lower one in some circumstances. Two corrections were the code cashing in on a suit where the E/W team had to be void, since N/S had all the remaining unplayed cards in that suit.

6.6 Problems the heuristic solved incorrectly

As discussed in section 6.1, 17 of the 166 non-degenerate, completed problems were incorrectly solved after 200 iterations. There are several reasons why the problems may have been incorrectly solved. As shown in section 3.2, the heuristic can converge on a wrong answer when it is unable to distinguish which node from the information set it is actually at, but thinks it can. A second failing that the heuristic can have is when faced with two answers that are very close in value. With a finite number of iterations, you can not be certain that it has differentiated between them.

As implemented for our experiment, the heuristic always picks the highest valued card (or suit) when there is a tie. This is not always the correct play. In some instances, the heuristic picked a higher card than the human expert suggested because it thought that the two choices were equally good.

6.7 An interesting problem

One problem that had very unusual behavior, was this one. North (the dummy) had $\spadesuit$ 8, $\heartsuit$ 9 8 7, and $\diamondsuit$ J 5. South had $\spadesuit$ K 9 3, and $\diamondsuit$ A Q 10. The unknown cards were— $\spadesuit$ A 10 6, $\heartsuit$ K Q 10 6 5 2, and $\diamondsuit$ K 9 8. North is leading from diamonds. To the human expert, the jack of diamonds is the correct card to play, as follows [David Mutchler, private correspondence]:

> South has one sure trick (the Ace of diamonds) and possibilities of developing additional tricks by finesses in spades (lead a spade from North, hoping East has the spade Ace, so that South's spade King is promoted) or diamonds (lead the diamond Jack from North, hoping East has the diamond King, and play the Ace whenever East plays the King). However, as soon as East or West wins a trick, they are likely to win the rest of the tricks, given the large number of hearts they share. Thus, the spade finesse must be rejected, as it allows East/West to win a trick immediately (with the spade Ace).
>
> With regard to the diamond finesse, if the finesse wins (i.e., if East has the diamond King), then South wins three tricks, while if the finesses loses (i.e., if West has the diamond King), then South probably wins no tricks (since East/West then take lots of heart tricks). Thus, the diamond finesse has expected outcome of 1.5 tricks. This is better that the alternative of simply playing the diamond Ace (no finesse), which has

expected outcome a little better than 1 trick. (It wins two tricks if East has the singleton diamond King, otherwise one trick.)

In sum, the correct play is to take the diamond finesse. The correct card to lead from North for taking the finesse is the diamond Jack. Leading the diamond Jack never does worse than leading the diamond 5 (but see below) and wins an additional trick when East has all three diamonds.

To the computer, 599 of the simulations showed that the jack and 5 were both equal. The one hand that the 5 was better East was dealt the ♡ Q 10 6 5 2, and ♢ K. West was dealt ♠ A 10 6, ♡ K, and ♢ 9 8. In this case, the 5 is the better choice since it allows the North-South team to win an additional trick, as follows [David Mutchler, private correspondence]:

Ordinarily, the diamond Jack and 5 are equally good cards to lead, as the Jack does better only when East has all three diamonds. Apparently, this case (East having all three diamonds) never arose in the 600 simulations. To the human expert, *who does not see the East/West cards*, leading the diamond 5 is never better than the Jack. However, *if the East/West cards are visible*, then there are unusual hands where the leading the diamond 5 is better than leading the Jack. The hand listed above, which occurred during the simulations, is one such hand. When the diamond 5 is led, East plays his singleton King and South wins the Ace. Now South can play a diamond back to the Jack and lead a spade from North toward his King. Eventually South wins a trick with his spade King, as well as the two diamonds he already has won and a third diamond when he is back in the lead via the spade King. All this works only because the heart suit is blocked; East/West can take only one heart trick when East gains the lead with the spade Ace.

In sum, leading the diamond 5 is better than leading the Jack only if the East/West hands are distributed in a certain unusual manner. But South cannot recognize when the East/West hands are so distributed, unless he is peeking at their cards! Thus, the

Monte Carlo heuristic (which simulates such peeking), errs in its advice.

Chapter 7

Summary

7.1 Contributions of this thesis

In this thesis, we have developed a heuristic suitable for use in solving imperfect information games, and examined its theoretical behavior. We have also run an experiment to verify its validity in practice, and to identify bridge specific factors influencing its behavior. From those factors, we have suggested some advice for how many iterations to perform the heuristic, and attempted to generalize these factors to games other than bridge. When comparing the heuristic's choices versus the expert's, several of the inconsistencies were the heuristic finding new alternatives which the expert had not considered. Lastly, we identified and implemented the adaptations that alpha-beta and similar algorithms will need to work with games that don't have strict alternation of players.

7.2 Further work

During the development of the code, several areas where further work could be done were identified.

The code to deal unknown cards has several deficiencies which lower the overall performance of the code. It does not take into consideration past play when generating the unknown hands.

Knowledge that a player is void in the suit lead is not used. Also, given the cards played by a player, other educated guesses can be made as well, particularity from cards played during 2nd hand ("2nd hand low") and 4th hand ("Win or lose as cheaply as possible"). Lastly, knowledge that could be derived from the bidding process is ignored. Coding a way to interpret bidding data is a sizable undertaking in and of itself, even though the bidding process doesn't encourage deception (unlike poker for example).

As suggested by Levy [16], alpha-beta could be called with tighter bounds— namely the number of tricks remaining which must be won to make or break the contract. When the contract is actually reachable, this would presumably speed up the game-tree search significantly. Knuth & Moore [10] discuss the problems associated with calling alpha-beta with tighter cutoffs.

Since the alpha-beta code is the big time consumer, making a slower but more realistic hand generator would make the code more accurate, with negligible additional overhead.

Due to CPU time constraints, the code has not been run on hands with 7 or more cards. A partial run of the 7 card problem set (with 20 instead of 200 iterations) was run, and the results appeared to be consistent with the behavior on other problems, however the number of sampled problems were only half the size of the other sets, and they were only examined in a cursory manner.

Well known programming shortcuts such as transposition tables, translation to specialized hardware or machine code, and only searching to a limited depth, instead of to the end of the tree, should make this heuristic practical to use as far as real time constraints are concerned.

In our experiment, we identified the factors that influence the heuristics convergence on a correct answer, and assumed that they were independent of each other. This is obviously not the case. We therefore propose that a second experiment be done. This experiment should be like ours— 10 hands each suitable for no-trump bids, the north-south team with the stronger hands, etc. Before running a problem, one should identify the factors that we examined in chapter 6 which this problem has, and determine the relative weights of those factors. For example, if we had a hand where we were picking the choice of suit (0th hand), had 6 cards, in three suits, with both a weak team strength and visible strength; we might pick weights of 15%, 40%, 5%, 20%, and 20% respectively. From our experiment, we can generate a graph of the expected accuracies, which we can then compare to the actual performance of the heuristic on this problem.

Bibliography

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[2] Thomas Anantharaman, Murray S. Campbell, and Feng hsiung Hsu. Singular extensions: adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, April 1990.

[3] Bruce W. Ballard. The $*$-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(1,2):327–350, March 1983. Part of a special issue on Search and Heuristics edited by Judea Pearl and published in book form as *Search and Heuristics*, North-Holland, Amsterdam (1983).

[4] Darrin Barr. Experiments in heuristic search utilizing sampling and precomputation. Master's thesis, University of Tennessee, August 1992.

[5] Hans J. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205–220, September 1980. Reprinted in *Computer Games I*, edited by D. N. L. Levy, Springer-Verlag, New York, 1988.

[6] Jean R. S. Blair, David Mutchler, and Cheng Liu. Heuristic search in one-player games with hidden information. Technical Report CS–92–162, Dept. of Computer Science, University of Tennessee, July 1992.

[7] Jean R. S. Blair, David Mutchler, and Cheng Liu. Games with imperfect information. In *Games: Planning and Learning, Papers from the 1993 Fall Symposium*, Raleigh, 1993. AAAI Press Technical Report S9302, Menlo Park, CA.

[8] G. W. Brown. Iterative solutions of games by fictitious play. In T. C. Koopmans, editor, *Activity analysis of production and allocation*, Cowles Commission Monograph 13, pages 374–376. John Wiley & Sons, New York, 1951.

[9] William Feller. *An Introduction to Probability Theory and Its Applications, Volume I.* John Wiley & Sons, New York, third edition, 1968.

[10] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, Winter 1975.

[11] Daphne Koller and Nimrod Megiddo. The complexity of two-person zero-sum games in extensive form. *Games and Economic Behavior*, 4(4):528–552, October 1992.

[12] Richard Korf. Generalized game trees. In *Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI–89)*, pages 328–333, San Mateo, CA, 1989. Morgan Kaufmann Publishers.

[13] Richard E. Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 48(1):99–111, February 1991.

[14] H. W. Kuhn. Extensive games and the problem of information. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games, Vol. II*, pages 193–216. Princeton University Press, Princeton, 1953.

[15] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class Othello program. *Artificial Intelligence*, 43(1):21–36, April 1990.

[16] David N. L. Levy. The million pound bridge program. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence*, pages 95–103. Ellis Horwood Limited, Chichester, 1989.

[17] R. Duncan Luce and Howard Raiffa. *Games and Decisions.* John Wiley & Sons, New York, 1957.

[18] Carol A. Luckhardt and Keki B. Irani. An algorithmic solution of n-person games. In *Proc. of the Fifth National Conference on Artificial Intelligence (AAAI–86)*, pages 158–162, Philadelphia, 1986. Morgan Kaufmann Publishers.

[19] Shelby Lyman. Some top players are defeated by computer gamer. *The Knoxville News-Sentinel*, page F7, June 12, 1994.

[20] Stephen Kent Owens. Programming a computer for playing contract bridge. College Scholars thesis, University of Tennessee, April 1993.

[21] Eric Rasmusen. *Games and Information — An Introduction to Game Theory.* Blackwell Publishers, Oxford, 1989.

[22] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959. Reprinted in *Computers and Thought*, edited by Edward A. Feigenbaum and Julian Feldman, McGraw-Hill, 1963.

[23] Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, April 1990.

[24] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, Seventh Series, 41(314):256–275, March 1950. Reprinted in *Compendium of Computer Chess*, edited by D. N. L. Levy, Batsford, London, 1988.

[25] Martin Shubik. *Game Theory in the Social Sciences.* The MIT Press, Cambridge, MA, 1982.

[26] James R. Slagle and John K. Dixon. Experiments with some programs that search game trees. *Journal of the ACM*, 16(2):189–207, April 1969.

[27] John von Neumann. Zur theorie der Gesellschaftespiele. *Mathematische Annalen*, 100:295–320, 1928.

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

[29] E. Zermelo. Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians*, volume 2, pages 501–504, Cambridge, 1913.

Vita

Howard James Bampton was born in Norristown, Pennsylvania on June 8, 1969. On June 10, 1987 he graduated from Methacton High School. In a unsuccessful attempt to avoid bureaucracies, he attended a small school— Lebanon Valley College, from August of 1987, until August of 1989, when he transferred to Villanova University. In May of 1991, he received his Bachelors of Science in Computer Science. Between August of 1991 and July of 1994 Howard worked on and eventually finished a Master of Science degree in Computer Science. Howard's interests include most forms of games (board, war, computer, play-by-mail/e-mail), skiing, pottery, and cats.