Extending Computational Game Theory: Simultaneity, Multiple Agents, Chance and Metareasoning

by

Ronald J. Bodkin

This report is a slightly modified version of a dissertation submitted to the Department of Electrical Engineering and Computer Science on September 1, 1992 in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: algorithms, artificial intelligence, chance, computer gaming, game theory, metareasoning, multiple agents, multi-stage games, probability, rationality, search, simultaneity, strategic interaction

Abstract

This thesis extends the class of games that can be solved efficiently by computer play. Previous research has focused on two-player zero-sum games of perfect information without chance occurrences. However, computer analysis and control of strategic interactions (e.g., in planning, communication and social science problems) requires more general and more useful forms of games.

We present efficient solutions to two-player zero-sum games where players act simultaneously, including games that have chance occurrences. We also provide efficient solutions to games of perfect information for arbitrary numbers of players, using bounds on payoff sums. We analyze best-case performance, and describe experimental investigations of the efficacy of our algorithms. The theoretical and experimental results indicate that our algorithms obtain significant improvements over the performance of the most obvious algorithms to solve these games. We also detail corresponding extensions to the methods of heuristic play and improved search invented for two-player zero-sum games of perfect information.

Acknowledgments

I would like to thank my advisor, Jon Doyle, for providing assistance and valuable suggestions and for his patience in reading my thesis (and in waiting to read my thesis). I also would like to thank Michael Frank, who read the thesis and implemented the game-specific aspects of the Abalone program, i.e., almost all the program. Thanks also to Pete Szolovits, David McAllester and everyone in MEDG who listened to my ideas and gave me encouragement, suggestions and challenged me to improve my ideas. Vijay Balasubramanian kindly read over some of my thesis, and provided useful advice he learned from writing his own Master's thesis. Thanks also to Mark Torrance for helping me with the diagrams.

My thanks also go out to Stuart Russell who generously provided me with a copy of the source code for his MGSS othello player, and Timothy Huang, who sent me the program and discussed possible extensions to MGSS with me.

This research was supported in part by the USAF Rome Laboratory and DARPA under contract F30602-91-C-0018, and by National Institutes of Health Grant No. R01 LM04493 from the National Library of Medicine.

Contents

Chapter 1

Introduction

Artificial intelligence has long been involved in considering how to make computers play games. In 1949 Shannon discussed the principles of computer play of chess, and Turing manually executed an algorithm to play chess in 1951. Since that time, computer game playing has become a thriving sub-community in Artificial Intelligence. However, it is our contention that it is important to broaden the focus of computer game play.

The general attitude towards games in AI has been to regard them as a kind of puzzle (like chess or other parlor games). John McCarthy has defended computer chess as "the drosophila of AI"[McC90]. While it defends such research, this standpoint towards games holds them as essentially trivial and worthwhile only as a device to discover techniques to be used for other purposes. It is certainly true that computer play of games does offer opportunities to discover useful techniques of general applicability. Many techniques that were invented for computer chess have been employed in other areas of AI (such as iterative deepening, use of heuristic functions, and the approach of producing on-line solutions). Research into computer chess, despite the strong programs it has engendered, still has much work to be done in the areas of strategy and comprehending positions. Computer programs to play Go remain at a very weak amateur level, and it is expected that remedying this situation will also require discovery of new techniques.

However, we believe that considering games to be merely a kind of puzzle misses the broader importance of the topic. It is our contention that the study of game playing algorithms ought to be broadened to include strategic interactions of a general kind. The notion of a game as a general strategic interaction is taken directly from the game-theory community (founded in large part by Von Neumann[vNM44]). It is this conception of games of which we speak in this thesis. We believe that algorithms for dealing with strategic interaction are an important area of consideration in their own right. Just as considering how a computer can make decisions to accomplish goals in the absence of other intelligent agents is an important and worthy area of study, so is considering how one can make decisions when interacting with other intelligent agents. We provide more concrete examples of how such a theory can be useful in section 1.2.

1.1 Proposed Extensions

Now, in keeping with this broader concept of what kinds of strategic interactions ought to be considered, we believe that an important first step is to extend the extremely narrow range of games that have currently been dealt with in a general fashion. Such games have only two agents, alternating moves, no (external) chance factors and the interests of the agents are totally opposed. In order to lay the foundations for handling more general classes of games, we consider how to efficiently decide upon good moves. In this manner we directly extend the existing focus of computational game theory. Naturally, there are many unsolved problems in formulating a prescriptive theory of such interactions that are also applicable to single-agent behavior (such as strategy and planning). The lack of solution to many of these problems will also limit the quality of computer play in our more general classes of games. Indeed, to some extent, our work focuses on direct extension because it is likely to be needed for such applications and does not require us to posit solutions to these very hard problems. However, we believe that the possible uses of extended computational game theory are sufficiently promising that beginning this inquiry will prove advantageous. Moreover, generalizing the game problem may well provide insights that will be useful for more general problems (and for more traditional kinds of games).

There are also deep philosophical questions unique to the problem. These concerns tend to focus on how a game ought to be played. One can solve the games (i.e., prescribe a manner of play) using the assumption of interactions among rational agents. This assumption is typically made in application of game theory. For the most part, we also adopt this assumption, though we do consider some alternative approaches. However, there is no consensus on how rational players will play arbitrary games, even when facing opponents who are also rational and when this rationality is common knowledge. We discuss this topic in sections 4.5 and 5.2.2.

There is an additional concern about rationality. In interactions where there is not total conflict, the consequences of modelling irrational opponents as rational can be far more severe than in those with total conflict. This is especially important when neither the agent nor its opponents can consider all the ramifications of an action, but instead must use some heuristic estimates of value in deliberations. So it is important to consider the idea of rationality in the context of limited computational resources and also how one might construct models of the behavior of other agents. Both of these, naturally, are involved in single-agent problems and in traditional games. However, the manner in which these problems present themselves in more general games is more pressing and quite unique. We discuss these matters later in the thesis, but we focus on the issue of making extensions to the basic framework of interaction.

1.2 Potential Uses

This research has both prescriptive and descriptive potential. Prescriptive uses of game theory include the traditional application of playing games for fun, but also transcend this. Recent work in distributed AI has begun to explore the use of game theory, e.g. [LR92, GDW91]. The focus of such work is individual agents pursuing selfish ends to accomplish various tasks (presumably designed to be of utility to the system's users). Advances in computational game theory might also be employed for modelling, recommendation and analysis of real-world strategic interactions. Possible domains of use include foreign policy analysis, various kinds of arbitration, and strategic business planning. Indeed, computational game theory should be applicable in many of the fields in which games are employed (such as games for educational, operational and research purposes).

We believe that computational game theory also has descriptive potential. The idea is that one can study models of strategic interactions that are too complicated to allow analytic solutions. Application of such techniques in the social sciences would be comparable to the use of econometric modelling in economics. Because game theory is already extensively used in economics and political science (a survey of some such applications can be found in [Shu82]), descriptive uses of computational game theory appear promising.

1.3 Thesis Overview

In this thesis we begin by presenting relevant background material from game theory and computational game theory, i.e., algorithms for computing solutions to games.

After this background, we provide specific algorithms to handle two-player zero-sum simultaneous play games (including an extension to handle games with chance) and to handle sequential play games with arbitrary numbers of players (and bounded payoffs). We analyze the theoretical best-case performance of both of these algorithms, and we provide simulation results to demonstrate the performance benefit of using these algorithms. For the two-player zero-sum simultaneous play game algorithms, we programmed a political simulation game to determine these results. For the N-player sequential play games, we provided simulation results from abstract game trees and preliminary qualitative performance results when playing the three-player version of the game Abalone$^{\text{TM}}$.

For the algorithms to solve both classes of games, we also describe how to extend heuristic solution methods and various optimization techniques that have been developed for two-player sequential play games. We discuss several additional issues relating to each class of games.

For the simultaneous play games, our algorithms employ a number of sub-algorithms to achieve overall goals, and we present an informal, qualitative analysis of the various possible sub-algorithms for these tasks. We present statistical evidence from our case study to support the major assumptions we make when analyzing these sub-algorithms. Our approach to handling these sub-algorithms incorporates a certain degree of metareasoning, and we describe how one might further extend a metareasoning approach.

The chapter on N-player perfect information games also discusses opponent rationality and possible improvements to the usual approach to solving approximation games when dealing with games that have more than two players or that are not zero-sum.

In the concluding chapter, we summarize our accomplishments and we present several possible ways that this work can be extended to improve the utility of computational game theory.

Chapter 2

Background

Before describing our extensions to computational game theory, we review some results from the two major research topics that relate to this topic. We incorporate results and notation from game theory and also from the existing work on computational game theory.

2.1 Game Theory

We restrict our attention to finite, discrete games. That is, games that have a finite number of alternatives available at each position, and finitely many positions. Other categories, such as differential games and potentially infinite games, are interesting topics for future research.[1]

We first give a definition of a game in extensive form. Such a game consists of:

1. N, the number of *players*.
2. A finite tree of game *states*, Γ. Let Γ_V indicate the *successors* of a vertex V in Γ. We define τ to be the *leaves* (or *terminal* nodes) of Γ, ν to be the *interior* (or *non-terminal*) nodes of Γ and A to be the *root* of Γ.
3. An *agent* function $a: \nu \to \{0, \ldots, N\}$. This function determines which agent (either a player or Nature) acts at a given vertex. Players are represented by their number (1 to N) and nature by 0. We take P to be the vertices V in ν such that $a(V) \neq 0$ (i.e., *player* vertices, where a player acts), and C to be $\nu - P$ (*chance* vertices, where Nature acts). Nature acts at random, as described below.
4. A *payoff* function $u: \tau \to R^N$, for $R \subseteq \Re$ (we generally use $R = \Re$). $u_i(V)$ denotes $u(V)_i$. This function associates a vector of *utilities* for the players with each terminal vertex.
5. An *ordering* or enumeration of outcomes for each interior vertex: $\mathcal{S}_V: \{1, \ldots, c(V)\} \to \Gamma_V$, where $c(V) \equiv |\Gamma_V|$. We number the choices available at any stage to refer to them, and the ith successor is produced by choice i. This associates each choice with a successor vertex. We generally write V_i to denote $\mathcal{S}_V(i)$.

[1] Note that a solution of sorts to infinite games can be found by solving finite approximations thereto.

6. A partition H of the nodes in P into *information sets*. The idea is that the player to move at a vertex is able to determine which information set he is in, but he cannot determine the specific vertex among the members of the information set at which he is playing. It is required that for every information set h in H, that every vertex V in h has the same player $a(V)$ and the same number of choices $c(V)$ (since we refer to choices at a vertex by number, having the same number of moves is sufficient to model having the same moves available). We define $a(h)$ and $c(h)$ to be these constant values (for any V in h) for $a(V)$ and $c(V)$, respectively.
7. A *probability* function, d, which maps each V in C to a probability distribution over Γ_V, $d_V: \{1, \ldots, c(V)\} \to [0,1]$ (with $\sum_{i=1}^{c(V)} d_V(i) = 1$).

The idea is that at every chance vertex, a random successor is chosen according to the distribution d_V, and at other non-terminal vertices the player selects a choice from those available at the vertex's information set. The final outcome of play is a payoff vector (of utilities) at one terminal vertex. We use ordinal utility functions, so the payoffs are defined only up to positive affine transformations. So, if we take any game and modify a given player's payoff function by such a transformation, the game is equivalent (specifically, no concept for solving games should be changed by such transformations).

We define H_i to be player i's information sets, i.e., $\{h \in H | a(h) = i\}$.

A game in which every information set (in H) is a singleton is called a game of *perfect information*; any other game has imperfect information. We assume that our games have perfect recall, so that no vertex in an information set can have another vertex in the information set as a descendant. We assume that our games are *non-cooperative*; this means that players are not able to make binding (enforceable) agreements. However, one can model cooperative games using non-cooperative games by explicitly adding negotiation and commitment to (the representation of) the game.

We assume the game is one of *complete information*, that is, every player knows the payoffs of the other players.[2]

We now turn to defining the concept of a *strategy*. A *pure strategy* for player i, s_i, is a map from each h in H_i to $\{1, \ldots, c(h)\}$. Let S_i be the (finite) set of all pure strategies for player i. A *mixed strategy* σ_i is a probability distribution over S_i, (i.e. $\sum_{s_i' \in S_i} \sigma_i(s_i') = 1$ and $0 \leq \sigma_i(s_i') \leq 1$ for each s_i'). Let Σ_i be the (infinite) set of all mixed strategies for player i. The *support* of a mixed strategy, σ_i is $\{s_i | \sigma_i(s_i) > 0\}$.

Note that there is a natural embedding of pure strategies into mixed strategies; this maps s_i into the (degenerate) mixed strategy σ_i, where $\sigma_i(s_i) = 1$, and $\sigma_i(s_i') = 0$ for $s_i' \neq s_i$.

One can also present any game in strategic form. This consists of a set of players, a set of pure strategies for each player, and a map from the cartesian product of pure strategies to payoffs, u. To create the strategic form for a game in extensive form, one takes the expected value of the payoffs from such play. We do not discuss the technical details of such conversion here.

[2] Harsanyi[Har67] has presented a method for handling incomplete games by using an equivalent game of complete, but imperfect, information.

Without presenting the formal definitions[3] we will use the notion of a game, G', being a *substructure* of another game G, which applies if G' results from narrowing the choices available to the players at some of their information sets in G.

2.1.1 Solutions to Games

As is standard, we assume that all the players are rational, and that this rationality is mutual knowledge.

Rationality means that players act to maximize their expected utility. However, in the context of a game, this is insufficient to fully specify how players who are all rational and all aware of this will act. At this point, it suffices to note that many refinements of this concept of rationality have been presented, including some that would give a solution to arbitrary games.[4]

We define additional concepts relevant to solving games, and proceed to describe how one might define the manner in which a rational player would play, given that all the players are rational and that this fact is common knowledge.

Let us define σ, a *strategy profile*, to be an element of

$$\Sigma \equiv \Sigma_1 \times \ldots \times \Sigma_N$$

An *i-incomplete profile*, σ_{-i} is an element of

$$\Sigma_{-i} \equiv \Sigma_1 \times \ldots \times \Sigma_{i-1} \times \Sigma_{i+1} \times \ldots \times \Sigma_N$$

We define the pure strategy analogs: $s \in S$, $s_{-i} \in S_{-i}$, for

$$S \equiv S_1 \times \ldots \times S_N$$

$$S_{-i} \equiv S_1 \times \ldots S_{i-1} \times S_{i+1} \times \ldots \times S_N$$

Also, (σ_i', σ_{-i}) denotes the (completed) strategy profile $(\sigma_1, \ldots, \sigma_{i-1}, \sigma_i', \sigma_{i+1}, \ldots, \sigma_N)$. Likewise, (s_i', s_{-i}) denotes $(s_1, \ldots, s_{i-1}, s_i', s_{i+1}, \ldots, s_N)$.

We extend the payoff function u to (mixed) profiles by defining

$$u(\sigma) \equiv \sum_{s \in S} \left(\prod_{j=1}^{N} \sigma_j(s_j) \right) u(s)$$

This is the expected value of utility (taking the profile as a joint distribution over mixed strategies).

A mixed strategy σ_i *strictly dominates* another mixed strategy σ_i' if it is always better for its player:

$$u_i(\sigma_i, s_{-i}) > u_i(\sigma_i', s_{-i}) \text{ for all } s_{-i} \in S_{-i}$$

[3][HS88] p. 53 gives such a presentation

[4]We further discuss this topic in chapter 5.

Weak domination is similar; in this case the above inequality is non-strict, but there is at least one s_{-i} for which it is strict.

Two strategies σ_i and σ_i' are said to be *duplicates* if every player receives the same payoff whichever is played:

$$u(\sigma_i, s_{-i}) = u(\sigma_i', s_{-i}) \text{ for all } s_{-i} \in S_{-i}$$

Note that this holds for all the players' utilities.

In this thesis, we use domination and duplication only in the case where a pure strategy is dominated or duplicated by a mixed (possibly pure) strategy. Note that the definitions of domination and duplication given above entail that the given relations will hold for any i-incomplete mixed strategy profile as well as any i-incomplete pure strategy profile.

A strategy σ_i is a *best reply* to σ_{-i} if

$$u_i(\sigma_i, \sigma_{-i}) \geq u_i(\sigma_i', \sigma_{-i}) \text{ for all } \sigma_i' \in \Sigma_i$$

A Nash equilibrium is a strategy profile σ such that for each player i, σ_i is a best reply to σ_{-i}. Because the players are rational (i.e., they seek to maximize payoffs, are of the same kind and are aware of this), any solution must be an equilibrium point. Every game has at least one Nash equilibrium (proved in many introductory game theory texts, e.g. [FT91]). However, there may be no equilibria in pure strategies (in some sense this is the reason for using mixed strategies). Also, there may be multiple equilibria that yield different solutions.

In a two-player zero-sum[5] game, every equilibrium point has the same payoff called the minimax value of the game. Additionally, every strategy in an equilibrium profile guarantees its player this value. Given our rationality assumptions, both players will pick an equilibrium strategy (it does not matter which one), thereby guaranteeing that they will obtain their minimax value for the game. In chapter 3, we give a linear program to compute a solution to any such strategic-form game. For two-player zero-sum games, a Nash equilibrium in pure strategies is called a *saddle point*. The name stems from the fact that the payoff for player 1 is the largest value in its column and the smallest value in its row, in a matrix where player 1's strategies correspond to rows. Indeed, this is both a necessary and sufficient condition (in a two-person zero-sum game) for a pure strategy profile to be an equilibrium point. We sometimes refer to the support of a player's solution strategy as the solution support.

Also, in a game of perfect information, the players can use backward induction to deduce best play. Players at a vertex with only terminal successors will pick moves to optimize their payoffs.[6] But players at vertices with subgames of depth at most 1 will deduce this behavior (since their opponents are rational), and they will pick moves to optimize their payoffs given the backed-up value of the vertices with only terminal successors. This process proceeds inductively until the game has been solved. Arguments can be made about the wisdom of adopting such rationality assumptions and about their heuristic application in approximation games. We do not consider these issues at great length for the case of two-

[5] Any constant-sum game can be made into a zero-sum game, by the (positive affine) transformation of subtracting the constant.

[6] For now, we ignore the complication of how to handle the case where a player has optimal choices that lead to the same payoff for him, but different payoffs for others. We discuss this matter in chapter 4.

player zero-sum games; in this case the issues are essentially the same as in two-player zero-sum games of perfect information. At the least, playing in this way is conservative in that it guarantees one will obtain a minimum payoff. We do consider the issue of rationality as it pertains to multiple-player games in chapter 4. We extend this idea of using backward induction to simultaneous play games in chapter 3.

2.2 Computational Game Theory

Computational game theory has dealt almost exclusively with two-player, zero-sum games of perfect information (hereafter described as 2ZPI games). Typical of this kind of game (and those that have been most examined) are chess, go, othello and checkers. The only exceptions to this are a few specific games like bridge and poker, which have been examined from the perspective of highly game-specific planning and rule-based heuristics[Lev87]. While some valuable approaches and insights have been advanced in these specific areas, the only class of games that has been examined by the AI community in a general manner (specifically, producing efficient algorithms for solution) is 2ZPI games.[7] Indeed, to our knowledge, no one has published an efficient algorithm to handle games incorporating chance.

2ZPI games (without chance) have been presented in their extensive forms. Note that such a game can be normalized so the players always alternate moves. We give a naive algorithm for determining the solution of such games, by back-induction. Because the game is zero-sum, we use only $u_1(V)$, as $u_2(V) = -u_1(V)$, for any terminal V.

```
Function Naive-Solution(vertex V)
(It returns a value and a principal variation.)

if V ∈ τ then return (u(V), Λ)
else
  (v, q) ← Naive-Solution(V_1)
  p ← (1, q)
  for i = 2 to c(V)
    (w, q) ← Naive-Solution(V_i)
    if (a(V) = 1 and w > v) or
       (a(V) = 2 and w < v) then
      v ← w
      p ← (i, q)
  return (v, p)
```

A principal variation (hereafter, a PV) is a set of moves that can be played if each player uses a solution strategy, i.e., playing the moves thereon guarantees that each player will receive his minimax value for the game. We use Λ to denote the empty list.

[7] Naturally, it is easy to produce a very slow algorithm to solve any two-player zero-sum game. Wilson's work on computing equilibria for two-player games [Wil72] offers a more efficient approach to solving such games.

Note that if more than one move gives a maximal value, the first one encountered is used (arbitrarily). In a two-player zero-sum game, all these choices are equivalent. For presenting subsequent algorithms, we return only the solution value of the game. We do not indicate how to return moves or the complete PV, as doing so is straightforward but notationally tedious.

There are three important efficient (asymptotically optimal) algorithms for solving such games: the well-known α-β and SCOUT algorithms, and the less known SSS* algorithm.

Both SCOUT and α-β are depth-first searching routines. This allows them to use a minimal amount of space (only on the order of the depth of the game tree) and to update game state by making and unmaking only a single move at a time. For complicated games, the part of the game tree that is being examined is usually generated dynamically (when searching), according to rules for legal moves in a position and rules that determine whether a position ends the game.

SSS* is a best-first algorithm that examines a subset of the nodes that are examined by α-β. It is more complicated and not as relevant to our subsequent discussion. We refer the reader to [Pea84] for a complete description.

2.2.1 The α-β Algorithm

α-β is a directional algorithm: it examines the leaves of the tree using a strictly in-order traversal (ordered by move numbers). The naive algorithm given above is also directional. α-β employs a lower bound α and an upper bound β to cut off search. If the value of a position lies above the upper bound (for a move by player 1) or below the lower bound (for a move by player 2), search is cutoff. The move leading to the position is said to be refuted. Note that deep cutoffs can occur. That is, search information from an ancestor that is not the immediate parent of a vertex can result in a cutoff (at that vertex). We can write the algorithm more formally as follows:

```
Function α-β(vertex V, number α, number β)
(It returns a value.)

if V ∈ τ then return u_1(V)
else
  if a(V) = 1 (for the first player) then
    b ← α (α lower bounds the values of interest)
    for i = 1 to c(V)
      r ← α-β(V_i, b, β) (evaluate this move)
      if r > b (i.e., it is better than the current best) then
        if r > β (i.e., it exceeds the upper bound) then
          return r (cut-off search)
        b ← r (set the lower bound)
    return b (the best move value)
  else (for player 2)
    do the dual of the above:
    b ← β
    for i = 1 to c(V)
      r ← α-β(V_i, α, b) (evaluate this move)
      if r < b then
        if r < α then
          return r
        b ← r
    return b
```

Calling α-$\beta(V, -\infty, \infty)$ evaluates a position V. Indeed, if we have m, M such that for each V in τ, $m \leq u_1(V) \leq M$, then α-$\beta(V, m, M)$ will evaluate a position V. A proof that α-β returns a valid solution is given in [KM75].

2.2.2 The SCOUT Algorithm

SCOUT uses a test and an evaluate algorithm. The basic idea is to test a move to determine whether it is better than the best known move, and to evaluate it only if it is.

```
Function SCOUT-Eval(vertex V)
(It returns a value.)

if V ∈ τ then return u_1(V)
else
  b ← SCOUT-Eval(V_1) (evaluate the first child)
  for i = 2 to c(V)
    if (a(V) = 1 and SCOUT-Test->(V_i, b)) or
       (a(V) = 2 and SCOUT-Test-<(V_i, b)) then
       (this move is the best of the first i moves, for the player)
      b ← SCOUT-Eval(V_i) (set the best move)
  return b (return the best value)
```

```
Function SCOUT-Test-> (vertex V, b)
(It returns a boolean.)
if V ∈ τ return u_1(V) > x
else
  if a(V) = 1 then
    for i = 1 to c(V)
      if SCOUT-Test-> (V_i, b) (player 1 can force a value of b) then
        return true (the value of V is more than b)
    return false (player 1 cannot obtain b)
  else (for player 2)
    for i = 1 to c(V)
      if not SCOUT-Test-> (V_i, b) then
        return false
    return true
```

SCOUT-Test-< is the dual of this.

2.2.3 Asymptotic Costs

The most common measurement of the cost of searching a game tree is the number of terminal nodes examined. If we assume that every node in the game tree has at least two children[8] then all the algorithms in question will have their running time dominated by the time spent examining terminal nodes.

Let us take $I_A(b, d)$ to be the number of terminal evaluations performed by algorithm A in searching a uniform game tree (for a 2ZPIwithout chance moves) of depth d and breadth b. We consider this number in the best and worst cases, and also consider it for an average case. In the average case, I_A denotes the expected number of evaluations in a tree where terminal values are independent random variables, with a continuous distribution function F.

The algorithms achieve improved performance by not examining all the terminal nodes. To measure this improvement, we compute a branching factor for the algorithms; this determines the branching factor that would cause the naive solution algorithm to examine the same number of nodes. The asymptotic branching factor is defined as:

$$R_A(b) \equiv \lim_{d \to \infty} [I_A(b, d)]^{1/d}$$

Best Case: In this case,

$$R_{\mathrm{SSS}^*}(b) = R_{\alpha\text{-}\beta}(b) = R_{\mathrm{SCOUT}}(b) = b^{1/2}$$

This is the lowest possible value for any algorithm that solves this class of games. [KM75] proves that

$$I_{\alpha\text{-}\beta}(b, d) = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

[8]Any game can be placed into this form, by replacing vertices with a unique successor vertex by that vertex; this is done until no such vertices remain.

This best case is achieved when the tree is ordered such that the first move is a move that gives the player his minimax value for the node. We call a tree that is so ordered best-ordered. Knuth and Moore also provide a proof that this is the fewest possible terminal evaluations for any algorithm searching this class of games. Hence the above value for $R_{\alpha\text{-}\beta}(b)$ and its optimality. The value for SSS* comes from the fact that it examines a subset of the nodes that α-β does. SCOUT also has this bound; for a best-ordered tree we obtain the best case:

$$I_{\mathrm{SCOUT}}(b,d) = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

Worst Case: In this case, α-β examines all b^d terminal nodes once. So, for the worst case:

$$R_{\alpha\text{-}\beta} = b$$

In the worst case, the Test routines in SCOUT could not do more than call Test on all b children, so the number of terminal nodes that must be evaluated to test a depth d tree is at most b^d. And SCOUT-Eval cannot do more work than evaluating all b children and testing $b-1$ children. So for $d > 0$,

$$I_{\mathrm{SCOUT}}(b,d) \leq b I_{\mathrm{SCOUT}}(b,d-1) + (b-1)b^{d-1} \leq d b^{d-1}(b-1)$$

Also, $I_{\mathrm{SCOUT}}(b,d) \geq b^d$ because on a tree where the moves are sorted in increasing value to their player, each evaluation recursively evaluates all b children. So we conclude that SCOUT's worst case is:

$$R_{\mathrm{SCOUT}} = b$$

Average Case: For the average case (as described above), we have

$$R_{\mathrm{SSS}^*} = R_{\alpha\text{-}\beta} = R_{\mathrm{SCOUT}} = R^*(b)$$

where $R^*(b)$ is the positive solution to $x^b - (1+x)^{b-1} = 0$

A proof of these equalities is contained in [Pea84].[9] This is also the lowest possible value for any algorithm that solves games (proved in [Tar83]).

2.2.4 Heuristic Play

Because even in the best case solving 2ZPI games takes an exponential amount of time, there are many games that cannot be solved exactly. Indeed, most of the games that have been examined are of this type. Shannon first described the basic idea of how one might play such a game[Sha88]. The idea is to solve an approximation game; a game in which certain non-terminal vertices are treated as terminal vertices and given a payoff value that approximates the expected value of the payoff of the subgame rooted at the vertex. This is done by use of a heuristic function that scores various features of the game. [Sam59] describes a classic approach for using learning to generate the parameters in such

[9] Pearl's proof that $R_{\mathrm{SCOUT}} = R^*(b)$ is flawed: we supply a correction in [Bod92].

a function, [LM90] describes a Bayesian approach, and [FU92] describes a system that also automatically generates features.

Shannon described two types of search. His type A search involves searching an approximation game in which all the vertices at a given depth are treated as terminal. He also proposed a more selective type B search that would search "forceful" variations.[10] Nowadays, we tend to apply this by using some other criteria to determine whether vertices are to be treated as terminal. In chess (for which this was first considered), the initial goal was to search to quiescent positions.

2.2.4.1 Selective Search

The most common technique for doing selective search is to adopt a non-standard measure of depth. Such a measurement is updated whenever a move is made: it is either increased or it remains constant. Typically, such measurements are either incremented by one, or simply kept constant (an *extension*). An extension can be made based on static, game-dependent, factors. For example, a common use in chess is to extend search when a piece has just been captured or if the player is in check. Extensions can also be made in a dynamic, general fashion such as singular extensions[ACH90] or Null-Move Quiescence Search[Bea90].

The idea of singular extensions is to not increment depth when exploring a *singular* move. A move is singular if its value is better than that of all the other moves available at the vertex by the singular margin, Q, which should be "a significant amount." In describing this, we always adopt the perspective of the first player making the move: for the second player, one considers the dual.

In practice, it is too expensive to determine what moves are singular by this definition. Instead Anantharaman *et al.* do so only for moves on the PV (this requires iterative deepening, as described in section 2.3.1). They also extend some moves at vertices that are *fail-high*. A vertex is fail-high if it returns a lower bound higher than the current value of the PV, v. The moves so extended are those that are *fail-high singular*. A move is fail-high singular if it is made at vertex that is fail-high and it returns a value that is better than v and all the other moves lead to positions that are determined to have a value below $v - Q_h$ in a depth $d - r$ ($r \geq 1$) search. In this, Q_h is the singular fail-high margin (analogous to the singular margin). The shallower search is required to avoid destroying the efficiency gained by not searching the other moves. The idea of obtaining information by use of a shallower search at internal nodes is discussed extensively in chapter 3.

Singular extensions were devised for the α-β algorithm. To test whether a move's value is below x, one searches using a "zero-width" bounds window of $\alpha = x - \epsilon$ and $\beta = x$. In this, ϵ should be smaller than the difference between any two payoff values. This idea of using windows in α-β is also described in section 2.3.3. Naturally, one can use singular extensions for SCOUT (by using the Test procedure).

New Approaches to Selective Search: In addition to quiescence searching, a number of other ideas for searching more selectively have been proposed. All of these ideas show

[10] An additional criterion for Shannon's type B search was to not examine "totally pointless" variations; α-β appears to satisfy this requirement by virtue of its cutoffs of refuted moves.

promise, but none of the approaches has been demonstrated to be superior when actually playing games. However, we believe that these suggestions suggest a promising area for future improvement of game playing programs.

McAllester, in [McA88], proposed the idea of using *conspiracies*. A conspiracy is a set of leaf vertices (in a search tree) such that changing each leaf's value can change the value of the root position by a certain amount. To use conspiracy theory, one can search those nodes that are members of the smallest conspiracies that can alter the root value by some margin. The intention is to increase confidence in the value of the root score, by increasing the number of leaf values that would have to change to alter it by the margin. An analysis of some of the possible advantages, and current shortcomings of using conspiracy theory is given by Schaeffer ([Sch90]).

Russell and Wefald, in [RW89, Rus90], have developed an approach to computer game play based upon their theory of metareasoning. Their MGSS* and MGSS2 algorithms attempt to expand those leaf vertices that are most likely to change the best move at the root. This approach also gives more selectivity; it attempts to search those leaves that whose search will give the most expected benefit to the player.

Rivest, in [Riv88], has proposed propagating values with an approximation of the minimum and maximum functions. This permits the use of sensitivity analysis, to determine the leaf node upon which the root score most depends.

These approaches all suffer because they expand nodes in a best-first manner. This requires the time overhead of making and unmaking several moves to reach successive positions and the space overhead of storing a partial game tree (the space required is exponential in the depth of the tree). It may be possible to devise lower overhead approximations to these algorithms such as modifying a depth-first search procedure to expand using a close approximation (possibly in conjunction with iterative deepening).

The MGSS* and MGSS2 algorithms search only singleton conspiracies (which means that they have a limit on the number of vertices they can examine, no matter how much time they are allocated). This limitation is an outgrowth of the "single-step" and "meta-greedy" metareasoning assumptions that Russell and Wefald employ. Together, these assumptions lead to considering only single computational actions, and to assuming that the value of such an action can be approximated by the benefit it would give if after it was made, no further computation was undertaken. This limitation on the amount of consideration possible must be circumvented for such a metareasoning approach to be a possible alternative for use in playing games.[11]

2.2.4.2 New Approaches to Approximation

The biggest theoretical problem with the heuristic of searching approximation games is that it does not allow us to capture information about the uncertainty in position estimates. Pearl has demonstrated the possibility of search depth pathology when searching approximation trees (in [Pea84]). The pathology is that searching more deeply can actually increase the probability of making an erroneous move. He proved that uniform game trees with leaf

[11] Russell has suggested using these meta greedy algorithms to search the leaf nodes in an α-β search, thus providing a kind of selective extension.

values that are independent and identically distributed exhibit this pathology. Pearl also demonstrated that a game with traps would not exhibit the pathology, and he observes that value dependencies might also remove such pathology. However, even for those cases where searching approximation games more deeply is beneficial, explicitly accounting for the uncertainty of terminal position values has the promise of improving decision quality.

There are several alternatives to the approximation approach given above that do not use a simple heuristic payoff function. Berliner's B^* algorithm ([Ber79]) uses two heuristic estimates: a pessimistic estimate and an optimistic estimate. The algorithm searches until one move's pessimistic value exceeds the optimistic value of all the other moves (by either trying to raise the value of the best move, or by lowering the value of the other moves). This algorithm has not been used for actually solving games, though it does represent an interesting approach to accounting for uncertainty. Palay (in [Pal83]) extended B^* to use probability distributions instead of upper and lower bounds.

Other suggestions are based on having terminal estimates represent a probabilistic quantity. Most research in this area has dealt with games that have two outcomes: one of the players wins and the other loses. In this context, one can assign a probability of victory to a terminal node. Pearl (in [Pea84]) suggested the idea of using the product rule to propagate such probabilities, so that if a player has k moves, leading to positions with probabilities of winning of $p_1, \ldots, p_k$, the probability of winning at this vertex is $1 - \prod_{i=1}^{k}(1 - p_i)$.

Baum, in [Bau92], points out that using probabilistic combination suffers from the problem that players must actually select moves. In other words, it is not sufficient that there is likely to be some way to win from a position because the player must select a move that actually does win. He suggests that one should use probability propagation to account for the closer scrutiny that can be given to a given position if it is actually reached during play. Accordingly, he presents a method of interpolating between minimax propagation and probabilistic combination.

To extend such approaches to dealing with expected values would require that we use a class of probability distributions that is closed under the operations of taking the maximum and minimum, or at least a tractable approximation thereto. Palay [Pal83] discusses possible approaches to this problem, using either polynomial approximations or representative points to model distributions.

Hansson & Mayer have used their metareasoning approach to develop a system that maximizes the expected utility of the chosen move. Their BPS system (described in [HM89]) treats expansion of a node like receipt of experimental evidence and performs a Bayesian update of the expected value of a move based on this evidence. They also use metareasoning to control the search procedure (possibly a fixed metareasoning rule, like that employed by Russell). Their approach to the problem of handling expected utilities is to compute a heuristic probability of various outcomes occuring.[12]

[12] This approach is a useful extension of the idea of assuming two outcomes (a win or a loss), but would be hard to employ in a situation where many outcomes are possible.

2.3 Optimizations

A number of additional techniques to improve the performance of programs that play 2ZPI games have been devised. Newborn (in [New89]) discusses these topics in greater depth, in the context of computer chess.

2.3.1 Iterative Deepening

One of the most important techniques is iterative deepening. This was initially motivated by the difficulty of determining how deep to search an approximation tree (it is hard to estimate the time needed to search to a given depth). Instead, use of iterative deepening entails simply searching increasingly deep trees: to depths $1, 2, 3, \ldots$ until an allocated amount of time runs out.[13] The cost of the earlier iterations is dominated by that of the last completed iteration (for trees that are "nearly" of uniform depth), so this approach is virtually cost-free. Iterative deepening also allows for improved search during an iteration, by using information stored from the previous iteration. This factor can be sufficient to actually improve the performance of iterative deepening over simply doing a search to the depth reached by the final iteration. Iterative deepening was pioneered by Slate and Atkin (described in [SA77]).

One piece of information stored during an iteration is the principal variation (the PV). On the next iteration, each vertex on the PV (from the previous iteration) has the move that is in the PV ordered first. This tends to improve search by increasing cutoffs (for SCOUT, increasing the likelihood moves will be successfully bounded by test). By storing principal variation from the previous iteration, we can use information from an iteration that was stopped, because time ran out. Specifically, if there is a new best move at the root, it should be used in preference to the previous iteration's (since it is better than the first move, which was the best move from the last iteration).

2.3.2 Transposition Tables

Another useful method for improving efficiency is to maintain a transposition table. This table is used to prevent repeated search of states that can arise through different sequences of moves. The transposition table is usually implemented as a hash table that stores information about previously searched states.[14] Before searching a state, the transposition table is checked to determine whether the state has already been searched. If it has been searched, then it may be possible to use the value stored. This can be done only if the state was previously searched to the required depth and if the value information suffices. The information will suffice if either the value determined by search was exact, or if only a bound on the value was generated but the bound is at least as tight as the bound required in the current context. Otherwise, the state must be re-searched. However, the transposition table

[13] If at least half the allocated time is used at the end of an iteration, the next iteration should not be started.

[14] Ideally, the table stores a unique key for each state, though, in practice, a hash function is computed and extra check bits are stored. While this allows for the possibility of mistakenly using hashed information about a different state, this is extremely unlikely with a good hash function.

also contains the best move found when the state was previously searched. So if the state must be searched, this move is ordered first to improve performance. In this manner the transposition table also conveys useful information from previous iterations to the current iteration. Note that the table can also be used to assist move ordering by checking whether the successors of a state are in the table. If search must proceed, then the moves are ordered based on transposition table values.

2.3.3 Windowing

Various techniques for improving the efficiency of α-β search use the idea of not setting the initial α and β bounds to $-\infty$ and $+\infty$. We describe a simple "windowing" technique (it appears in [New89]). We adopt the perspective of the first player; the dual is performed when searching a move of the second player. One sets the initial (α, β) values to be $(V - \lambda, V + \lambda)$, where V is the value from the previous iteration and λ is a margin that is set to be significantly larger than typical amount by which the root score changes from one iteration to the next. If no moves have a value of at least $V - \lambda$, then a new pass (in which all the moves are re-searched) begins with the window $(-\infty, V - \lambda)$. If a move has a value of at least $V + \lambda$, then one uses the window $(V + \lambda, V + \lambda + \epsilon)$ for the remaining moves (ϵ should be smaller than the difference between any two payoff values). If another move is found with a value of at least $V + \lambda + \epsilon$, then a second pass is performed, searching the initial move (that had a value of at least $V + \lambda$), the current move (that had a value of at least $V + \lambda + \epsilon$) and all moves ordered after this. In the second pass all these moves are searched using the starting window $(V + \lambda + \epsilon, +\infty)$. The α value is, as usual, raised to the highest value of any evaluated move. This procedure might not determine the exact value of the best move, but it will determine the best move.

There are other, recursive, techniques that use windowing at the other vertices in the tree, with the same idea of not obtaining as much unimportant value information, thereby expediting search. These techniques are effectively a combination of the SCOUT method of testing before searching and the use of upper and lower bounds as in α-β. One such approach is the NegaScout algorithm [Rei83, RSM85].

2.3.4 Non-Speculative Forward Pruning

In some games, search can be expedited by non-speculative forward pruning (first described in [Fre83]). This procedure uses a bound on the amount by which the evaluation function can change in a single move. In some cases, one can cutoff search at internal nodes without expanding any children (by computing the extent to which the heuristic value of the current state can change after making d moves, if one is doing a depth d search). For α-β, this can be done if the value of each successor can be bounded below α, or above β. For SCOUT, the relevant condition is whether the value of each of the leaves is always above or below the test value.

2.3.5 Move Ordering

Move ordering is an area of critical importance in any game-playing program: both α-β and SCOUT achieve best-case performance when moves are best-ordered.[15] Game playing programs use many techniques for move ordering that allow them to come very close to best case performance. In addition to the move ordering information from the transposition table and the PV, programs use various heuristics to order moves. Some are game-dependent and use various (static) features of moves in a manner similar to that used by the heuristic function that assigns payoffs to vertices that are terminal in the approximation game. One popular dynamic method is the killer heuristic (invented by Gillogly [Gil72]). For the α-β algorithm, it stores moves that generated cutoffs on a killer list, and orders these moves at or near the first entry when subsequently encountered. In the SCOUT algorithm, a move that caused early exit during a test can be stored as a killer move. There are numerous different methods of ordering, adding moves to and deleting them from the killer list. The history heuristic (invented by Schaeffer [Sch83]) generalizes the killer heuristic. The idea underlying this heuristic, as with killer, is that moves that are good in one context are often good in another.

2.3.6 Other Optimizations

Some addition techniques that have been used to improve performance include game-specific opening moves databases and databases or special rules for endgame play. Another technique that provides a fairly significant constant speedup of programs is incremental updating. This is done by making minor changes to global data whenever moves are made (and unmade). This is applicable for changing the overall state of the game (instead of copying the data structure that represents this state), for computing (parts of) heuristic evaluation functions and for computing other functions (like transposition table hash codes). The incremental techniques used by the current computer chess champion, Deep Thought, are described by Hsu[HACN90]. Many programs search when the other player is deciding on his move; this is done by guessing the opponent's best move (the second move on the PV) and searching as if it were made; if the guess is right, then the search made while waiting for a move can continue, saving time.

2.3.7 Parallelism

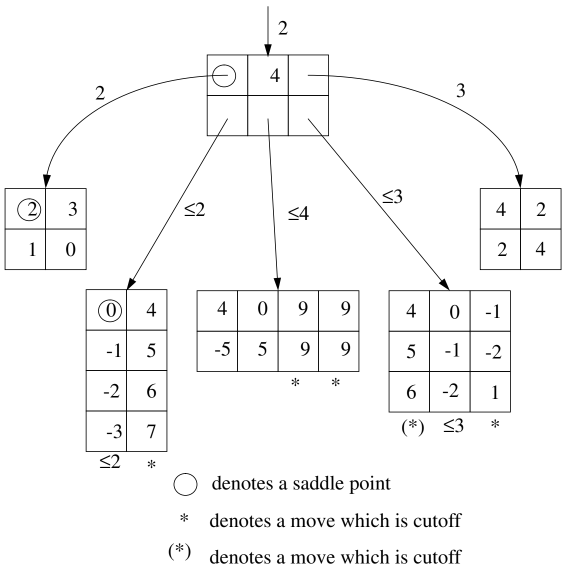
The best-known technique for using parallel processing to compute the α-β algorithm is principal variation splitting ([MC82]). This technique relies on iterative deepening. When performing principal variation splitting, all the processors search down to the last vertex in the PV, then the processors independently divide up each of the children of that vertex and search them. This is then done for the parent of that vertex on the PV, and so on. Fuller descriptions of principal variation splitting and a survey of other approaches to parallel search in α-β appear in [New89, MOS86]. A parallel version of Scout is presented in [AD83].

[15] If the best move is not first, having relatively strong moves ordered early is better than having a random ordering (or one where weak moves are ordered early).

Chapter 3

Simultaneous Play Games

We consider a class of games in which all the players move simultaneously, and then discover the outcome of their respective moves. This is more general than games of perfect information, for we can consider the other players, who do not move when a single agent does, to have a single "null" move at this stage. However, such games are less general than arbitrary games in extensive form. Games of this type are naturally illustrated as shown in figure 3-1; this method of illustration motivates our name for this class: "matrix tree" games. Such games are also known as "multi-stage games with observed actions."[1]



((*) denotes a move that could be cutoff with a different ordering)

Figure 3-1: An Example of a Matrix Tree Game

Figure 3-1 represents a zero-sum matrix tree game with two players. The cells with numbers are terminal positions; the number indicates the utility of the outcome for player

[1]Fudenberg [FT91] p. 70

1 (and player 2's utility is the opposite, since the game is zero-sum). In this figure, player 1 chooses rows, and player 2 chooses columns.

This particular game can be modelled by an extensive form in which player 2's information set for any move consists of all successors to player 1's moves (and one can extend this approach to modelling arbitrary N-player matrix tree games). However, we prefer to use a matrix-tree normal form to present this kind of game.

3.1 Notation

A (finite) N-player game in matrix-tree normal form consists of:

1. N, the number of players
2. A finite tree of game states, Γ. Let τ be the leaves in Γ, ν be the interior vertices in Γ and A be the root of Γ. We denote the successors of $V \in \nu$ by Γ_V.
3. An alternatives function, $a: \nu \to \mathcal{P}^{N+1}$, for $\mathcal{P} \equiv \{1, 2, 3, \ldots\}$. It determines the number of possible alternatives that each agent has at a given non-terminal vertex. We write $a(V)$ as $(a_0(V), \ldots, a_N(V))$.
4. A successor function, $\mathcal{S}$, which maps each $V \in \nu$ to a map $\mathcal{S}_V: \{1, \ldots, a_0(V)\} \times \ldots \times \{1, \ldots, a_N(V)\} \to \Gamma_V$. This associates each combination of alternatives with a (unique) successor[2]. We generally write $V_{a_0,\ldots,a_N}$ for $\mathcal{S}_V(a_0, \ldots, a_N)$.
5. A payoff function, $u: \tau \to R^N$, for $R \subseteq \Re$ (we generally use $R = \Re$). $u_i(V)$ denotes $u(V)_i$. This function associates a vector of utilities for the players with each terminal vertex.
6. A probabilities function, d, which maps each $V \in \nu$ to a probability distribution, $d_V: \{1, \ldots, a(V)_0\} \to [0, 1]$ (so that $\sum_{i=1}^{a(V)_0} d_V(i) = 1$).

Sometimes, we write $(x_0, (x_1, x_2, \ldots, x_N))$ to denote $(x_0, x_1, \ldots, x_N)$.

The idea is that the vertices of the tree are stages in which each player selects a move, and there is (simultaneously) a chance event (determined by the probability distribution). The outcome is determined by the unique successor corresponding to these moves and this chance event.

Note that we could make the definition allow for any pointed poset (i.e., a partially ordered set with a least element; the root) instead of trees. We can replace such a poset with a tree in which common subtrees are "duplicated" without changing the important theoretical properties of the game. However, it is generally preferable to deal with trees (so that each node represents the sequences of moves used to reach a state and not just the state in question). Algorithms to compute functions on a game that do not use such information can be made more efficient by use of a pointed poset in computations, at least to the extent that computations on one substructure are not repeated.

[2]That is, we assume the map is bijective. This is done for the same reason that we use a tree and not a pointed poset, as discussed immediately following this definition.

We extend the payoff function to all the vertices, by back-induction. To do so, we employ a solution concept for games in strategic form. So we have a function, $\mathcal{U}: R^{a_1 \times \ldots \times a_N \times N} \rightarrow R^N \times ([0,1]^{a_1} \times \ldots \times [0,1]^{a_N})$, for any values of $a_i \in \mathcal{P}$. The idea is to map strategic forms (i.e., payoffs for each strategy profile) into a solution value (a vector) and (mixed) solution strategies for each player. We define $U(G) \equiv \mathcal{U}(G)_1$, i.e., the value of the game. Since we use this concept for individual stages only, it need specify a solution only for games whose strategic forms have no less "structure" than their extensive forms. Specifically, each player has a single information set. We refer to moves at a stage, and strategies at that stage interchangably: strategies refer to strategies in a matrix game whose payoffs are those of the solution to the corresponding subgame. To extend the payoff function to all of Γ we define, for $V \in \nu$,

$$u(V) = U(\{M_j | j \in \{1, \ldots, a_1(V)\} \times \ldots \times \{1, \ldots, a_N(V)\}\}),$$

where

$$M_j = \sum_{i=1}^{a(V)_0} d_V(i) u(V_{i,j}).$$

3.2 A Naive Solution Algorithm

There is a naive back-induction algorithm similar to the naive minimax algorithm for solving N-player, zero-sum games of perfect information. This algorithm uses a procedure that computes $\mathcal{U}$. For simplicity, we do not detail how to return a "tree" of strategies—only the top-level mixed strategies are returned. Note that in all the search algorithms, we can dynamically configure "approximation games" (games in which certain non-terminal nodes are taken to be terminal and given a heuristic estimate of the payoff of the subgame rooted at the node). These are essentially the same as the approximation games that have long been used in computational game theory (discussed in chapter 2). One way to do so is to pass a depth parameter, d, to the search algorithms. If $d = D$ (for D the maximum depth of search), we take V to be terminal and compute a heuristic payoff value. The depth value is incremented at each recursive call (though non-standard measures of depth can be used, like those discussed in section 2.2.4). The idea of searching an approximation game (in this context) is discussed in more detail in section 3.5. For now we note that even when we will search a full game tree, more efficient algorithms can profit by searching approximations of the game tree.

```
Function Naive-Solution(vertex V)
(It returns a value and a set of mixed strategies.)

if V ∈ τ return (u(V), ∅)
else
  let G be an array (of vectors R^N) of dimension a_1(V) × ... × a_N(V)
  for all choices j ∈ {1,...,a_1(V)} × ... × {1,...,a_N(V)}
    G(j) ← 0
    for i = 1 to a(V)_0
      (t, l) ← Naive-Solution(V_{i,j})
      add d_V(i)t to G(j)
  return 𝒰(G)
```

This algorithm will correctly compute the value and next-move of a matrix game as defined above. That is, $u(V) \equiv \text{first}(\text{Naive-Solution}(V))$.

In the rest of this chapter we deal with two-player zero-sum matrix-tree games. Because of their zero-sum nature, we write a payoff value, p in lieu of the 2-dimensional vector $(p, 1-p)$. Until the section on chance play (section 3.10), we also restrict our attention to those without chance alternatives (i.e., those for which every non-terminal vertex V has $a(V)_0 = 1$). Accordingly, we omit the chance element from our notation when presenting algorithms, so that we write V_{a_1,a_2} for $\mathcal{S}(V)(1, a_1, a_2)$ and we write $a(V)$ for $(a(V)_1, a(V)_2)$. It is easy to extend a solution algorithm to deal with chance alternatives in a naive fashion, and efficiently handling chance alternatives is largely orthogonal to efficiently handling player moves. Accordingly, we defer discussion of chance until the end of the chapter.

As we noted in the background chapter, the standard axioms of rationality give a solution concept for such strategic form games. Indeed, to solve a strategic form game with the payoff matrix $A_{m\times n}$, one solves the linear program:

$$\begin{array}{ll} \text{maximize} & w \\ \text{subject to} & \displaystyle\sum_{i=1}^{n} x_i A_{ij} \geq w \quad j = 1 \ldots m \\ & \displaystyle\sum_{i=1}^{n} x_i = 1 \\ & x_i \geq 0 \qquad i = 1 \ldots n \end{array}$$

The objective variable w is the value of the game, x is the solution strategy for player 1, and the dual program gives the solution strategy for player 2. Details on this, and other aspects of linear programs appear below.

It is worth mentioning how a computer can use such solutions to actually participate in a game. If the computer computes a solution in (non-degenerate) mixed strategies, then it must select one of the strategies in the support of the solution in accordance with the probability distribution of the mixed strategy.

3.3 The Extended Scout Algorithms

We present two families of algorithms (parameterized over a number of sub-algorithms or "policies") to efficiently solve two-player, zero-sum, matrix-tree games. We intend these algorithms to achieve a degree of improvement comparable to that obtained by efficient algorithms, like α-β or SCOUT, over the naive method of solving two-player, zero-sum, perfect information games. These two methods rely on bounding the value of a position (possibly indirectly, as in an α-β "deep cutoff") and using such bounds to prune the search tree. The idea is that a given move can be *refuted* by showing it leads to a value for the player no more than that available to him if he selects a different move.

Improvement by Domination: One method of improving searches is to extend the simple notion of refutation to domination (more precisely, to weak domination or duplication). That is, we use bounds on the payoff for a given vertex to establish that certain moves are dominated (hence do not have an impact on the value of the game), without evaluating all the positions in the game rooted at this vertex.

Theorem 3.3.1 *The solution of the substructure of a game induced by deleting a weakly dominated or duplicated pure strategy is also a solution to the original game.*

Proof: We need to establish for each of the (mixed) solution strategies for the substructure, that it ensures its player will obtain at least her minimax value for the game.

Suppose the deleted pure strategy is Y, and it is a strategy for player p. The conditions of the theorem entail that we have a linear combination $a_1X_1 + \ldots + a_kX_k$ ($X_1, \ldots, X_k$ are other pure strategies) such that for each pure strategy, Z, of the opponent, $a_1u_p(X_1, Z) + \ldots + a_ku_p(X_k, Z) \geq u_p(Y, Z)$. Now, suppose this holds for Y and Y is in the support of a solution strategy (of the original game), with probability $b > 0$ of being played. In this case, a new strategy, in which the probability for X_i is increased by ba_i would obtain at least as large a payoff against the opponent's solution. And the payoff can be no larger, because the opponent's solution strategy guarantees him his minimax value for the game against any strategy. Hence, this new strategy is also a solution that yields the same minimax value of the game. So any solution of the induced game has the same value as any solution of the original game, and must also be a solution of the original game. $\square$

An immediate corollary is that we can iteratively remove multiple dominated strategies and the solution to the remaining substructure will be a solution to the game.

Improvement by Verification of Equilibrium: Another method of improving searches is to establish, for each player, that a given subset of moves is the only subset that the player needs to use to obtain her minimax value for the game. To do this, one first solves this substructure (a game consisting only of these moves). Then one verifies that each subset is indeed the only set of moves that needs to be played, i.e., that the solution profile for the subgame is an equilibrium for the whole game. This is done by showing that should either player choose another move, his payoff against the opponent's strategy in the substructure is no better than the value of (the solution to) the substructure, i.e., demonstrating that each strategy is a best reply to the other. This can be done by bounding the value of each move not in the substructure against each move in the support of the opponent's solution strategy.

Theorem 3.3.2 *The solution to the substructure of a game is also a solution to the game if, for each move not in the substructure, the value to its player can be bounded above by the value of the solution to the substructure.*

Proof: Given the fact that the solution profile is an equilibrium of the substructure, the expected value for a player of any of his pure strategies in the substructure must be bounded above by the value of the game to him, v. However, we directly verify that v upper bounds the expected value for that player of any of his pure strategies that are not in the substructure. So for any mixed strategy, its expected value (for the player) against the opponent's substructure solution strategy is upper bounded by v. Hence, for each player their solution strategy for the substructure is a best reply (for the entire game) to the opponent's solution strategy for the substructure. Therefore the profile is an equilibrium for the game, and a valid solution (because any strategy that is part of an equilibrium profile guarantees its player will obtain her minimax value). □

3.3.1 Qualitative Requirements for Evaluation

Unlike in a sequential game, there are some matrices that we cannot efficiently search without the possibility of searching a given cell more than once; to establish a bound on a move generally requires testing it against more than one of the opponents' moves. Hence, even if a bound is met against one move, those subgames that were bounded earlier must be re-searched if it a bound is not met against another. The substructure method mentioned above can perform efficiently without the possibility of re-search by never testing, but always evaluating moves. However if a player has but a single move, this approach would not perform well. In general, the performance of this approach would be degraded in games where there are many "unbalanced" stages, i.e., stages in which the players have highly different numbers of moves available to them.

Because of the possibility of re-search, any algorithm that attempts to use bounds to prune the tree will be much more like SCOUT than α-β. Intuitively, the only case in which using a directional control strategy akin to that of α-β makes sense is the case where only one choice remains for a player. This means that there is little benefit in adopting such an approach.

We employ a test algorithm for use in the dominating and substructure evaluation algorithms. This algorithm attempts to determine whether a given game position can be bounded by a given bound. If the test algorithm returns a bound, then the bound does hold. However, it might not establish that a bound holds, even if it does hold. Occasional failure to establish bounds is unavoidable in an efficient algorithm, because there are many cases where determining that a bound can not be met effectively requires solving the game. Moreover, there is usually no single guaranteed value against which to test because we are often testing a linear combination of several variables against a value, so no single variable has a bound that determines the truth of the overall inequality.

Policies: Note that there are numerous places in the evaluation and test algorithms where we select bounds for which we test. Often, there is a trade-off between choosing tighter bounds and making more recursive calls, or between tightening the bound on one cell and loosening that on another. These are the primary causes for the existence of several places in the algorithms where there are choices for "policy." There are also other places where choices should be made that are dependent on the specific game involved. Static evaluation and move ordering heuristics are good examples of this.

3.3.2 The Test Algorithm

The algorithm Test-$\leq$ tests for upper bounds. We do not present the dual Test-$\geq$ algorithm for testing lower bounds. In the algorithms in this chapter, we use asterisks (*) to mark places where a sub-algorithm needs to be supplied to perform the stated computation. We discuss the sub-algorithms for Test in sections 3.4.1.2 and 3.4.2.3.

```
Function Test−≤(vertex V, upper bound y)
(It returns an upper bound on u(V).)

if V ∈ τ, return u(V)
else
  let (m,n) ≡ a(V)
  loop
    choose* a collection R of rows
    or, if no collection looks promising* return ∞
    let q ≡ |R|
    choose* weights ω for R  s.t. ∑_{i=1}^{q} ω_i = 1, ω_i > 0
    for i = 1 to n
      choose* lower bounds b_{i1},...,b_{iq} s.t. ∑_{j=1}^{q} ω_i b_{ij} = y,
      to be the bounds for the ith column of each member of R
    choose* an enumeration^3, O of {1,...,n} × R
    for k = 1 to nq
      if O(k) is marked, choose* looser bounds for its column
      A_{O(k)} ← Test-≤(V_{O(k)}, b_{O(k)})
      if A_{O(k)} > b_{O(k)}, it failed so exit the inner loop
      and if A_{O(k)} < b_{O(k)}, it returned a tighter bound than required
        mark the next bound in its column for subsequent loosening
    if all tests succeeded
      compute the tightest bound using the bounds in A
        (using a linear program)
      return that bound
```

[3] An enumeration of X means a bijective map from $\{1,\ldots,|X|\}$ to X.

3.3.3 Dominating Evaluation Algorithm

This algorithm attempts to reduce the work involved in evaluating a game by showing that certain moves are dominated, as described earlier. We discuss the sub-algorithms for dominating evaluation in sections 3.4.1.1 and 3.4.2.2.

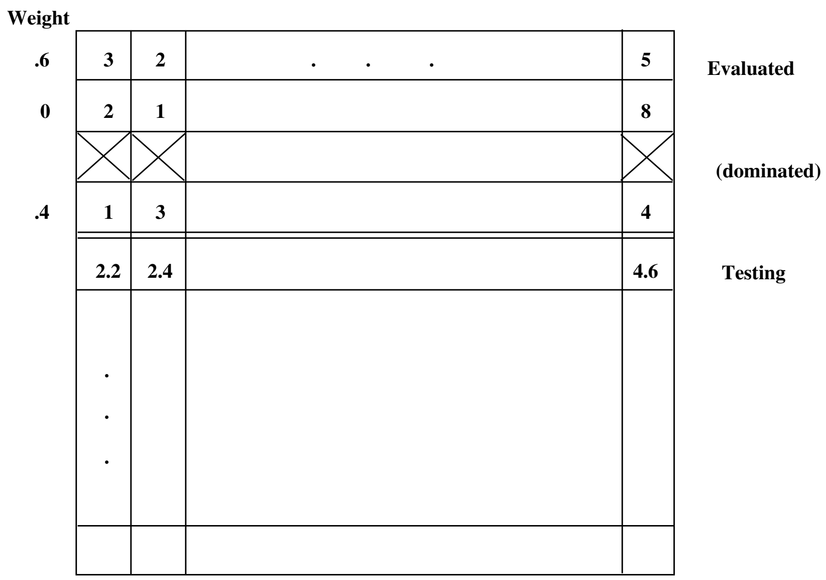


Figure 3-2: The Dominating Evaluation Algorithm

Figure 3-2 illustrates the process of using previously evaluated (undominated) moves to set bounds to test a move.

```
Function Dominating – Eval(vertex V)
(It returns a value and pair of mixed strategies.)

if V ∈ τ return u(V), (Λ, Λ)
else
  let (m, n) ≡ a(V)
  order moves*
  while there are unevaluated cells
    possibly* reorder moves
    select* either a row or column:  it must be the highest
      ordered move for its player that contains an unevaluated
      cell
    if we selected a row i
      choose* upper bounds b_j for v_ij , j = 1..m
      for j = 1 to m
        B ← Test- ≤ (V_i,j , b_j)
        if this failed (B > b_j)
          if we can tighten an earlier bound in the row
            back-track and tighten* earlier bound
          else
            v_ij ← Dominating-Eval(V_i,j)
            if the value does not meet the bound
              exit the inner loop
        else if B < b_j
          loosen some* bounds (i.e., increase b_k for some k > j)
      if we did not exit due to a value failing to meet the bound
         the row is dominated and we eliminate it from
         consideration
      else
         evaluate some* unevaluated cells in the row
         if not all the cells have been evaluated, possibly* set
           new bounds on the other cells to try to show the row
           is dominated (if this is not possible simply evaluate
            any unevaluated cells in the row)
    else
      perform the dual of the above on a column j
   solve the remaining substructure (contained in V)
     (by a linear program)
   return the solution value, and (mixed) strategies
```

Whenever we evaluate a cell, its value is also stored. So, if $V_{i,j}$ is evaluated, v_{ij} is set to Dominating-Eval$(V_{i,j})$.

3.3.4 Substructure Evaluation Algorithm

This algorithm attempts to reduce the work involved in evaluating a game by solving a substructure, and showing that the other moves are not strict best replies to the solution of that substructure. Note that by simply not setting and testing bounds one obtains an algorithm that solves games only by evaluation (which is efficient for "balanced" matrices).

We discuss the sub-algorithms for substructure evaluation in sections 3.4.1.3 and 3.4.2.2.

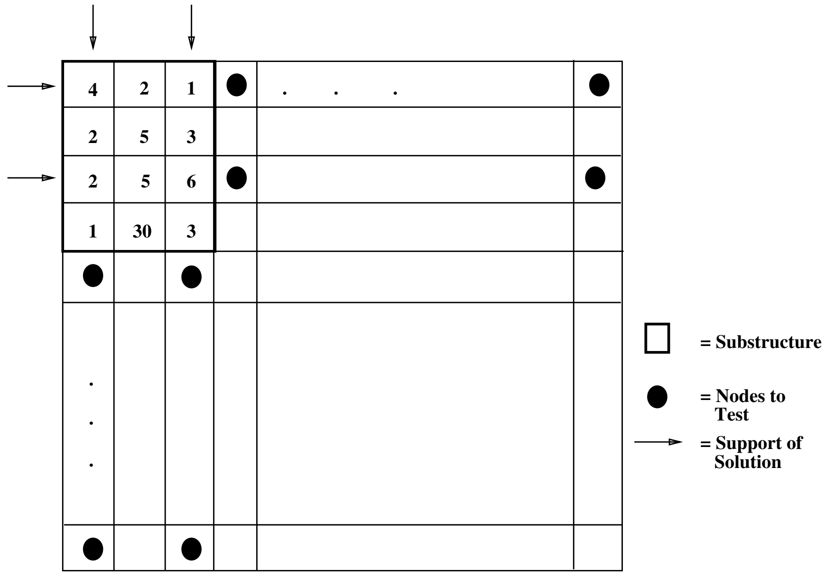


Figure 3-3: The Substructure Evaluation Algorithm

Figure 3.3.4 illustrates the process of evaluation: testing those subgames that are reachable by playing strategies in the support of the solution to the substructure.

```
Function Substructure − Eval(vertex V)
(It returns a value and pair of mixed strategies.)

if V ∈ τ return u(V), (Λ, Λ)
else
  let (m, n) ≡ a(V)
  pick* a k × l substructure
  loop
    for any unevaluated cells (i, j) in the substructure
      store Substructure-Eval(V_{i,j})
    solve (using a linear program) the substructure;
      let its value be v
    if there are no unevaluated cells, return this solution
    order* the moves (of both players) not in the substructure
    loop over each such move, in the order just imposed
      if the first player is making the move
        Test the move to show that he cannot obtain more than v
          against the second player's strategy.  To do this:
        Set* upper bounds on this move against each of move in the
          support of player 2's solution such that the weighted
          sum of bounds is v
        Order* the bounds by decreasing likelihood of failure.
        Test-≤ the resulting subgames against each bound.
        if a bound is tighter than needed, loosen* the
          remaining bounds
        if any bound fails, possibly* set and test new bounds,
          otherwise evaluate the cell and if the value meets the
          bound, proceed; otherwise evaluate all the cells (for
          this move), and if the value of the move against player
          2's strategy is more than v, add it to the substructure,
          and restart the loop
      else (for a move of the second player)
        perform the dual of the above
  return the solution to the substructure
```

In this algorithm, we store previous test and evaluation information. Any moves (of either player) that are fully evaluated against the support of the opponent's solution strategy are ordered to be tested first. In this case, testing requires only taking a dot product (of the weights and values) and comparing this against the game value.

3.3.5 Comments on the Algorithms

Before discussing the details of policy choices, we want to discuss a few issues about the computations in these algorithms, and expectations about the games in question.

Note that these algorithms all use depth-first search, allowing for minimal space overhead and quick computation in actual use.

If we use floating point computations, then there can be numerical error (chiefly in solving linear programs). If computations are done in floating point, then the evaluation solutions found will be only Nash ε-equilibria[4] (ε is determined by the computational accuracy of the system in use). Even if the subgames of a game have integer payoffs, the value of the game can be a non-integral rational number. It is possible to guarantee the return of true equilibria (for payoffs that are arbitrary rationals) by use of (exact) rational numbers instead of floating point. Naturally, doing this will result in somewhat slower computation.

Another issue is the solution of linear programs. The oldest and most commonly used algorithm to do this is the simplex method[Dan63]. This has a very good apparent average case run time of $O(m \log n)$, for an $m \times n$ input matrix (i.e., m constraints in n variables).[5] However, the simplex method takes exponential time in the worst case.

A theoretically preferable algorithm is due to Karmarkar([Kar84]); this has a polynomial worst case performance: $O(n^{3.5}L)$, for input that is L bits long and encodes an $m \times n$ matrix. If the length of a number is constant, then $L = O(mn)$. Ye ([Ye91]) has presented an improved version, with a worst case performance of $O(n^3 L)$. These algorithms have average case performances that are quite competitive with that of simplex.

Both linear programming algorithms allow the simultaneous solution of a primal and dual problem, so it is best to solve only the problem that is expected to take less time.

Linear programming algorithms require variables to be positive. As such, if our payoff values are not always positive, we need to rescale payoff values (by a positive affine transformation) to ensure that all the payoffs are positive (so that the objective value, w, is positive). Naturally, the solution value needs to be scaled back to the original utility scale. This rescaling can be combined with a direct method of transposition: if the matrix game A has solution value v and strategies (x, y), the game with payoff matrix $-A^T$ has value $-v$ and solutions (y, x). We can do this to allow for faster solution of programs; the simplex method is faster for $n < m$ when using an $m \times n$ matrix.

The evaluation algorithms given above both require the game to have certain characteristics in order for them to reduce the time taken below that of naive search. Substructure evaluation requires that the solution strategies' support is small relative to the total number of moves (though even if half the moves available are in the solution, it can still recursively evaluate as little as a quarter of the subgames). We believe that this is very reasonable: in the simulation we used, saddle points were quite common, and solution strategies almost always had support cardinality less than four (the total number of moves at any stage was between 10 and 200). Dominating evaluation requires many moves to be dominated; this implies that the solution support has small cardinality (though not vice-versa). Practical experience again suggests that this is reasonable.

3.4 Policies

We now discuss the various policies that might be used. Our discussion is somewhat informal and speculative in places; we rely on practical experience from the case study and qualitative assessments to guide our considerations. However, the ultimate objective in setting policies is to obtain good performance, so we mainly judge the collection of policy choices by the results obtained in the case study (discussed in section 3.9.3). In many cases, we believe

[4] See Fudenberg & Tirole[FT91] p.33

[5] Chvátal[Chv83] p. 46

that improvement of the policies can be made, but that the resulting improvement would be of a second-order nature.

For both the test and evaluate algorithms, we try to pick bounds that maximize the probability that all the bounds will be met. This is a slight oversimplification; an optimal policy would prefer a slightly lower probability of meeting bounds if it meant a significantly smaller expected cost of computation. For example, it would be better to isolate a single bound as likely to fail (to reduce the expected amount of computation before a failure), if it did not lower the overall probability of success "too much." We discuss the use of metareasoning as it pertains to this and other aspects of the policies below.

To allow us to estimate probabilities of success and to order moves, we employ game-specific heuristics to estimate the expected value and deviation of the subgames. In our various computations, we treat the value of each subgame as a normally distributed random variable. While it is possible to simply use constant time heuristics to assign these values, we believe it is superior to obtain the relevant information by a shallower (recursive) evaluation. Use of constant time heuristics is traditional for move ordering in 2ZPI games. One can also use shallower evaluation in a manner similar to the method we advocate for these games, but it is not generally done because it is assumed to reduce speed too much. For simultaneous play games, the need for more precise information to set bounds (instead of just ordering moves) leads us to regard the extra time spent as worthwhile.

Let us specify how to perform this shallower evaluation. If we are searching an approximation game and we will search d levels below some vertex, we back up information from searching $\pi(d)$ levels below that vertex, where $\pi(d) < d$, and π is monotone non-decreasing. If $\pi(d) \leq 0$, then constant time heuristics are used instead. To obtain this information, we generally perform an evaluation of the game itself, searching to $\pi(d)$ levels below it, and we also evaluate certain additional subgames in this manner.

Two possible families of discount functions are $\pi(d) = d - r$, for r a positive integer and $\pi(d) = \lfloor yd \rfloor$ for $0 < y < 1$. We provide an analysis of the asymptotic cost for these π functions in section 3.8.

Note that any game has a finite depth, so this approach is not restricted to approximation games. However, if the game tree is not fairly uniform in depth, one should attempt to search more shallowly everywhere, in order to ensure this search does not take a long time. Alternatively, one can simply search such games using iterative deepening and use a transposition table to store the (exact) evaluations of shallow subtrees; these techniques are discussed in section 3.5.

Our general approach is to use the value of the search as the expected value, and to estimate deviations based on static factors. We note that, in general, the random variable of the value of a 2-player zero-sum strategic form game, whose subgames' values are independent normal random variables, is not a normal variable. For example, in a $m \times 1$ game of independent standard normals, it is the maximum of m normals, which is not a normal. However, using a normal approximation should work reasonably well in the worst cases and quite well for games in which both players have similar numbers of "high-quality" moves. Note that taking the expected value of a game to be value of the expected values of the subgames is also an approximation; the expected value of an $m \times 1$ game of independent standard normals is not 0. We discuss more sophisticated approaches to setting means and deviations in section 3.6.1.

If one is using simple heuristic estimates of the expected value and deviation of individual moves (as we do when not doing a shallow search), one can use various simple functions to combine expected values and deviations of single moves to obtain these values for the subgame reached by playing individual moves. Any procedure that uses such information should be optimized for the simple, regular, nature of the information. For example, when optimizing functions of these values, drastic simplifications are available. The most natural combination functions to use are addition for expected values and multiplication for deviations, so $\mu_{ij} = \mu_i^1 + \mu_j^2$ and $\sigma_{ij} = \sigma_i^1 \sigma_j^2$ (the superscripts indicate the player whose move is being estimated). One could also use a static estimate of covariance to modify means and deviations, but it is probably better not to add this complexity, since the motivation for not even doing a static evaluation of the successor positions is to minimize search overhead.

An additional idea is to modify the mean and deviation estimates (at both twig and deep nodes) for subgames that have not been evaluated or tested. This modification would occur in response to unexpected evaluation and test results encountered during the search. The basic idea is to increase the deviation in subgames that result from the moves that lead to the unexpected result, and to alter the expected value in the direction of unexpected change. If a move leads to unexpected values more than once, this effect should be stronger. However, it is not clear that this approach will improve performance (since the unexpected result might just be a result of a specific interaction). While this approach is worthy of further consideration and analysis, we do not assume it is being used in the remainder of this chapter.

We distinguish between deep nodes and twig nodes. Deep nodes have non-terminal successor nodes and twig nodes have only terminal successor nodes. We make this distinction because testing a leaf node is basically the same as evaluating the node. One could optimize the heuristic evaluator to reduce the work required to test against a bound (by stopping once the value is determined to be above or below a certain number). We expect that few heuristic functions could be tested against a bound significantly faster than simply being evaluated.[6] So, at a twig level, it seems to be unwise to try to test nodes instead of simply evaluating them. This necessitates the use of different policies at twig nodes.

3.4.1 Deep Node Policies

We first present policies for the dominating evaluation algorithm, then for the test procedure and finally for the substructure algorithm.

3.4.1.1 Dominating Evaluation Policies

Move Ordering: Moves are ordered by evaluating the game rooted at the node to depth $\pi(d)$. We let μ_{ij} and σ_{ij} be the expected value and deviation, respectively, of the subgame reached by the play of player 1's ith move, and player 2's jth move. Some of these values will not be computed by the (shallow) evaluation of the game (though any cell reachable by playing moves that are both in the support of the solution strategies has its mean and deviation computed by the evaluation). However, not all the mean and deviation values are

[6] If a heuristic function is of this kind, it is preferable to use non-speculative forward pruning, as described in section 3.5.4.

used. If these values are required (but not already known) for some cell, they are computed by searching the given subgame to depth $\pi(d) - 1$.

Moves in the support of the solution are ordered in decreasing probability of play. Other moves are ordered by their expected value against the opponent's play (and after all moves in the support). That is, for ω_i the probability that the ith move of player p will be played, we order those moves j not in the support of the solution strategy of the other player by $\sum_{i=1}^{m} \omega_i \mu_{ij}$. Naturally, player 1's moves are ranked in decreasing order, and player 2's in increasing order. Note that this policy does not use the heuristic values for those cells that are reached by play of moves that are both not in the support of their player's solution strategy.

If $\pi(d) = 0$, we order moves by a heuristic estimate of the quality of the move, as described under twig node move ordering (section 3.4.2.1).

We do not use reordering of moves since it requires us to modify the estimates of means and deviations in response to search information. As we mentioned earlier, we are assuming this type of modification is not being performed.

Selecting Unevaluated Moves: We propose to work only on rows or columns when evaluating a given subgame (i.e., we either select just moves of player 1 or just moves of player 2). If there are more columns than rows we use columns, otherwise we use rows (ties could be broken by some less arbitrary scheme). The best case is better for this policy selection (of working with the larger number of moves), and on average we believe that less work needs to be done per move, and the proportion of moves dominated is likely to be increased.

We do not believe that switching between rows and columns is helpful. The reason is that the effort spent in evaluating moves of one player cannot be used when testing moves of the other. Because having more moves of one player evaluated improves the ability to dominate subsequent moves, we believe that selecting the moves of only one player will result in having more moves be successfully shown to be dominated, instead of evaluated.

Setting Bounds: We describe the case of testing a row (i.e., a move of the player 1) in this and subsequent sections on bounds. The behavior for moves of the player 2 is the dual of this.

Let m, n be the number of moves available to players 1 and 2 (respectively), and $n_1, \ldots, n_r$ be the previously evaluated (undominated) rows, and k be the row currently being tested. We use a linear program to set upper bounds b_j for each position to maximize the minimum of $(b_j - \mu_{kj})/\sigma_{kj}$. Maximizing the probability of success on the least likely bound is expected to be a close approximation to setting bounds that maximize the overall probability of success.

The b_j are a linear combination of the (evaluated) predecessors, and the linear weights are determined by the linear program, so $b_j = \sum_{i=1}^{r} x_i v_{n_i j}$, where the weights x_i are variables. So the linear program used is:

$$\begin{array}{ll} \text{Maximize} & w \\ \text{Subject to:} & \sum_{i=1}^{r} x_i v_{n_i j} - \mu_{kj} \geq \sigma_{kj} w \quad j = 1 \ldots m \end{array}$$

$$\sum_{i=1}^{r} x_i = 1$$
$$x_i \geq 0 \qquad i = 1 \ldots r$$

Note that some of the b_j can be such that $(b_j - \mu_{kj})/\sigma_{kj} > w$ (i.e., they are likelier to be met than the least likely bound). We order bounds for testing in increasing order of $\frac{b_j - \mu_{kj}}{\sigma_{kj}}$ (if $\sigma_{kj} = 0$, we do not perform a test, and we do not order the bound since it is certain to be met).

However, we might want to test for domination, even if the maximum value for w is negative, so we need to maximize $w + Z$. In this, we need Z to be sufficiently large that $w + Z$ must be positive. Accordingly, we take

$$Z = \max_{j \text{ s.t. } \sigma_{kj} \neq 0} \frac{\mu_{kj}}{\sigma_{kj}}$$

For $\sigma_{kj} = 0$, we have an exact value, so if the program has no solution, then it is impossible for the move to be dominated.

We also employ a parameter W such that if $w < W$, we assume that success is sufficiently unlikely that we do not test. This threshold value W should be set to minimize computation. It can be set statically by a game-specific heuristic function, or dynamically to approximate the point where

$$\frac{P_{\text{success}}}{P_{\text{failure}}} < \frac{E[\text{cost}_{\text{failure}}]}{E[\text{cost}_{\text{success}} - \text{cost}_{\text{failure}}]}$$

Naturally, P_{success} is a function of W, and the expected cost of computation requires a model incorporating information about future search. A discussion of a metareasoning system containing such models is discussed in section 3.6.3. However, we will assume a game-specific heuristic function to determine W.

If w falls below the threshold W, we evaluate the subgame corresponding to the "most problematic" bound b_j (this is the one that was ordered first, as the most likely to fail). We can proceed to rerun the linear program on the remaining cells, using the evaluated value for μ_{kj} and $\sigma_{kj} = 0$. However, if μ_{kj} is sufficiently low (below $b_j - \phi W$, for a parameter ϕ such that $0 \leq \phi \leq 1$), we simply evaluate all the remaining cells, in preference to solving another linear program. If the linear program has no solution, we evaluate all the cells in the row (since the move cannot be dominated). If it does have a solution, and $w \geq W$, we start testing, else we pick another "most problematic" subgame and evaluate that.

On a parallel machine, we might evaluate all the tightly bounded (problematic) cells at once. An alternative is simply to evaluate all the cells at this point, to increase the amount of parallelism.

An alternative method is to use a non-linear optimization method to optimize

$$\prod_{i=1}^{r} \Phi\left(\frac{b_i - \mu_{ki}}{\sigma_{ki}}\right)$$

Here, $\Phi(x) \equiv \int_{-\infty}^{x} e^{-t^2/2}/\sqrt{2\pi}\, dt$ is the cumulative distribution function for a standard normal. We expect that the linear programming approximation is sufficiently good that it is not worth the extra time required to use a nonlinear procedure.

Loosening Bounds: The bound, b_j must have been "tight," i.e., $(b_j - \mu_{jk})/\sigma_{jk} = w$ (otherwise we will not obtain better bounds). We might also require it be improved by a certain margin. If we do loosen bounds, we simply re-compute the bounds linear program, but using μ_{ik} as the established value (by a successful test or evaluation) for any subgames already tested, and using $\sigma_{ik} = 0$ for these subgames (in this way, we will not need to retest subgames).

Failure To Meet Bounds: If a bound cannot be met, we evaluate the subgame, and proceed just as we did in the case where $w < W$. That is, we might solve a new linear program and act depending on the value of $w < W$, or we might just evaluate the remaining subgames for the move. If we have evaluated or successfully tested any subgames, we set μ_{ik} to be the value (or established bound) and $\sigma_{ik} = 0$, just as when loosening bounds.

A more ambitious alternative to this policy is to heuristically alter the expected value and deviations of the cells in the row. The deviation of a cell whose bound failed would be increased, while the expected value is raised (for a failed upper bound). The cells that were bounded would have deviation decrease and expected value decrease. Naturally, these random variables are not very well approximated by normal variables; it would be preferable to use a more general class of variables for this task (though normal approximations might still suffice). This issue is further discussed in section 3.6.2.

To set new bounds one should try to optimize expected cost, which would include favoring setting bounds that are no stronger than those that have already been met (as there is no search required for these cells). We adopt the most conservative policy of never testing the same subgame twice.

In any case, we believe that our policies set initial bounds sufficiently well that retesting will tend to cost more than it is worth; generally bounds failure means that the move is not dominated, and attempting to retest is likely to be more expensive than merely evaluating. In the case study, we tabulated statistics on how often a move was actually dominated (by the previous moves), but was not shown to be dominated when using this policy. The results (in section 3.9.3) support our belief that retesting is not necessary.

3.4.1.2 Testing Policy

The only sub-algorithms needed for the test algorithm are those used to select bounds or determine that no testing should be done.

However, there is more complexity in setting bounds for tests. We must do the following:

- Determine the most advantageous collection (if any) of rows/columns to use.
- Determine the linear combination of weights to assign to each of these.
- Determine the individual bounds per column/row.

We address all of these questions for the case of determining an upper bound; the other case is analogous. We present policies in a bottom-up manner (handling the last question first).

Picking Bounds for Individual Rows: We consider testing an upper bound of y. We take the values of the q subgames in the row to be the independent random variables $X_i \sim N(\mu_i, \sigma_i^2)$, where $N(\mu, \sigma^2)$ indicates a random variable with normal distribution having mean μ and variance σ^2. At this point, we already have a vector of column weights ω s.t. $\sum_{i=1}^q \omega_i = 1$.

Our task is to pick bounds ξ_i s.t. $\sum_{i=1}^q \omega_i \xi_i \geq y$ in such a way as to maximize $P\{X_1 \leq \xi_1 \wedge \ldots \wedge X_q \leq \xi_q\}$. We assume that $\sum_{i=1}^q \omega_i \xi_i = y$, because if $\sum_{i=1}^q \omega_i \xi_i > y$, we can loosen (increase) one of the ξ_i bounds without decreasing the probability of success. If $\sigma_i = 0$, we simply set $\xi_i = \mu_i$ (the bound is established without testing), and we solve a similar problem for the remaining subgames for which $\sigma_i \neq 0$. For example, if $J = \{i \in \{1, \ldots, q\} \,|\, \sigma_i = 0\}$, then we test whether $\sum_{i \in \{1,\ldots,q\}-J} \omega_i X_i$ is bounded above by $y - \sum_{i \in J} \omega_i \mu_i$. Naturally, if $J = \{1, \ldots, q\}$, we do not need to compute any bounds (or perform any tests); we can simply compute whether the bound is met. Now, let us consider the problem where, for each i, $\sigma_i \neq 0$.

And, for F the probability of success,

$$\begin{aligned} F(\xi_1, \ldots, \xi_q) &= P\{X_1 \leq \xi_1 \wedge \ldots \wedge X_q \leq \xi_q\} \\ &= \prod_1^q P\{X_i \leq \xi_i\} \\ &= \prod_1^q \Phi\Big(\frac{\xi_i - \mu_i}{\sigma_i}\Big) \end{aligned}$$

Let P be the maximum value for F, given any set of bounds $\{\xi_i\}$. We approximate $P \approx P_0^q$, where P_0 is the probability given by taking each variable to be at the same point on a standard normal. This means setting $\frac{\xi_1 - \mu_1}{\sigma_1} = \ldots = \frac{\xi_q - \mu_q}{\sigma_q}$. So $\xi_i = \mu_i + \sigma_i \frac{\xi_1 - \mu_1}{\sigma_1}$, for $i > 1$. Hence, for

$$S \equiv \frac{\sum_{i=2}^q \omega_i \sigma_i}{\sigma_1} \text{ and } M \equiv \sum_{i=2}^q \omega_i \mu_i,$$

we have,

$$(\omega_1 + S)\xi_1 - \mu_1 S + M = y$$

So we obtain,

$$\sigma_1 = \frac{y - M + S\mu_1}{1 + S}$$

Therefore,

$$P_0 = \Phi\Bigg(\frac{y - \sum_{i=1}^q \omega_i \mu_i}{\sum_{i=1}^q \omega_i \sigma_i}\Bigg)$$

and we use

$$\xi_i = \sigma_i \frac{y - \sum_{i=1}^q \omega_i \mu i}{\sum_{i=1}^q \omega_i \sigma_i} + \mu_i$$

as a first approximation. This approximation should work quite well except when the probability of success is nearly 0 (when there is no point in proceeding further) or nearly 1 (in which case it will still indicate a high probability). However, if ω_i is small, this approximation becomes quite poor, as loosening such bounds requires only a slight tightening of other bounds.[7]

We take this approximation as the starting point for a numeric optimization procedure (we employ a conjugate gradient method—the Polack-Ribiere method), doing a fairly coarse (within .1% of the expected optimum) optimization. To do this, we work with a function of the first $q-1$ variables (though any $q-1$ variables would do). We use table lookup to obtain values for $N_{0,1}, \Phi$ (where $N_{0,1}$ is the density of a standard normal variable). To compute the gradient, we use:

$$\frac{\partial F(\xi)}{\partial \xi_j} = F(\xi)\left[\frac{N_{0,1}\left(\frac{\xi_j-\mu_j}{\sigma_j}\right)}{\sigma_j \Phi\left(\frac{\xi_j-\mu_j}{\sigma_j}\right)} - \frac{\omega_j N_{0,1}\left(\frac{\xi_q-\mu_q}{\sigma_q}\right)}{\sigma_q \Phi\left(\frac{\xi_q-\mu_q}{\sigma_q}\right)}\right]$$

We could use this first approximation in estimating the probability of succeeding when testing a combination of rows, and we use essentially this approximation when picking bounds to test in dominating evaluation.

This method of setting bounds is also employed for setting bounds against the opponent's strategy in the substructure evaluation algorithm. In that case, the column collection and weights are given by the solution to the substructure, of course.

If a bound was established using a tighter value than required (so it was marked), we can loosen the subsequent bounds for the row by solving a new bounds equation for the untested moves. For this, we handle subgames that have been tested in the same way we handle those that have a zero deviation (i.e., we take the value to be the bound that was established and the deviation to be zero).

Choosing Columns and Their Linear Weights: The test procedure has two major requirements: to operate inexpensively (it should be significantly less costly than evaluation) and to succeed in establishing those bounds that do hold. Naturally there is some conflict between these objectives. To balance these needs, we believe that an almost ideal solution for choosing the column collection and the linear weights for these columns would be to maximize the probability of overall success minus a penalty function $\mathcal{Y}$ that is monotone increasing in the number of columns to be tested.

There are two reasons to avoid using large column collections. One is that large collections increase the overall cost to successfully establish a bound. The other is that the opportunity cost of testing a linear combination of all the columns in a given collection includes any set of tests that tests as many subpositions. In particular, one can perform tests on several small sets (even singletons) instead of one large set. For example, if the probability of having a test of column i succeed (by itself) is p_i, then the first two columns should not be tested as a combination unless the probability of success exceeds $1-(1-p_1)(1-p_2)$ (the probability of either of the two succeeding).

Accordingly, $\mathcal{Y}$ needs to be made sufficiently large that both these factors are incorporated. Making $\mathcal{Y}$ sufficiently large that it results in reducing the cost of successful tests

[7]We do not expect to actually set bounds for collections where ω_i is small—we describe why when we consider how to choose column collections and weights.

seems to be largely a matter of setting a value that seems to minimize overall cost. To account for the opportunity cost of testing columns, an additional penalty should be added. At a minimum, we should have $\mathcal{Y}(n) \geq \sum_{i=2}^{n} q_i$, where q_k is the kth largest probability of success for a single column (i.e., of the values for p_i). This suggests that $\mathcal{Y}$ be a sum of a basic monotone function and a function of the individual column probabilities. For example, one might use

$$\mathcal{Y}(n) = \max\left(\sum_{i=2}^{n} q_i, r(n-1) + s\sum_{i=2}^{n} q_i\right)$$

where r, s are parameters in the range $(0, 1)$, which are set to obtain good performance. We also discuss issues of performance when we discuss our retesting policy below.

Before discussing approaches to computing column collections, we make a few qualitative observations about what kind of column collections one would expect to be optimal. The most important is the idea that the combination ought to be *essential*: if any proper subset is nearly as likely to succeed as the collection, then this subset should be tested instead. It seems that a collection is essential only if its members are highly compatible, in the sense that for each column there is a move that would threaten to raise the value above the lower bound, and the column is needed to combat that threat. As we have noted before, we expect that the games we can handle effectively will have small sets of support for solution strategies, so we expect that any essential collection will be small.

It seems to be impractical to directly maximize the probability of success with the penalty function subtracted, since the penalty function makes the objective function have discontinuities at the point where weights become zero. There are two distinct approaches to this problem. One approach is to optimize a continuous approximation to the actual objective function. After performing this optimization, one can then attempt to use this approximation as a basis for "reduction." The other approach is to employ a combinatorial optimization technique to select a collection first, then optimize the probability for that collection.

Optimizing a Continuous Approximation: We consider the first approach, with the probability of success (without subtracting $\mathcal{Y}$) as our approximation function. To implement this approximation one needs to solve the following non-polynomial constrained optimization problem:

$$\begin{array}{lll} \text{Maximize} & \prod_{i=1}^{k}\prod_{j=1}^{l}\Phi\left(\frac{b_{ij}-\mu_{ij}}{\sigma_{ij}}\right) & \\ \text{Subject to:} & \sum_{i=1}^{k}\omega_i b_{ij} \geq y & j = 1\ldots l \\ & \sum_{i=1}^{k}\omega_i = 1 & \\ & \omega_i \geq 0 & i = 1\ldots k \end{array}$$

In this problem, we are examining a $k \times l$ substructure of an $m \times n$ game. Later in this section we discuss why one might wish to examine only a substructure, and how one might select the substructure.

To perform this constrained optimization, one can use the approach of adding a continuous penalty function (not the same as $\mathcal{Y}$) that grows rapidly as the weights leave the appropriate region (i.e., if $\omega_i < 0$ or $\sum_{i=1}^{k} \omega_i > 1$). A description of how to define the appropriate penalty function appears in various texts on optimization (e.g., [Bel70]). The constraints on the bounds can be enforced by a similar approach, or by treating one bound in each row as a dependent variable (as we did when setting bounds for individual rows).

If the solution is not "sufficiently close" to being within the appropriate region, one can restart the optimization procedure using the original solution, but increasing the penalty weights. However, we generally expect that for appropriately large initial penalty values, the solution will be close to lying within the desired region. If the solution has weights that are negative, we simply set these to zero (i.e., these columns will not be in the collection). After this, we normalize the weights, so that their sum is one (keeping a constant proportion between weights). We also recompute the individual bounds using the old bounds as the initial value.

Having performed this procedure, we then begin a process of reduction. This process removes columns that do not contribute significantly to the probability of success, thereby lowering the size of the column collection. The goal is to remove columns with a small weight that offer only a second order benefit, i.e., we try to determine an "essential" subset of the collection.

We propose to reduce by iteratively removing a column and redistributing the weight from the removed column to the other columns in proportion to their existing weight. Naturally, we will not remove a column with a weight of 1. Let us consider an example in which we start with three columns, with linear weights of .5, .3 and .2. If we remove the third column (with a weight of .2), then the other columns will have new weights of .625 and .375. After adjusting weights, we will need to recompute the bounds on individual rows.

To decide which column should be removed in this fashion, we can adopt a greedy approach of removing the column that would maximize the objective function (including $\mathcal{Y}$). An alternative is simply to remove the column with the smallest weight. We also have a choice as to how to stop the process of reduction. One approach is to stop once the new collection has a lower value for the objective function. An alternative is to proceed until there is only a doubleton left. In this case, one stores the best collection (and its weights and bounds). We expect that any choice for these policies will give roughly the same benefit. As such we suggest simply removing the column with the smallest weight (it reduces the amount of computation required), and reducing until only a doubleton remains.

For the policy of just removing the column with the smallest weight, we might also look at all three doubleton subsets once the collection size is reduced to size three, since it is relatively inexpensive to do this, and ought to improve performance.

If the procedure never finds a collection that is better than the most likely single column, it has failed. In this case a column with maximal probability of success should be used instead.

After arriving at a best collection, it might be helpful to reset the weights for the collection by optimizing the probability of success (just as was done before).

This procedure might not determine a global maximum for the objective function (or even the approximation thereto). To handle this concern, we can perform several optimizations, using different initial values for the weights. This allows us to pick the highest valued solution from the results of optimization (and the initial seed weights). We discuss several possible initial values below, since we believe that these are sufficiently good approximations to the solution to be considered as a possible approach on their own. In any case, the objective of this procedure is to produce good bounds, not optimal bounds, and we expect that the procedure will produce good bounds. The biggest concern with optimizing a continuous approximation is whether it produces a sufficient improvement over simple heuristic approximations to justify the time required to use it.

Combinatorial Approaches: Let us consider the possibilities for using combinatorial optimization. Naturally, computing the maximum probability weights for any given combination of rows will tend to be quite slow. One approach would be to use an exact or approximate optimization of the probability of success (likely to be quite slow). Another would be to use a fairly coarse mesh of weights and include these in the space of combinations (and pick exact weights by optimizing the exact probability after determining the column collection).

We believe that greedy approaches, like hill-climbing or greatest-descent least-ascent will not prove very effective. As we noted before, we expect that only essential collections will be optimal, and this entails that the members of the collection will be highly compatible. However, a greedy technique would attempt to add to an already good subset, which does not seem like a promising method for determining essential collections.

Accordingly, the only methods that appear reasonable for this kind of combinatorial optimization are simulated annealing and genetic algorithms. While these approaches are worthy of further consideration, we believe that the relatively large cost of this kind of combinatorial optimization would not sufficiently improve performance to make either of them worth using. If one were to employ such methods, then it would be best to use them only for tests of deep trees with bounds that are not easily met. This case is one in which it is easy to set poor bounds, and the cost of doing so is quite large.

Simple Approximations: Let us consider less ambitious but more easily computed approaches to this problem. These are basically a set of heuristic approaches; quickly computed approximate solutions. Naturally having good approximations will greatly improve either of the aforementioned optimization approaches, by allowing the approximation to be used as an initial value for the optimization.

The simplest approximation is simply to select the single column with maximal probability of success. If its probability of success, q_1, is at least $1 - \mathcal{Y}(2)$, then this singleton is guaranteed to be the best collection (we assume $\mathcal{Y}(1) = 0$; if not then the value against which to compare is $1 + \mathcal{Y}(1) - \mathcal{Y}(2)$). For the function of $\mathcal{Y}$ we described above, we need the probabilities for these columns in any event. Because of the requirement for a great deal of complimentarity to favor the use of a combination of more than one column, we believe that in many situations testing single columns will be the best choice. Also, whenever the value for the game is expected to easily meet the bound and a single move in the solution has a very good probability of successfully establishing the bound (for any subgame that has been evaluated), it seems to be a good idea to not bother performing additional shallow evaluations, but simply to test this single move. If this fails, then one ought to select the next collection more carefully.

Another simple idea is based on the fact that the solution strategy (for a search to depth $\pi(d)$) for player 2 is the (mixed) strategy that is expected to be strongest. So we believe that in most cases, picking a subset of the solution strategy will produce good bounds. This can fail in the presence of unusual deviations or a mixed strategy with a smaller support that has a lower expected value against the opponent's solution, but whose moves are more "consistent" in response to opponent play. The solution strategy's support and weights should be improved by employing the reduction approach described above.

If the game has deviations that vary a great amount, we might use an approximation that is related to using the solution strategy. In this case, we maximize the least likely bound's probability of success. This is also something of an extension of the approximation we computed for a single row. Also, we approximate the probability of success for a given row by P_0. To obtain this approximation we can solve the quadratic program

$$\begin{array}{lll} \text{Maximize} & w & \\ \text{Subject to:} & \sum_{i=1}^{k} \omega_i \mu_{ij} - y - w \sum_{i=1}^{k} \omega_i \sigma_{ij} \geq 0 & j = 1 \ldots n \\ & \sum_{i=1}^{n} \omega_i = 1 & \\ & \omega_i \geq 0 & i = 1 \ldots n \end{array}$$

We would actually maximize $w + Z$, picking $Z > 0$ so that $w > 0$.

Another idea for generating an approximation is to try to quantify the concept of compatibility between columns. However, unless this measurement function were very simple, we would have to solve another difficult combinatorial optimization problem.

One useful heuristic for any optimization approach is to limit our computations to a $k \times l$ substructure of moves that are expected to have an impact on the outcome of optimization. This has the further benefit of reducing the number of subgames that must be evaluated using a shallow search.

One criterion for selecting the moves in the subgame is to include only those moves whose expected value (against their player's opponent's solution strategy) is within a certain threshold proportion of the value of the game. This has the advantage that the only subgames that need to be shallowly evaluated are those that are reachable when one player plays his solution strategy.

A more accurate criterion is based on the probability of subgames meeting the upper bound. A move of player 1 is selected if all of its subgames' probabilities of not meeting the bound are within a threshold proportion of the most likely subgame's probability. A move of player 2 is selected if all of its subgames' probability of success are within a threshold proportion of the maximum of the minimum subgame probabilities in a given column.

Retesting and Deciding When to Fail: These policies are rather similar to the approach stated above for dominating evaluation. We have a threshold value W so that any collection tested must have a probability of success that is at least this large. We also believe that using a fixed limit on the effort allowed is a good idea; setting a limit on the number of columns that can be tested appears to be a good idea. One might set the thresholds and limits statically or dynamically, based on a metareasoning model of computation (see section 3.6.3).

For retesting, we need to update the means and deviations for random variables, and recompute the value of the approximations or various solutions to optimizations seeded with previously optimal values. We described how to update these values in section 3.4.1.1. We can immediately discard any solution that proposes establishing a bound that is at least as tight as one that already failed.

3.4.1.3 Substructure Evaluation Policies

Initial Substructure Selection: Our initial substructure consists of one move for each player. The move used is the likeliest move in the solution strategy of the approximation game evaluated by a search to depth $\pi(d)$ (ties can be broken arbitrarily).

We could make the initial substructure consist of the support of the solution to the depth $\pi(d)$ approximation of the game. However, we believe that starting with a 1×1 substructure is the best approach (to reduce the number evaluations made initially).

Move Ordering: In substructure evaluation, we believe that it is best to order moves in decreasing probability of refuting the current strategy of the opponent (i.e., that obtained in solving the current substructure). This allows us to reduce computation, since if the tests will fail, it is best for them to fail early. There is a slight computational cost to not using a static ordering. This stems from the requirement that additional cells need to be shallowly evaluated whenever a move is placed into the substructure (those cells reached by playing the newly placed move against any moves not in the substructure). However we expect that the improved likelihood of early refutation is worth this slight cost. This cost can be ameliorated by ignoring any moves that have a very poor expected value (for their player) against their opponent's solution in the depth $\pi(d)$ game. These would simply retain their low ranking without requiring additional searches of cells reached by playing these moves.

Setting Bounds: These bounds are set like those used in Test-$\leq$ for testing a row, given a linear combination of column weights. We set bounds to maximize the probability of success, as described in section 3.4.1.2. Naturally, if some of the subgames have already been evaluated, we set their mean to be the evaluated value, and their deviation to be zero. If we set bounds that have already been met, we do not test their value. If we previously established a tighter bound, we compute new bounds on the remaining moves (just as if the subgame had been evaluated and had that bound).

As with the bounds in dominating evaluation, we can set a threshold probability of success below which the moves are simply evaluated.

3.4.2 Twig Node Policies

The performance of (efficient) twig node policies has little impact on the asymptotic performance of the program. While we defer a precise formulation, the idea is that the asymptotic branching factor, R, (defined in section 2.2.3) is not altered, because even in the best case, all the algorithms' run times are exponential in the (smaller) number of moves available to one player. But, for a $m \times n$ twig node, the number of evaluations used for an efficient policy is upper bounded by $O(mn)$ and lower bounded by $\Omega(\min(m, n))$, so that the asymptotic cost is not altered (i.e., exponentially many twig nodes are evaluated anyhow, and $\lim_{d\to\infty}(Cmn)^{\frac{1}{d}} = 0$). Intuitively, policies at twig nodes cannot do better than reduce the effective depth of search by one.

3.4.2.1 Move Ordering

We employ a heuristic function (depending on the game state in which a move is made) to estimate the value of a move by itself. Moves are ordered by the value of this function.

3.4.2.2 Evaluation

As noted in section 3.4, it is unwise to test nodes instead of evaluating them. Accordingly, dominating evaluation cannot be appreciably better than naive evaluation at twig nodes. Hence, at a twig node, the substructure evaluation algorithm is the only realistic option. Naturally, the substructure algorithm should use no tests (we described how to perform substructure evaluation using only recursive evaluations in section 3.3.4).

There might be some games for which the cost of solving linear programs is sufficiently expensive (relative to leaf evaluation) that it is worth considering evaluating more leaves in order to solve fewer linear programs. One approach to reducing the time spent solving programs is simply to scan for saddle points. To do this, one evaluates all the cells in a row or column and then takes a best reply to it and evaluates all these cells, continuing this process until a saddle point is found or no best reply to the evaluated moves is unevaluated. If a row and a column are best replies to each other, this is a saddle point (naturally one stores search information to avoid reevaluating cells). If the best reply has already been searched, one can continue by scanning a move that has not been completely evaluated; stopping only when a saddle point is found or when all the subgames have been evaluated. If every subgame has been evaluated, one uses the linear program for solving games and returns the solution.

However, once the best replies to the searched moves have been searched, the substructure of searched moves is closed under best reply. One can then start to employ the usual algorithm; taking the initial substructure to consist of the searched moves. If some move fails to be bounded by the solution to the substructure, it is (as usual) added to the substructure. However, before solving this substructure, one scans for a saddle point (starting with the new move), and continues adding to moves to the substructure until it is again closed under best-reply (or a saddle point is found). Once this is done, the new substructure is solved, and the solution tested (and if it fails, we again add moves until it is closed under best-reply). This process continues until the substructure's solution is determined to be a solution to the game or until a saddle point is found.

Note that the substructure need not be closed under the best-reply correspondence for it to contain a solution to the game. That is, the best-reply to a (pure) strategy in the support of a solution strategy need not be in the support of the other player's solution strategy. An extension of the idea of requiring closure under best-reply is to require the substructure to be closed under k-th best replies. This would further reduce the number of linear programs solved.

There are other possible techniques for reducing the number of programs solved. For example, one could use a policy of requiring the substructure's size to grow by a certain amount before solving it.[8] This growth can be achieved by evaluating all cells of the support of the strategy and adding the required number of replies to the substructure, choosing those with highest expected value. A dynamic (metareasoning) approach could use a rapidly computable heuristic model of the chance of success and the extra cost of failure to select when to assume that the substructure contains a solution and stop expanding it.

Any technique that tries to reduce the number of linear programs solved will tend to not only evaluate more cells, but also to solve larger linear programs (when evaluation is completed, the substructure will be significantly larger).[9] This leads us to expect that not attempting to reduce the number of linear programs is the best approach to use in most games.

3.4.2.3 Testing

We describe the case of testing an upper bound. Lower bounds are analogous. We present a simple policy that avoids performing expensive operations, like computing multiple linear programs, at the cost of possibly evaluating excess leaf nodes.

For this policy, we employ a threshold, l. This is the maximum number of columns that we are willing to examine to verify that the bound holds. This threshold can be set statically, or using a metareasoning approach.

The approach we employ is to search for a single column to establish the bound, or for a single row that guarantees the bound will not hold. If neither is discovered, we evaluate all the cells in the l columns, and return the value of the linear combination (the solution to this substructure will be a valid upper bound).

[8] For example, one might require geometric growth. This would reduce the number of programs solved to be $O(log(m+n))$, for a $m \times n$ matrix.

[9] The sole exception to this is if we discover a saddle point consisting of two moves, neither of which was ordered as the best move for its player.

```
Function Twig − Test− ≤ (vertex V, upper bound y)
(It returns an upper bound on u(V).)

let (m,n) ≡ a(V)
determine the threshold number of moves, l
for each (i,j) ∈ {1,...,m} × {1,...,l}
  e_ij ← false
c ← 1 ; r ← 0
test-column:
  x ← −∞
  for i = 1 to m
    if not e_ic then v_ic ← u(V_i,c) ; e_ic ← true
    if v_ic > x then
      if v_ic ≤ y then
        x ← v_ij
      else
        for j = c+1 to l
          if not e_ij then v_ij ← u(V_i,j) ; e_ij ← true
          if v_ij ≤ y then
            c ← j
            if i > r then r ← i
            goto test-column
        if i − 1 > r then r ← i − 1
        goto combination-test
  return x (column c establishes this upper bound)
combination-test:
  for i = r+1 to m
    f ← true
    for j = 1 to l
      if not e_ij then v_ij ← u(V_i,j)
      if v_ij ≤ y then f ← false
    if f then return −∞ (none of the subgames in row i
      are bounded above by y)
  for i = 1 to r
    for j = 1 to l
      if not e_ij then v_ij ← u(V_i,j)
  return the value of the m × l substructure whose payoffs
    are contained in v
```

In this algorithm, e_{ij} is used to store which subgames have already been evaluated. r is used to keep track of the last row for which some move was found to lead to a subgame whose value is at most y. Indeed, all the rows up to r have this property, since for the column that was searched immediately before the maximal value of r was set, the subgames in each of the rows up to r were bounded above by y (and if r was set to i, some other move leads to a subgame that was so bounded). Therefore, we first search the rows after r, since it is only these rows that can lead to an early failure.

This algorithm could be modified to dynamically add or remove moves. We could compute constraints on any linear combination that would allow the upper bound to be met, however this would require us to solve several linear programs (and would lead to poor worst case behavior). It might be worthwhile considering what kind of constraints can be quickly maintained and tested.

3.5 Optimizations

As in 2ZPI games, we can use various techniques to improve performance (either to speed search or to improve solution quality). Many of the techniques for these games are directly applicable. Others require some modification in this context. We described the relevant techniques for 2ZPI games in chapter 2.

3.5.1 Iterative Deepening

This important method is essentially the same as for 2ZPI games. The program begins by performing a depth 1 search. It then increments the search depth and starts a new iteration. This process continues until the tree is searched, or if the program has a certain amount of time allotted to search, until there is insufficient time for another iteration (no new iteration should start once half of the allotted time has been used). Search during an iteration should be halted as soon as time does run out.

If time does run out during an iteration, the solution from the previous iteration must be used in some cases. However, information from the partially completed iteration can often be used. For both evaluation algorithms, this information consists of partial results from the partially completed search conducted at the root.

For dominating evaluation, information can be recovered once all the moves (for the player whose moves are being tested) in the support of the previous iteration's solution have been evaluated. The result returned is obtained by solving the substructure consisting of all the fully evaluated moves.

For substructure evaluation, use of the new iteration requires that all the moves in the support of the previous solution are either in the current substructure or have been tested against a solution and found to not be a strict best reply to it. If this holds, then the solution to the substructure can be used as the solution to the root. This criterion (for substructure evaluation) is a conservative one. That is, before this holds, the new information might be better but it is not clearly an improvement.

It is arguably better to set the substructure at the root (only) to consist of all moves in the support of the previous iteration. This reduces the delay before useful information is produced, albeit at the cost of possibly slowing the iteration (by evaluating excess subgames).

As with 2ZPI games, iterative deepening allows better time control, gathering of useful information and is of little cost (since almost all the time is spent on the last iteration).

Note that even if we are going to search the full game tree, using iterative deepening (with approximation games) should expedite search.

3.5.2 Non-Standard Measures of Depth

We can use non-standard measures of depth to allow for "quiescence" search, as in 2ZPI games. The basic idea is that the measurement of depth is not incremented if certain conditions are met. These can be (rare) game-specific conditions relating to the state that suggest it is important to consider successors more carefully. One way to handle this is to extend those positions whose deviation is above a threshold value. Alternatively, one can simply use game-specific features in the state.

A dynamic measurement like singular extensions can also be used. Singular extensions can be extended to simultaneous play games fairly straightforwardly: depth should increase if either player's solution strategy is singular. This is defined to mean that the strategy's support is a singleton and the expected value (determined by the the search to depth $\pi(d)$) of any other move against the opponent's solution strategy is below a constant "singular margin." In simultaneous play games, quiescence appears to be less of an issue because in such games one expects not to have the problem of having a single player's move that can radically alter the position. If we assume that the other player always has reasonably strong moves, there is more stability than in games where one player always has the "initiative."

3.5.3 Transposition Tables

Another important optimization technique is to use a transposition table. This is used to prevent re-search of states that can arise through different sequences of play or that have been previously searched (this is important because we do shallower searches to improve move ordering and if we use iterative deepening). The information in the table is also of auxiliary use, even if search has not been conducted to the required depth for the appropriate kind of information (i.e., upper and lower bounds or evaluation). Most commonly, this information can be used instead of conducting a shallow search in various places.

If a game state appears in the transposition table, it may be possible to use the stored information in lieu of re-searching the state. As in 2ZPI games, this can be done if we have stored appropriate information from a previous search to the required depth. To avoid evaluating a game state, then the state has to have been evaluated (to the required depth). If it is to be tested, then to avoid testing the position, we must have successfully established at least as tight a bound as sought, or unsuccessfully tested with at least as loose a bound as sought, or have evaluated the position. It is preferable to test for this information in the parent of the given state, so that bounds can be set more appropriately, and evaluation information is incorporated (by setting means exactly and setting deviations to zero). The policies for setting bounds described how to handle the case where the value of a position is exactly known.

If all the subgames to be tested have bounds of the appropriate kind, it is worth checking to see if the desired bound has already been met. This is likelier for substructure evaluation, since fewer subgames will be tested. Bounds (of the appropriate kind) established by a sufficiently deep search are used only to prevent re-search and not to influence the setting of bounds. However, if one implements a method for incorporating test information in the distribution of variables this information should also be used. Incorporating test information into distributions is discussed in section 3.6.2.

Transposition tables should hold the search depth for each of: evaluation, upper bounds and lower bounds. They should hold the tightest range of successful and unsuccessful test values (at the deepest level that a bound of the relevant kind has been tested). They should also hold (for evaluation) a solution value and solution strategies. If a position has been evaluated or bounded absolutely (i.e., all the terminal positions encountered were true terminal positions), the depth is stored as the maximum depth of the (true) game tree. Naturally, the random variable of the value (in a depth d approximation game) of a state whose value has been searched to depth d has zero deviation.

It is also possible to store additional information, like move ordering information, or unexpected results encountered, to assist re-search of a subgame.

Details like hash codes, and removing old states from the table are relatively unimportant and should not vary appreciably from those typically employed in chess programs. In any case, these are mostly game-specific issues.

3.5.4 Non-Speculative Forward Pruning

Other optimizations can rely on special properties of the game (or heuristic functions). We can use non-speculative forward pruning. For our purposes, we use bounds on the amount that the heuristic value can change after the players both move (if we have performed a shallow search, then we need to consider only the change in value possible after $d - \pi(d)$ moves). We can use this information to successfully test positions without search (and to return the tightest bound possible).

3.5.5 Miscellaneous Optimizations

When using iterative deepening, it could be beneficial to store the information obtained from searching the subgames in the "principal tree." The information of this type that was gathered on the last iteration can be used for move ordering. This information is used like transposition table entries, except it is never overwritten. The principal tree is an extension of the concept of a principal variation: any node that can be reached by best-play (in the approximation game just searched) is in the principal tree. It is a tree because if either solution strategy's support is not a singleton, then more than one successor can be reached by best play. Gathering this information requires the overhead of storing it for each cell that is evaluated. Because of this, we believe that it is generally preferable to store this information only for the root and to rely on the transposition table for the other levels. This information could also be used to restart search for the next move at depth $d-1$ instead of depth 1, should the resultant position be in the principal tree, where d is the depth of the iteration that produced the solution. This advantage is fairly minor.

One can further use storage to reduce search effort by building a full search tree. This tree would store search information for each node in the tree to facilitate search when testing a new bound or when evaluating after testing.

An alternative is to modify the algorithms to return more information about search without actually building a full search tree; returning information on what subgames' values were problematic could assist further search. This is similar to some optimizations that can be made for SCOUT.

If there are vertices at which one of the players has but a single move, some of the routines can be optimized. Specifically, linear programs can be replaced by simple maximization or minimization, and bounds setting and test choices become trivial. If one is solving a game where this situation occurs, it is beneficial to handle it specially when it does. Also, we note that if one adopts the testing policy of recursively testing each move, then both evaluation algorithms reduce to the SCOUT algorithm in the case of a two-player zero-sum game of complete information. For dominating evaluation, the domination test reduces to simply testing the next move's value. For substructure evaluation, it is the same because we never need to retest a bound. The reason for this is that if we find a move that is better than the current best move, then all the previous moves that were bounded by the previous best move will be bounded by the (weaker) bound of the value of the new best move. Because achieved test values are stored, the other moves are never re-searched, so moves are tested at most once and evaluated if and only if they are better.

Another kind of optimization can be incorporated in valuing moves at a twig node (for ordering). Various dynamic ordering heuristics, like the killer heuristic or the history heuristic, can be incorporated into the heuristic that scores single moves.

Some of the ideas for optimization of 2ZPI games that we listed in section 2.3.6 are applicable to simultaneous play games. Game-specific opening and endgame techniques can be used. Performing incremental updates is also beneficial for these games. In a simultaneous-play game, searching on the opponent's time is not much of an issue[10]

3.6 Metareasoning

Our proposed policies already incorporate certain aspects of metareasoning. The chief use is in handling bounds by estimating failure probabilities. However, in several other places in our policies, we have employed various approximations and static heuristics for control (such as fixed thresholds, penalties and re-test policies). As we noted, these could profitably be replaced by dynamic policies that attempt to compute the actual expected gain of setting new bounds or evaluating cells or testing a new collection of columns or rows. We discuss various approaches to doing this. It is a promising area for future research.

3.6.1 Determining and Propagating Expected Values and Deviations

As we noted above, simply taking the expected value of a position to be the solution of the game with the expected values of the subposition generally gives an approximation and not the actual value. We discussed some approaches to this issue in section 2.2.4.2. We now consider possible approaches to this problem for simultaneous play games.

The least sophisticated approach to setting expected values and deviations is to simply use a static heuristic estimate for a state (without using information from a depth $\pi(d)$ search). Slight improvements can be made (without using a shallower search) by incorporating transposition table or principal tree information, a killer heuristic (or a generalization thereof), and heuristics about moves, as is done in many 2ZPI games. However, as we noted

[10]If the program has a deadline time, by which time moves must be received, and it submits a move before the deadline (this could happen if it had less than half the total time remaining after finishing an iteration), it might be able to search while waiting for its opponent to submit a move.

before, this approach seems unwise, because we need to use this information for computing probabilities and not merely to order moves.

The approach described earlier is to use a shallow search to obtain expected values and to take the expected value of the game to be the solution value from the search of the subpositions. The deviations are obtained by use of static game features. A possible use of metareasoning would be to determine the deviations for searching a leaf node to depth one by sampling, indexing the deviations by game-specific feature-based categories (much as Russell and Wefald did for their MGSS algorithms). A slight extension of this is to gather statistics for deviations between depth d and depth $d-1$ search, according to feature-based categories. This suffers the disadvantages that it is quite expensive to do so for all the depths and that it does not account for the mean and deviation information from successive subgames.

A more feasible idea is to modify the deviation for expanding a leaf node by various features for deep backup, making it a function of the depth, branching factor information and various qualities in the payoff matrix (including shallower search information). The deviation of a subgame that was evaluated by only performing a depth $\pi(d)$ search should be increased (especially if $\pi(d) = 0$). Quiescence search should also reduce the deviation of the resulting position, i.e., deeper search reduces deviation.

The ideal method would be to derive a tractable, low-discrepancy approximation to the actual random variable in question; using deviation information from single-step lookup and determining mean and deviation approximations from those of the subgames. One likely approximation would be to use the actual mean and deviation values, but to assume the variable is normally distributed. However, efficiently determining these precise values is unlikely to be possible. It does not appear that one can analytically determine the distribution of the random variable of a game whose payoff values are independent normal variables.

If one had a sufficiently accurate model of the value of positions, it could be preferable to select a strategy that uses the *expected value* of an (approximation) game and not the minimax value thereof. However, for any (tractable) approximation of which we can conceive, it seems better to simply take the expected value to be the minimax value.

Another possible approach to determining deviations is to approximate by Monte Carlo simulation (though this is probably too slow). Other analytic approximations can be computed. A simple approach is to take the linear combination of the random-variables of subgames reached by play of solution strategies (weighted by probability of being played). This is again a normal variable, assuming the subgame values are independent (or even if the values are jointly normal). However, this approach does not incorporate any information about moves not in the support of the solution strategies. This is likely to produce especially poor results in the case that the cardinalities of the solution strategies' supports are small.

Another simple approach to estimating deviation is to compute the random variable given by fixing one player's solution strategy. This is either a maximum or a minimum of independent random variables. We can combine the deviations of these variables by some simple method (such as taking the geometric mean). This approach only requires that we obtain information from a search to depth $\pi(d)$ for those subgames that can be reached by playing one solution strategy (as noted before, we need to increase the deviation of any subgames that have only been searched shallowly in this fashion).

3.6.2 More General Variables

Let us consider what properties a class of variables would need to capture all the information we would like to have about our random variables. If we assume that the values from a single-step backup are normally distributed, then this class needs to contain all normal random variables.

We would also like the class to be closed under the operation of solving a game whose values are variables in the class. Finally, we would like to have the class closed under certain kinds of the conditional information about the probability of a variable in the class. Specifically, if Y is in the class, then we would like to be able to include the distribution of Y, conditioned on the information that $P\{Y \leq z\} = p$ or that $P\{Y \geq z\} = p$. The idea behind this is to capture the increased likelihood that a subgames value is not bounded by z if we unsuccessfully tested the subgame against the value z. Also, if we test a subgame and successfully bound it, we could use $p = 1$ to assert that the value is bounded.

If one can produce a better approximation to this class than simply employing normal approximations for all these operations, it should be possible to improve performance. It is also worth considering how one can incorporate dependence among subgames into the model of these random variables.

Palay[Pal83] discusses approaches to the more restricted problem of dealing with distributions that are closed under the operations of taking the maximum and the opposite. It may be possible to extend approaches like polynomial approximation or using representative points to approximate this class of variables. Taking a maximum only requires closure under multiplication, which is easy to perform using such approximations. However, for simultaneous play games, it would be necessary to create an approximate distribution based on the solution to a matrix game of such variables. This is anticipated to be a much more difficult problem.

3.6.3 Metareasoning Control of Testing

As noted earlier, it appears promising to continue to test for bounds in the test and evaluation algorithms until the following condition holds:

$$\frac{P_{\text{success}}}{P_{\text{failure}}} < \frac{E[\text{cost}_{\text{failure}}]}{E[\text{cost}_{\text{success}} - \text{cost}_{\text{failure}}]}$$

Given our normal approximation for variables, the probabilities above are fairly straightforward to approximate.

To allow reasoning about the cost of computation, we need to pass an additional parameter, f, to the test procedure: the expected net cost of failure.

Suppose k bounds have been determined and ordered. Let the expected cost of testing the ith bound be $\mathcal{C}_i$ and the probability of successfully establishing the bound be p_i. We generally approximate the probability of successfully testing a bound by the probability of its holding, as we have done above. In addition to the possibility of a bound holding but not being shown by the test, there is also the possibility of determining a tighter bound that increases the probability of subsequent bounds being shown by the test. Both factors might be accounted for by slightly perturbing our distributions. We now consider under what conditions the contemplated test should be performed.

The expected cost of performing the test is:

$$\sum_{i=1}^{k} \mathcal{C}_i \prod_{j=1}^{i-1} p_j$$

That is, the cost is the expected cost, given the probability of failure for each bound. The expected gain from performing the test is:

$$f \prod_{j=1}^{k} p_j$$

That is, if all the bounds are established, then the program avoids the cost f. We are testing only whether these (best available) bounds should be used. If these bounds will not be used, none will. Hence, we do not consider the possibility of subsequent tests succeeding, which would reduce the benefit from succeeding on this test. So we proceed if and only if

$$\sum_{i=1}^{k} \mathcal{C}_i \prod_{j=1}^{i-1} p_j < f \prod_{j=1}^{k} p_j$$

Now, we also need to describe how to compute $\mathcal{C}_i$ and the related problem of what the expected cost of failure for a given subgame will be.

To compute $\mathcal{C}_i$, we employ a simple model of the cost of testing. This is approximately $vb^{d-1} + w$, for b the average branching factor in tests during the game, and v, w parameters to fit test costs to an exponential curve. All three parameters should be determined by statistical sampling. As with estimating deviations, one can make the parameters functions of various features (either game-based or dependent on the probability of success). Naturally, there is some amount of feedback; more accurate data should improve the performance of the Test procedure, which reduces the expected cost of testing, while increasing the likelihood of success.

To determine what expected cost of failure should be passed to a subgame, we use a similar computation to that used to determine whether to proceed. The main distinction lies in the fact that here we wish to incorporate the possibility of failing on these bounds, but subsequently successfully establishing other bounds.

Let f_i indicate the expected net cost of failure we will pass to the subgame that is tested using the ith bound. Then,

$$f_i = (1 - \prod_{j=i+1}^{k} p_j)(f - r) - \sum_{j=i+1}^{k} \mathcal{C}_j \prod_{l=i+1}^{j-1} p_j$$

where r indicates the residual gain from further tests (i.e., the expected gain of all further computation). Ideally, r would be the expected value of future computation, taking the expectations over the possible failure of the individual bounds. However, a reasonable approximation is to take r to be

$$\max\left(0, f\prod_{j=1}^{k} p'_j - \left(\sum_{i=1}^{k'} \mathcal{C}'_i \prod_{j=1}^{i-1} p'_j\right)\right)$$

In this formula, p'_i, $\mathcal{C}'_i$ and k' indicate the probability of success, expected cost and number of bounds for the "second best" bounds known.

Alternatively, one could simply let r be qf, for q a parameter in the range $(0, 1)$ that is set by sampling (and possibly dependent on features).

The overall cost of failure for an evaluation routine that calls Test is simply $\sum_{i=1}^{k} v_i b_i^{d-1} + w_i$, where v_i, b_i, w_i are parameters for the cost of *evaluating* a given subgame and k is the number of subgames that would be evaluated. Otherwise, evaluation routines perform the same calculations when determining whether to test bounds or to fail (i.e., evaluate).

3.7 Parallelism

The task of solving simultaneous play games lends itself quite well to parallelism. Whenever a collection of subgames must be evaluated, one can obtain nearly linear speed-up (in the number of processors) by doing so in parallel (assuming the subgames take approximately the same time to evaluate). Testing a collection of bounds also can be speeded by doing so in parallel (especially when the bounds are all likely to be successfully established). The nature of solving simultaneous play games, where there are many independent things to be examined at once, makes parallelism more promising in this context than in sequential play games.

The best technique for allocating parallel processors will vary by game (mainly dependent on branching factors and the cardinality of solution supports) and the number of processors available. A good approach is to use teams of processors, recursively splitting the processors when performing operations in parallel (if some processes finish early, additional processors can be allocated to unfinished teams). If the game has chance alternatives it is generally necessary to search each of them, so this is a very good place to incorporate parallelism. Handling chance alternatives is discussed in section 3.10.

Naturally, if only a few processors are available it is more sensible to use the processors on large parallel evaluations because such evaluations must all be done, whereas tests must be stopped once one fails. However, in substructure evaluation, there are not as many opportunities for large parallel evaluations (this is done only when a move has not been bounded), though using parallelism in testing also appears promising; our empirical research (described in section 3.9.3) suggests that much of the time in evaluation is spent in verification of a solution (i.e., all the bounds are successfully established).

The naive evaluation algorithm can be sped up by a factor of p, when using p processors, if the game tree is of uniform depth and branching factor, and almost as much in any game where subtrees of positions are reasonably balanced in terms of number of nodes. The dominating evaluation algorithm also ought to be sped up almost p times when using p processors.

There is clearly much potential for incorporating parallelism in these algorithms. It is a promising direction for future research to explore efficient policies to determine how best to schedule tasks to be performed in parallel. This might well be complemented by a metareasoning system (to determine the most promising places to split a processor team to act in parallel).

3.8 Asymptotic Analysis

We now compute the theoretical best and worst case performance for the various algorithms. We analyze the cost for uniformly deep matrix trees of depth d, where all the matrices are of size $b \times b$, for $b > 1$.

We believe that these results are indicative of performance on more general trees, including those where there is an imbalance in the number of moves that the players have at any stage. We have informally examined recurrences similar to those presented below for trees where the levels alternate between size $b \times c$ and size $c \times b$. In this case, we conjecture that the growth is proportional to $(bc)^{1/2}$ instead of b.

Costs for Non-Recursive Operations Let the time to perform a terminal evaluation lie between c_1 and C_1. We noted earlier that an $m \times n$ linear program can be solved for both the primal and dual problem, in time $O(mn^4)$. Let us take C_l to be such that the time to solve any linear program is at most $C_l mn^4$.

For the efficient algorithms (i.e., those other than naive evaluation), we assume that the time for other non-recursive operations (such as ordering moves and setting bounds) is bounded above by $C_2(b \log b)$ at deep nodes, and by $C_2 b$ at twig nodes. This assumes that we do not actually sort moves by score at twig nodes, but use a faster method of generating an approximate ordering. Using sorting at twig nodes cannot increase the total time for the algorithms by more than a factor of $O(\log b)$.

Naive Evaluation: Let us compute the cost of naive evaluation for both the best and worst cases. If $j(n)$ is the time to naively evaluate a game of depth n, then we have:

$$c_1 \leq j(0) \leq C_1$$

and

$$b^2 j(n-1) \leq j(n) \leq b^2 j(n-1) + (C_l + C_2) b^5$$

for $n > 0$. We conclude that $j(n) \in \Omega(b^{2n})$ and $j(n) \in O(b^{2n+3})$.

We shall now analyze the asymptotic costs for the best case and worst case performance of the more efficient algorithms.

3.8.1 Best Case

In the best case, no linear programs need to be solved (as can be inferred from the description of the best case for each algorithm).

We first treat the case where no shallow search (to $\pi(n)$ levels below) is performed, i.e., we assume constant-time heuristics for estimating the value and deviation of each subgame.

Test costs: Let $f(n)$ be the cost of using Test to successfully establish a bound on a subgame whose tree is of depth n (since the matrices are square, this will be the same for upper and lower bounds).

In the best case, a single move is tested and each subgame that the move can reach is successfully tested. Clearly, no less work can be done for a successful test.

Putting these observations together, we have

$$\begin{aligned} f(1) &\leq (C_1 + C_2)b \\ f(1) &\geq c_1 b \end{aligned}$$

and for $n > 1$,

$$\begin{aligned} f(n) &\leq bf(n-1) + C_2 b \log b \\ f(n) &\geq bf(n-1) \end{aligned}$$

Therefore, for all n,

$$\begin{aligned} f(n) &\leq \frac{(C_1 + C_2)b^n(b + \log b)}{b - 1} \\ f(n) &\geq c_1 b^n \end{aligned}$$

So, $f(n) \in \Theta(b^n)$.

Dominating Evaluation Costs: Let $g(n)$ be the cost of performing dominating evaluation on a game tree of depth n.

In the best case, one evaluates a single move and uses tests on all the other moves (of the same player), with all the tests succeeding (so that the moves were dominated). It is no more expensive to successfully test a move than to evaluate it (this is easily shown by induction), so this is indeed the best case. Because all but one move is dominated, no linear program is solved. We thus have the recurrences

$$\begin{aligned} g(1) &\leq C_1 b^2 + C_2 b \\ g(1) &\geq c_1 b^2 \end{aligned}$$

and for $n > 1$,

$$\begin{aligned} g(n) &\leq bg(n-1) + b(b-1)f(n-1) + C_2 b \log b \\ g(n) &\geq bg(n-1) + b(b-1)f(n-1) \end{aligned}$$

Solving these recurrences, we obtain $g(n) \in \Theta(nb^{n+1})$.

If we successfully scan for saddle points at twig nodes, the best case becomes $\Theta(nb^n)$, because in this case we have $c_1 b \leq g(1) \leq (C_1 + C_2)b$.

Substructure Evaluation Costs: Let $h(n)$ be the cost for substructure evaluation on a game tree of depth n.

In the best case, one evaluates a single subgame, and runs (successful) tests on all the replies for each player. Again, it is no more expensive to successfully test a move than to evaluate it. Also, in the best case we never add a move to the substructure, so we do not need to solve any linear programs. This yields

$$\begin{aligned} h(1) &\leq C_1(2b-1) + C_2 b \\ h(1) &\geq c_1(2b-1) \end{aligned}$$

And for $n > 1$,

$$\begin{aligned} h(n) &\leq h(n-1) + (2b-2)f(n-1) + C_2 b \\ h(n) &\geq h(n-1) + (2b-2)f(n-1) \end{aligned}$$

Thus we obtain $h(n) \in \Theta(b^n)$

3.8.1.1 Incorporating Shallower Search for Bounds

Let us now consider the effect of performing shallow evaluations to set bounds information. We shall assume the existence of two different discount functions, π_e for the evaluation functions, and π_t for the test function. The motivation for this is to allow for less costly searches in testing, so we assume that for each n, $\pi_e(n) \geq \pi_t(n)$.

We always perform a single depth $\pi_t(n)$ search when testing, and a single depth $\pi_e(n)$ search when evaluating. Let us assume that our evaluation policy also has us perform between $z_1(b)$ and $Z_1(b)$ depth $\pi_e(n)-1$ evaluations (on subgames). Likewise, we assume that our test policy requires us to perform between $z_2(b)$ and $Z_2(b)$ depth $\pi_t(n)-1$ evaluations.

Test Costs: We use f_g to denote the test cost if dominating evaluation is used for shallow search, and f_h to denote the test cost if substructure evaluation is used for shallow search. We shall use A to denote the cost function for the evaluation algorithm. For $\pi_t(n) \leq 0$, no shallow evalution is performed. Let w be the least n for which $\pi_t(n) > 0$. Then if $n < w_t$, we can use the earlier bound on the cost of testing without shallow search, so there exist constants c_3, C_3 such that: For $n < w_t$,

$$\begin{aligned} f_A(n) &\leq C_3 b^n \\ f_A(n) &\geq c_3 b^n \end{aligned}$$

For $n \geq w_t$,

$$\begin{aligned} f_A(n) &\leq A(\pi_t(n)) + Z_2(b)A(\pi_t(n)-1) + bf(n-1) + C_2 b \log b \\ f_A(n) &\geq A(\pi_t(n)) + z_2(b)A(\pi_t(n)-1) + bf(n-1) \end{aligned}$$

For $\pi_t(n) = n-k$, there are constants c_A, C_A such that for $n \geq w_t$,

$$f_A(n) \leq C_A b^n + A(n-k) + (b + Z_2(b)) \sum_{i=1}^{n-k-1} b^{n-k-1-i} A(i)$$

$$f_A(n) \geq c_A b^n + A(n-k) + (b + z_2(b)) \sum_{i=1}^{n-k-1} b^{n-k-1-i} A(i)$$

Dominating Evaluation Costs: For dominating evaluation, we likewise take w_e to be the least n such that $\pi_e(n) > 0$. For suitable constants c_4, C_4 we obtain the recurrences, for $n < w_e$,

$$\begin{aligned} g(n) &\leq C_4 n b^{n+1} \\ g(n) &\geq c_4 n b^{n+1} \end{aligned}$$

and for $n \geq w_e$,

$$\begin{aligned} g(n) &\leq g(\pi_e(n)) + Z_1(b) g(\pi_e(n) - 1) + b g(n-1) + b(b-1) f_g(n-1) + C_2 b \log b \\ g(n) &\geq g(\pi_e(n)) + z_1(b) g(\pi_e(n) - 1) + b g(n-1) + b(b-1) f_g(n-1) \end{aligned}$$

We conjecture that if $\pi_t(n) = \lfloor yn \rfloor$ (for $0 < y < 1$), then $g(n) \in \Theta(nb^{n+1})$ and if $\pi_t(n) = n - r$ (for $r > 1$), we conjecture that

$$g(n) \in \Omega\left(((1 + z_2(b))b)^n\right)$$

and

$$g(n) \in O\left(((1 + Z_2(b))b)^n\right)$$

In both cases, we conjecture that the choice for π_e is irrelevant.

Substructure Evaluation Costs: For substructure evaluation, there are constants c_5, C_5 such that for $n < w_e$,

$$\begin{aligned} g(n) &\leq C_5 b^n \\ g(n) &\geq c_5 b^n \end{aligned}$$

and for $n \geq w_e$,

$$\begin{aligned} h(n) &\leq h(\pi(n)) + Z_1(b) h(\pi(n) - 1) + h(n-1) + (2b-2) f_h(n-1) + C_2 b \\ h(n) &\geq h(\pi(n)) + z_1(b) h(\pi(n) - 1) + h(n-1) + (2b-2) f_h(n-1) \end{aligned}$$

We conjecture that if there is an n_0 such that for all $n > n_0$, $\pi_t(n) < n-1$ and if $Z_2(b) < Kb$, for some constant K, then $h(n) \in \Theta(b^n)$. This includes the case of $\pi_t(n) = \lfloor yn \rfloor$, naturally. For $\pi_t(n) = n - 1$, we conjecture that $h(n) \in \Theta(nb^n)$.

3.8.2 Worst Case

For our worst case analysis, we do not consider the effects of shallower searches. We conjecture changes in behavior below.

Test Costs: For testing, let us assume that our policy is to never recursively test more than $Z_3(b)$ moves, for $Z_3(b) \le b$. In the worst case, we always test this many moves, and fail on the last test for each collection tested. We assume the time to optimize bounds against a collection of size q is never more than C_5bq, so the cost of determining test bounds is bounded above by $C_5Z_3(b)b$.

At a twig node, in the worst case, we also solve a $b \times Z_3(b)$ linear program, in time that is bounded above by $C_l b\,(Z_3(b))^4$. We use f to represent the time required for a test, though in this case we bound the time required for any test (successful or not).

We obtain the recurrence:

$$\begin{array}{rcl} f(1) & \le & C_1 Z_3(b) b + (C_l\,(Z_3(b))^4 + C_2) b \\ f(n) & \le & Z_3(b) b f(n-1) + C_2 b \log b + C_5 Z_3(b) b \ \text{ for } n > 1 \end{array}$$

So, $f(n) \in O((Z_3(b))^{n+2}\, b^n)$.

From now on, we assume that $Z_3(b) \in O(\log(b))$, so we have $f(n) \in O(nb^n)$.

Dominating Evaluation Costs: In dominating evaluation, when we are evaluating a deep node, we cannot do worse than to evaluate each move after having tested all b subgames that that move can reach. If we recompute linear programs after every test that fails, we might need to solve b linear programs of each of the sizes $b \times 1, b \times 2, \ldots, b \times (b-1)$ and a final linear program of size $b \times b$. The cost of solving these linear programs is bounded above by $C_l(b(b-1)+1)b^5$, so it is bounded above by $C_l b^7$. If we do not recompute linear programs after tests fail, we can reduce the worst case cost by a factor of b.

At a twig node, we simply evaluate all the children and solve the linear program, giving a cost for linear programming that is bounded above by $C_l b^5$.

Using g to represent dominating evaluation costs, we obtain

$$\begin{array}{rcl} g(1) & \le & C_1 b^2 + C_l b^5 + C_2 b \\ g(n) & \le & b^2 g(n-1) + b(b-1) f(n-1) + C_2 b \log b + C_l b^7 \ \text{ for } n > 1 \end{array}$$

Therefore $g(n) \in O(b^{2n+3})$.

Substructure Evaluation Costs: For substructure evaluation, we cannot do worse than to test each move and to fail on the last test, then evaluate the move.

Let us consider the most tests that can be performed in a single substructure evaluation. If we have a $b' \times c'$ substructure and every move in the substructure is in the support of its solution, then we might perform as many as $(b-c')b' + (b-b')c'$ tests before adding to the single substructure (or successfully completing). For any values of b' and c' (between 1 and b), this value is bounded above by $b^2/2$. We cannot test more than $2(b-1)$ substructures, since we always add a move to the substructure before testing again, and the substructure starts with two moves, and is not tested if it contains all $2b$ moves. So we conclude that no more than $b^2(b-1)$ tests can be executed when performing a substructure evaluation.

We never solve more than $2(b-1)+1$ linear programs, at a total cost of at most $2C_l b^6$.

So, using h to denote the cost for substructure evaluation, we have the recurrences:

$$\begin{aligned} h(1) &\leq C_1 b^2 + C_2 b + 2C_l b^6 \\ h(n) &\leq b^2 h(n-1) + b^2(b-1)f(n-1) + C_2 b + 2C_l b^6 \text{ for } n > 1 \end{aligned}$$

Thus we obtain, $h(n) \in O(b^{2n+4})$.

Effects of Shallow Search: We conjecture that incorporating shallow searches will not alter the worst case bounds for either algorithm: the cost of shallower evaluations, being about $b^{2(n-\pi(n))} \geq b^2$ less expensive is dominated by the cost of the direct recursive operations.

3.9 Election Strategy—A Case Study

We devised a simple simulation that is a two-player zero-sum matrix-tree game. This simulation was devised primarily to verify the expected performance improvement of our algorithms, and secondarily to demonstrate a descriptive use of computational game theory. The primary focus of the project was not, however, to devise a complicated and accurate political model, although we did hope to create a simulation that would allow for us to examine the strength of play of such algorithms in a game that bears a qualitative resemblance to those that might be used to model phenomena in the social sciences.

The specific game in question has two candidates campaigning for election. The context of the game is to examine an election in a regionalized polity, in which regional campaign appearances improve a politician's popularity therein. Hence, the game focuses on the importance of geographical factors in elections.

3.9.1 Game Mechanics

In this game, the polity is divided into a total of R regions that form a graph. Each region has a certain proportion of the total voters. Voters are of five kinds: those strongly committed to one of the candidates, those weakly committed to one of the candidates and those who are indifferent. The categories are

1. strongly committed to candidate 1.
2. weakly committed to candidate 1.
3. indifferent.
4. weakly committed to candidate 2.
5. strongly committed to candidate 2.

The candidates have a certain degree of appeal to voters of each category (naturally, the appeal is stronger the more committed a voter is to the candidate).

Each turn, the candidates can use 8 "action points," each representing a combination of temporal and monetary constraints. Players can move to an adjacent region by expending three action points, and each campaign appearance costs one. We also experimented with a variant that allowed candidates to expend all 8 action points to fly to any region on the map. The only important aspects of a move are the number of campaign appearances made in each region during the turn and the final region the candidate visits during the turn. As such, we only used canonical moves, so that if there were multiple paths that could lead to the same number of campaign appearances in each region, and the same destination, only one of these moves would be generated. We also did not consider the possibility of making a non pareto-optimal move such as not using all one's action points, or moving in a cycle without campaigning during that cycle. Since the maximum number of times one can change location in a turn is two, the only way to move in a cycle is to move to a region and then immediately return to the original region. Such suboptimal moves can be excluded because campaigning always improves a candidate's standing, and in any state, having an improved standing in some region always leads to a better position: it makes it easier to win (and it makes it harder for the opponent to subsequently win these voters over to his side).

Campaign appearances only influence the voters in the region in which they are made. If candidate 1 makes k_1 campaign appearances and candidate 2 makes k_2 of them, then the proportion of the voters in category i that is shifted to category $i-1$ is $b_i k_1^t(1-b_{6-i}k_2^t)$ and the proportion shifted to category $i+1$ is $b_{6-i}k_2^t(1-b_{6-i}k_1^t)$. Voters who would shift to category 0 or to category 6 remain where they were (category 1 or 5, respectively). We set $t = .81$ and $b = (.1, .5, 1.0, 1.5, 2.5)$. b is a vector that models the degree to which a candidate appeals to more or less favorably inclined voters.

In the case of both candidates campaigning in the same region in a turn, this model produces an element of cancellation; but also a polarization effect occurs (whereby voters become more committed).

The scenario we used had the United States divided into seven regions:

- Pacific: Alaska, California, Hawaii, Oregon, Washington
- Mountain: Arizona, Colorado, Idaho, Montana, Nevada, New Mexico, Utah, Wyoming
- Central: Iowa, Kansas, Minnesota, Missouri, Nebraska, North Dakota, Oklahoma, South Dakota, Texas
- Midwest: Illinois, Indiana, Kentucky, Michigan, Ohio, West Virginia, Wisconsin
- South: Alabama, Arkansas, Florida, Georgia, Louisiana, Mississippi, North Carolina, South Carolina, Tennessee, Virginia
- North-East: Delaware, District of Columbia, Maryland, New Jersey, New York, Pennsylvania
- New England: Connecticut, Maine, Massachusetts, New Hampshire, Rhode Island, Vermont

The number of moves available to a player at any turn ranges from 15 to 49 when not using airplanes, and from 19 to 50 when using airplanes.

Starting Positions: To start the scenario, we assigned varying proportions of voters to the five categories of opinion of the candidates. In all the regions, most voters were undecided, and most of those who favored one candidate had a slight preference and not a strong one. We used randomized starting positions and initial distributions of voters. We tested starts that were fair (the candidates starting in the same region and with a symmetric distribution of votes) and those that were somewhat askew (allowing for somewhat different strengths in the various regions, and possibly different starting positions).

The game has a fixed number (20) of turns before campaigning ends and the election is held. The candidates attempt to maximize their probability of winning the election.

Final Outcomes: There are two final outcomes: a victory for candidate 1 or for candidate 2. Because our primary objective was to test the performance of the various algorithms, we used the simple approach of maximizing the expected vote difference. Accordingly, the value of a terminal vertex, to player 1, is the difference between the expected votes for him, and that for player 2. This objective allowed us to use simple heuristics to play the game.

Let p_i^j represent the proportion of the voters in region j who are in category i. We modelled the expected proportion of total voters who will vote for candidate 1 by $\sum_{i=1}^5 \nu_i p_i^j$, where $\nu = (.95, .8, .35, .05, .01)$. This vector reflects an expectation that a few voters will change their minds, and also models the idea that more opinionated voters are likelier to vote. The expected proportion of total votes for candidate 2 is, similarly, $\sum_{i=1}^5 \nu_{6-i} p_i^j$. So the expected difference in votes in region j is given by $P_j \equiv \sum_{i=1}^5 (\nu_i - \nu_{6-i}) p_i^j = .94p_1^j + .75p_2^j - .75p_4^j - .94p_5^j$. The overall score is then $\sum_{j=1}^R P_j w_j$, where w_j is the proportion of the total voting population contained in region j (determined based on the 1992 electoral college votes for each state).

3.9.2 Programming The Simulation

We implemented four evaluation algorithms: the naive evaluation algorithm, the dominating evaluation algorithm and two versions of the substructure algorithm. One version was the usual substructure algorithm; the other one "Pure Substructure," did not call Test on subgames (it tested moves by evaluating subgames).

We programmed the system using C++ and executed it on a SPARC-2 processor, to enable us to measure the actual time costs required for a reasonably fast implementation. We wanted to get an idea of the time required for such an implementation because our policies call for the use of linear programs and a conjugate gradient optimization technique. Also, because this class of games has not been programmed before, we did not wish to rely on the assumption that the number of terminal evaluations performed would be a reasonable estimate of the search time. We used the simplex method for solving linear programs, including solving the dual problem in those cases where it was expected to be faster.

For those policies where we previously described alternatives and did not clearly indicate a preference for one, we describe the specific policies that were implemented for the different evaluation algorithms. After this, we describe the specific heuristic functions we employed for this simulation.

3.9.2.1 Policy Choices

For all the policies, we employed static control of search instead of using metareasoning control. The specific values for various scaling parameters and thresholds were set by experimentation, to determine good performance, and we do not detail them here, as these are of little interest except for this specific game.

We implemented an earlier version of recursive value gathering, in which we simply obtained recursive search information for each subgame of a position (instead of evaluating just the game and only doing shallow search on unevaluated subgames whose value is required). This is expected to modestly worsen the performance of the algorithms.

Dominating Evaluation: We adopted the policy of not retesting subgames if a bound fails. At twig nodes, we performed a saddle point scan (so that leaves were evaluated until either a saddle point was found or all the subgames had been evaluated).

Testing: To choose columns and their linear weights, we had a threshold value so that if the value of the game (from the depth $\pi(d)$ search) was not such that the bound was very likely to be met, we performed a reduction on the solution strategy. The reduction strategy we used was to always reduce to a doubleton and we always eliminated a column with the smallest weight. If the reduction process produced a collection that had a higher score than any single column, we tested this collection. If the test failed, we did not test any other collections. If the reduction process did not produce a collection superior to any single column, we tested up to four moves (as singleton collections). We tested four unless a test succeeded.

For twig tests, we used a threshold of testing at most three moves.

Substructure Evaluation: We always used a substructure consisting of the highest probability moves in the support of the solution to the depth $\pi(d)$ game. We tested moves by their static move ordering, using alternating player moves. We did not impose any special conditions on twig evaluations to reduce the number of linear programs solved. Naturally, we did not perform tests when evaluating twig nodes.

3.9.2.2 Optimizations Used

We employed iterative deepening, including a mechanism for recovering partial information from incomplete searches. The game does not seem to require quiescence searching, as there are no features that indicate a state's terminal value is unreliable. However, it would be worthwhile to explore the consequences of using transposition tables, windowing and non-speculative forward pruning.

3.9.2.3 Heuristic Functions

We used $\pi(d) = d - 1$ when performing shallow search. We used a constant value of .65 for the deviation of each position. The function we employed to evaluate the expected value of a leaf node was similar to the probability of victory function. For each region j, we computed an advantage function,

$$\alpha(p^j) \equiv .9p_1^j + .6p_2^j - .6p_4^j - .9p_5^j$$

Here, p^j is the vector containing the proportion of voters in each category for region j.

The overall evaluation function was

$$\sum_{j=1}^{R} w_j \alpha(p^j) + \delta(l, p^1, \ldots, p^R)$$

Here, l is a vector of player locations.

δ is a function that captures the positional advantage of being located near to regions where there are many undecided voters. For $d(i,j)$ the length of the shortest path between regions i and j, we computed a raw proximity advantage, $\psi(l,j)$ for each region j:

$$\psi(l,j) \equiv w_j(d(l_1,j) - d(l_2,j))(2p_2^j + p_3^j + p_1^j)$$

Let those regions where this is positive have values (in decreasing order) of $\psi_1^1, \ldots, \psi_{R_1}^1$ and those where it is negative have values (in increasing order) of $\psi_1^2, \ldots, \psi_{R_2}^2$. We made δ produce an exponential decay, to capture the idea that positional advantage can be eroded by subsequent movement, so it is better to be close to a few excellent regions, than to many good ones. We defined:

$$\delta(l, p^1, \ldots, p^R) \equiv \zeta \left(\sum_{j=1}^{R_1} \eta^{j-1} \psi_j^1 - \sum_{j=1}^{R_2} \eta^{j-1} \psi_j^2 \right)$$

η is a constant to give exponential decay, and ζ is a scaling constant (we took $\eta = .7$ and $\zeta = .015$).

Most evaluation computations were performed incrementally (values for a region where neither player campaigned did not need to be recomputed).

Single moves were evaluated by determining the effect of making the move on the value of the state (against a "null-move" by the opponent, so that she did not campaign or move).

3.9.3 Observations, Notes and Results

This case study is intended to supply information for three different purposes. These are to evaluate the algorithms, to evaluate policy decisions and assumptions made when formulating policies, and to qualitatively assess the behavior of the program in the context of a simulation. For all of these purposes, we believe that this game is representative of a fairly general kind of game so that the information determined by examining this case study will be applicable to other games, including those of practical interest.

The most important goal of the case study is to assess the performance of the two improved evaluation algorithms in a realistic setting. To do this, we measured the search time and effective branching factor for the different algorithms, using several different starting positions and time limitations.

Our second objective is to measure the effect of specific policies and optimizations. Some of the statistics we collected allowed us to measure game and search qualities that have a bearing on various assumptions that we made when formulating policies.

Finally, we hoped to obtain qualitative information about the style and skill of computer play of this game. This information consists primarily of observations of the style of play of the computer, when playing games against itself or against human players. We also examined the value of look ahead by comparing having the computer play games against itself using disparate amounts of time for search.

Tests Performed: We collated data from the play of complete games and also for search of single positions. To facilitate performance comparisons, we never used the airplane variant. The test suite of single positions was generated randomly. To generate these positions, we picked a random turn number and assigned various voter distributions to reflect a "typical" distribution at this stage in the game. We used a total of 15 random positions. The algorithms searched to depth 3 in each position, except naive evaluation and substructure evaluation. Naive evaluation searched to depth 2 only, due to the much faster increase in search time for each additional depth of search and substructure evaluation searched to depth 4, due to the lower increase in search time. This test suite was performed primarily to compare the amount of time required for the various algorithms to search positions. We also collected statistics on various aspects of performance.

We tested play of complete games using a tournament format. Each tournament consisted of 10 games. These games were generated from 4 symmetrical positions and 3 asymmetrical positions. We searched the asymmetrical positions twice: allowing each algorithm to play both sides.

We used this tournament format to compare the two major algorithms (substructure evaluation and dominating evaluation), allowing the algorithms to search for 90 seconds per move on each position. The primary objectives of this test were to evaluate the quality of play for complete games, and to gather statistics about average depth of search and statistics that bear on policy decisions, in the context of positions that occurred during actual play.

We also executed a tournament in which one player used substructure evaluation with a time limit of 90 seconds per move and the other one used a search depth limit of 1 for substructure evaluation. This test was performed only to measure the advantage given by look-ahead.

3.9.3.1 Evaluating The Algorithms

We present data from both test procedures that indicates the overall speed of our algorithms.

We used iterative deepening only for time control purposes, i.e., no information obtained from an earlier search was used to assist subsequent searches. Accordingly, whenever we give statistics for a search to depth d, we indicate the statistics only for the dth iteration, and not the total for the first d iterations.

This data indicates that all three improved algorithms provided a dramatic reduction in the time and evaluations required to search to a given depth. The data indicates that the

Evaluation Algorithm	Search Depth		
	2	3	4
Substructure	0.06	4.39	8.86
Pure Substructure	2.55	7.96	
Dominating	0.47	5.14	
Naive	5.22		

Figure 3-4: log of Mean Times (in seconds) for Searching Positions in the Test Suite

Evaluation Algorithm	Search Depth			
	1	2	3	4
Substructure	4.13	8.21	12.77	17.17
Pure Substructure	4.14	10.67	16.11	
Dominating	4.13	8.79	13.57	
Naive	6.91	14.28		

Figure 3-5: log of Mean Leaves Evaluated for Searching Positions in the Test Suite

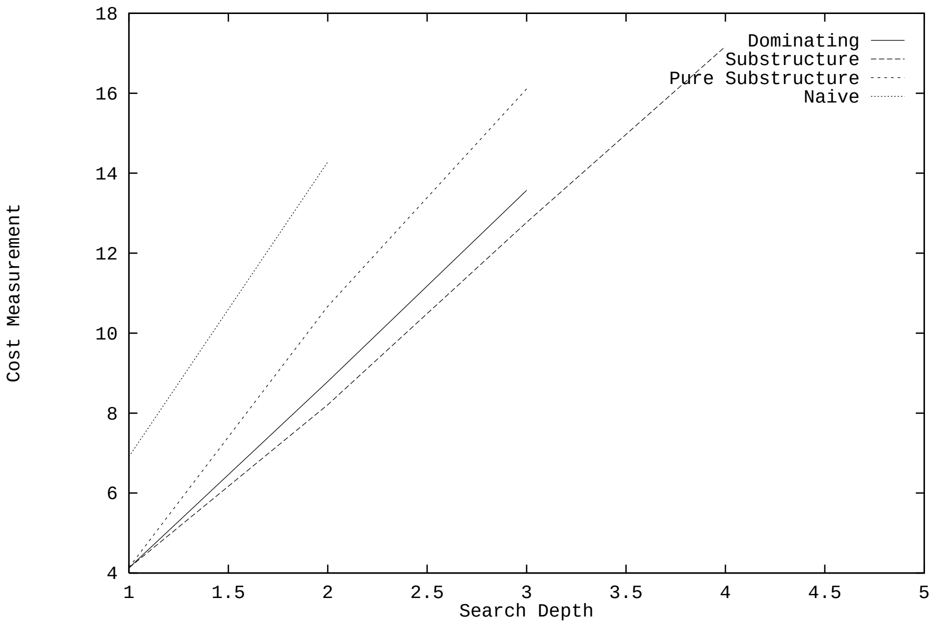


Figure 3-6: log of Mean Leaves Evaluated For Positions in the Test Suite

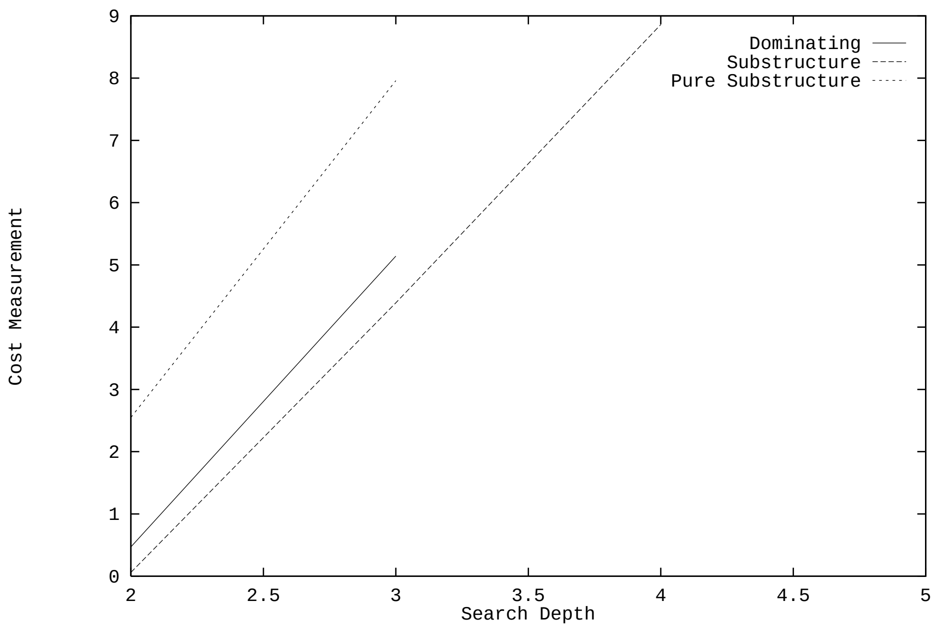


Figure 3-7: log of Mean Search Time (in seconds) For Positions in the Test Suite

search time is roughly proportional to the number of terminal nodes (though the different algorithms had slightly different proportionality constants).

Also, the substructure evaluation algorithm performed significantly better than dominating evaluation algorithm. Substructure evaluation was able to search roughly twice as deeply as naive evaluation, in a given length of time. This is demonstrated by the fact that the natural logarithm of leaf evaluations increased approximately half as quickly for substructure evaluation as for naive evaluation. The number of leaf evaluations performed per second (see figure 3-8) was approximately constant for each algorithm, which suggests that in a given length of time, substructure evaluation should be able to search approximately twice as deeply as naive evaluation.

Evaluation Algorithm	Search Depth		
	2	3	4
Substructure	3446	4334	4065
Pure Substructure	3351	3490	
Dominating	4110	4594	
Naive	8593		

Figure 3-8: Leaf Evaluations per Second when Searching the Test Suite

	Evaluation Algorithm	
Statistic	Substructure	Dominating
Mean Completed Search Depth	2.18	1.95
Frequency of Obtaining Partial Information	0.67	0.00

Figure 3-9: Search Statistics for the Tournament

Tournament Type	Mean Score	Deviation
Substructure vs. Dominating	2.19	7.36
Substructure vs. Depth-1 Substructure	21.44	6.88

Figure 3-10: Outcomes From Tournament Play

Results from Complete Games: For the tournament which pitted substructure evaluation against dominating evaluation, we computed the mean search time, the mean iteration completed and the frequency with which information from an incomplete search was obtained for both algorithms. This appears in figure 3-9.

3.9.3.2 Strength of Program Play

Figure 3-10 indicates the mean and deviation in the final score during tournament play, for the two tournaments. In both cases, a positive value indicates that the first algorithm received better payoffs. The payoff is the expected difference in votes, scaled by a factor of 538 (the number of electoral college votes).

This table demonstrates the effectiveness of using deeper search when playing this matrix-tree game.

Qualitative Observations: The program appears to exhibit strong play. It seems to campaign in regions that are advantageous for it, it responds to opponent threats, it moves at plausible times and it divides the amount of campaigning it performs in multiple regions in a plausible manner. In play against people, the program always obtained a commanding lead by the game's end.

3.9.3.3 Secondary Statistical Information

We give statistical data sampled from the execution of the dominating evaluation and substructure evaluation algorithms (we only tabulate this data for the normal substructure evaluation algorithm, and not for the one that strictly evaluated). We present the data collected from searching the test suite first, then that collected from the tournament in which a player using substructure evaluation played against one using dominating evaluation.

Statistic	Evaluation Algorithm: Substructure	Dominating
Time (in seconds)	6179	2866
Deep Evaluations	866	88
Deep Tests	2954	917
Twig Evaluations	129861	13243
Twig Tests	186629	181759
Leaf Evaluations	25138469	10000545
Move Evaluations	27204050	16030186
Additions to Substructure per Evaluation	0.13	-

Figure 3-11: Mean Values of Statistics for the Test Suite

We tabulated the mean number of recursive calls to test and evaluation procedures and the mean time; in both cases the number is the mean number per (test or game) position searched. We also collected the mean number of times (per evaluation) that a move was added to a substructure in substructure evaluation.

We also tabulated the frequencies of early success and failure in twig tests, the frequency of success in twig tests and in deep tests. We tabulated the frequency with which deep tests tested non-singleton collections of moves, and the frequency of success for those combinations.

We recorded the frequency with which the Test procedure failed to establish a bound that actually held, when Test was called by an evaluation routine. We measured only for those tests that were initiated by an evaluation routine because these are the only tests for which we always subsequently evaluate the actual value of the subgame. We also computed the frequency with which the bounds set by substructure evaluation did not hold, but the move to be bounded was not a strict best reply, i.e., the frequency of bounds failures that could have been prevented by setting different bounds.

Note that the mean values for the test suite display the costs for a depth 4 search using substructure evaluation, and for a depth 3 search using dominating evaluation. However, these statistical tables are intended to demonstrate the relative amounts of time spent by the algorithms performing various tasks, and not to compare performance.

This statistical data demonstrates that our fairly simple test policies were quite successful in this game, though further improvement may be possible. The substructure evaluation algorithm performed quite well, needing to perform very few evaluations—it did not need to grow substructures to a very large amount, and it spent most of its time verifying solutions. This explains the excellent performance of substructure evaluation.

Finally, we tabulated the support sizes of games searched by the naive evaluation algorithm. We determined that 99.77% of these support strategies were singletons, and the remainder were doubletons. This result strengthens our confidence that the solution supports in realistic games will be small.

Statistic	Evaluation Algorithm	
	Substructure	Dominating
Time (in seconds)	74	46
Deep Evaluations	9	4
Deep Tests	40	0
Twig Evaluations	1666	527
Twig Tests	2381	2785
Leaf Evaluations	312339	155368
Move Evaluations	329014	258840
Additions to Substructure per Evaluation	0.12	-

Figure 3-12: Mean Values of Statistics for the Tournament

Statistic	Evaluation Algorithm	
	Substructure	Dominating
Success in Deep Tests	0.9919	0.9915
Success in Twig Tests	0.9902	0.9974
Early Success in Twig Tests	0.9902	0.9974
Early Failure in Twig Tests	0.0098	0.0026
Non-Singleton Collections in Deep Tests	0.0000	0.0000
Success for Non-Singleton Collections in Deep Tests	0.0000	-
Failure to Establish a Test Bound that Held (Eval)	0.0016	0.0174
Bad Bounds for Move (Substructure)	0.0086	-

Figure 3-13: Frequency of Events for the Test Suite

Statistic	Evaluation Algorithm	
	Substructure	Dominating
Success in Deep Tests	0.9202	-
Success in Twig Tests	0.9738	0.9969
Early Success in Twig Tests	0.9738	0.9968
Early Failure in Twig Tests	0.0259	0.0031
Non-Singleton Collections in Deep Tests	0.0001	-
Success for Non-Singleton Collections in Deep Tests	0.0000	-
Failure to Establish a Test Bound that Held (Eval)	0.0075	0.0619
Bad Bounds for Move (Substructure)	0.0371	-

Figure 3-14: Frequency of Events for the Tournament

3.9.4 Conclusions

We believe that the better of the two algorithms for evaluating two-player zero-sum simultaneous play games is substructure evaluation. It improves on the performance of naive search approximately as much as α-β improves on that of naive search. The problem with dominating evaluation is that it evaluates irrelevant information and can even force evaluation based on unimportant results (like a slight improvement in exploiting a terrible blunder by one's opponent). However, even dominating evaluation exhibits a significant improvement over naive evaluation.

The results from our test games suggest that our algorithms should both prove effective in solving real games. These results also improve our confidence in the policies selected. For those aspects of the policies that we measured, we have seen that the performance was quite good. Overall, the most important result regarding the policies we used was that they were effective in dramatically reducing search for our case study.

3.10 Modifications to Allow Chance Alternatives

To this author's knowledge, no one has published an algorithm for searching 2ZPI games with chance occurrences that is more efficient than a simple back-up of the values from all the leaves. The specific problem is to allow cutoffs of moves at a vertex that follow a chance vertex. However, we can make a natural extension of our approach for handling simultaneous play games to handle chance in an efficient manner. We will first describe how this can be done for simultaneous play games, and then examine a simplification. This simplification can be considered a modification of SCOUT to handle 2ZPI games with chance.

To allow for chance, we need to extend the algorithm to having each pair of moves result in an *outcome*, instead of a unique subgame. An *outcome* is the expected value of the solutions to the subgames that can result from a given pair of moves, taking the expectation over the probability distribution for the chance alternatives at the vertex. We shall consider how one must modify evaluation and testing procedures when each pair of moves leads to an outcome, instead of a unique subgame.

Heuristic Values: We adopt the approach of modelling each outcome's value as a single random variable. Each subgame that can result from a single outcome is modelled as an independent normal random variable, as before.[11] However, the value of the outcome is a linear combination of these independent normals, so it is also a normal variable. In particular, if $a(V)_0 = k$, Y is the random variable of an outcome generated by the subgames $X_i \sim N(\mu_i, \sigma_i^2)$, and the probability of X_i occurring is ϕ_i, then $Y \sim N(\sum_{i=1}^{k} \phi_i \mu_i, \sum_{i=1}^{k} \phi_i^2 \sigma_i^2)$.

Incorporating Chance into Policies: It is quite straightforward to handle chance outcomes in our policies by treating outcomes as we would treat single subgames. Whenever we would evaluate a subgame, we evaluate each of the outcomes, and determine the value

[11] We make an exception for identical subgames. These are merged, so that the probability of such a subgame is the sum of the probability of all chance alternatives that lead to the given outcome.

by taking the linear combination of the weights given by the probability distribution. For the case where we would test a subgame, we need to perform an additional computation to set weights on the individual subgames in the outcome. The computation of bounds is performed exactly like the bounds computation performed when setting bounds on a single row in the course of testing a upper bound and when setting bounds for substructure evaluation (the procedure is presented in section 3.4.1.2). In both cases, one is setting bounds on a (fixed) linear combination of independent normal variables. When we set bounds for substructure evaluation and for single rows (or columns) in Test-$\geq$ (or Test-$\leq$), we can set better bounds by doing a single optimization. In these cases, we have a linear combination for moves, so we optimize a linear combination of all the possible subgames. The probability for a single subgame is simply the product of the linear weight for the move that leads to it and the probability for the chance alternative that leads to it. If multiple chance alternatives can lead to the same subgame, we again obtain the total probability by summation.

3.10.1 Specializing to Games of Perfect Information

In searching 2ZPI games with chance occurrences, one can clearly prune vertices at which a player moves if their parent vertex is also one at which a player moves (for example, one can pass cutoff information like in α-β). Unfortunately, it is not possible to pass such bounds information through chance vertices. However, we have developed an algorithm that allows us to prune at player vertices even when there are chance vertices intervening between player vertices.

The following modification of SCOUT improves search in game trees with chance by reducing the effective branching factor for player moves. This modification is effectively just a specialization of either of the evaluation algorithms for simultaneous play[12], with the modifications for chance presented above. In particular, this test function no longer provides a guarantee that it will return true if a bound holds, though if it does return true, then the bound must hold. The notation used for these algorithms is that for extensive form games, and not that for matrix tree games.

As in SCOUT, we have an evaluation function and two test functions. The evaluation procedure, CSCOUT-EVAL, is as follows.

[12] It can be considered a specialization of either algorithm, because at a stage where one player has a single move, both algorithms act in the same manner.

```
Function CSCOUT-EVAL(vertex V)
(It returns a value.)

if V ∈ τ then return u_1(V)
else
  if a(V) = 0 then
    b ← 0
    for i = 1 to c(V)
      b ← b + d_V(i)CSCOUT-EVAL(V_1)
  else
    b ← CSCOUT-EVAL(V_1)
    for i = 2 to c(V)
      if a(V) = 1 and not CSCOUT-TEST-≤ (V_i, b) or
         a(V) = 2 and not CSCOUT-TEST-≥ (V_i, b)
        b ← CSCOUT-EVAL(V_i)
  return b
```

The test function CSCOUT-TEST-$\geq$ is given as follows, and CSCOUT-TEST-$\leq$ is just its dual.

```
Function CSCOUT-TEST-≥ (vertex V, b)
(It returns a boolean.)

if V ∈ τ return u_1(V) ≥ b
else
  if a(V) = 0
    compute bounds z_1, ..., z_c(V) such that Σ_{i=1}^n d_V(i)z_i = b
    for i = 1 to c(V)
      if not CSCOUT-TEST-≥ (V_i, z_i)
        return false
    return true
  else
    if a(V) = 1
      for i = 1 to c(V)
        if CSCOUT-TEST-≥ (V_i, b)
          return true
      return false
    else (for player 2)
      for i = 1 to c(V)
        if not CSCOUT-TEST-≥ (V_i, b)
          return false
      return true
```

To set the bounds, we use the same approach as for setting bounds in simultaneous play games when we have a fixed linear combination. That is, we assume the value of the successors are independent random variables, so $u_1(V_i)$ is distributed as $N(\mu_i, \sigma_i^2)$. We then compute the most probable bounds in the same manner as for setting bounds on a single row when testing an upper bound.

As with SCOUT, one can improve performance by slightly increasing the algorithm's complexity. One idea is to combine test and evaluation when testing a chance vertex that is an (immediate) successor of a vertex that we are evaluating. This enables the use of evaluation when bounds fail (preventing the need to evaluate all the other children if the bound actually holds). Also, if one incorporates the idea of returning reference values (so that one returns a tighter bound if it is met), then chance vertices can loosen subsequent bounds if a tighter bound was returned by a test.

Now, let us compute the best case performance in a tree of depth $d = 4k$ where each player vertex has branching factor b and each chance vertex has branching factor c. For this tree, all vertices on the same level have the same agent, and the agents on the levels are $1, 0, 2, 0$, repeated k times. So levels alternate between chance vertices and player vertices, and the levels for player vertices alternate between players. This is illustrated in figure 3-15.

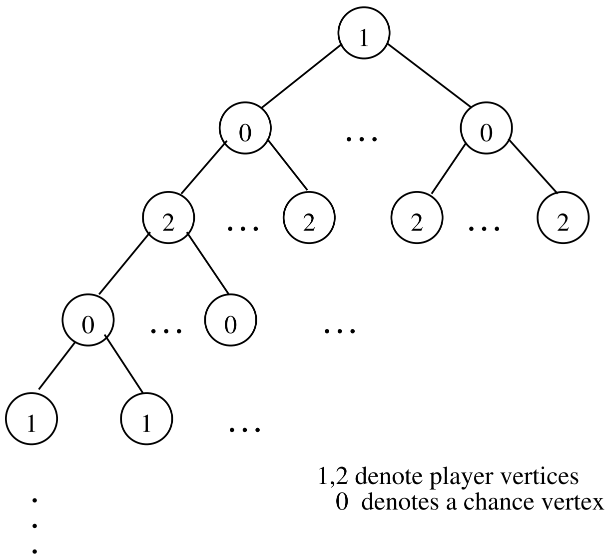


Figure 3-15: Assignment of Agents in the Analyzed Game Tree Structure

For the best case, we assume the tree is best-ordered at player vertices. This means that any test we perform from a player vertex can succeed. We also assume that if a chance vertex is tested against a bound that holds on its value then we will set bounds on the successors that can all be met. Our assumptions imply that every test performed will succeed.

We can also deduce that whenever we evaluate a player vertex, we evaluate the first child and we only test the other $b - 1$ children. Also, tests at player vertices at which an early cutoff occurs only recursively test one child. Tests at player vertices at which early cutoff does not occur recursively test b children. Evaluations at chance vertices evaluate all c successors. Tests at chance vertices test all c successors.

The pattern for tests is "extended through" chance vertices: the test at the next player vertex after the chance vertex will be cutoff early if and only if the test at the current player vertex will not. The reason for this is that CSCOUT-TEST-$\leq$ calls only CSCOUT-TEST-$\leq$ and CSCOUT-TEST-$\geq$ calls only CSCOUT-TEST-$\geq$. But when these tests succeed, they cut off early only for player 1 (for CSCOUT-TEST-$<$) and for player 2 (for CSCOUT-TEST-$\geq$). But the players alternate on player vertex levels in the tree, so we conclude this test pattern holds.

Also, the tests at the next player vertex after an evaluation will all cutoff early, because the test performed is one that the following player can establish on a single move.

We define several functions that give the number of terminal evaluations used in performing an operation on a depth n tree:

- f gives the cost for evaluation when the root is a player vertex.
- f' gives the cost for evaluation when the root is a chance vertex.
- g_w gives the cost for a test when the root is a player vertex, and the test is cut off early.
- g_l gives the cost for a test when the root is a player vertex, and the test is not cut off early.
- g'_w gives the cost for a test when the root is a chance vertex, and the tests at the children vertices will be cut off early.
- g'_l gives the cost for a test when the root is a chance vertex, and the tests at the children vertices will not be cut off early.

We have the base conditions:

$$\begin{aligned} f(0) &= 1 \\ f'(0) &= 1 \\ g_w(0) &= 1 \\ g'_w(0) &= 1 \\ g_l(0) &= 1 \\ g'_l(0) &= 1 \end{aligned}$$

For $n > 0$, we have:

$$\begin{aligned} f(n) &= f'(n-1) + (b-1)g'_w(n-1) \\ f'(n) &= cf(n-1) \\ g_w(n) &= g'_l(n-1) \\ g'_w(n) &= cg_w(n-1) \\ g_l(n) &= bg'_w(n-1) \\ g'_l(n) &= cg_l(n-1) \end{aligned}$$

Folding the cost of chance vertices into player vertices yields, for $n > 1$:

$$\begin{aligned} f(n) &= cf(n-2) + c(b-1)g_w(n-2) \\ g_w(n) &= cg_l(n-2) \\ g_l(n) &= cbg_w(n-2) \end{aligned}$$

Solving for g_w yields, by substitution:

$$\begin{aligned} g_w(2) &= c \\ g_w(n) &= c^2 b g_w(n-4) \text{ for } n > 3 \end{aligned}$$

So we have

$$g_w(2k) = c^k b^{\lfloor k/2 \rfloor}$$

Hence,

$$\begin{aligned} f(2k) &= cf(2k-2) + c^k(b-1)b^{\lfloor (k-1)/2 \rfloor} \\ f(4k) &= c^2 f(4k-4) + c^{2k}(b-1)b^{k-1} \\ &= c^{2k}\left(1 + 2\left(b^k - 1\right)\right) \end{aligned}$$

Therefore,

$$R_{\text{CSCOUT}} = c^{1/2} b^{1/4}$$

As a final observation, we note that searching efficiently in a game with chance is quite like searching simultaneous play games efficiently. We believe that handling chance alternatives requires the use of something like a test procedure, since there is no specific bound that one can set on single chance outcomes. The discussion on this topic contained in section 3.3.1 is equally applicable in this context.

3.10.2 Non-Speculative Forward Pruning With Chance

Non-speculative forward pruning can also be used to expedite search of chance alternatives. If one can bound the difference in payoffs between various chance alternatives, it is possible to search only one or some, and to use the bound on the change to bound the overall outcome of a given pair of moves. A similar idea can be used to handle improbable events, by simply bounding the absolute impact of the event.

Chapter 4

N-Player, Sequential Games

In this chapter we consider algorithms for playing N-player games with perfect information (i.e., sequential play). We employ the extensive form description of a game, as presented in chapter 2. We do not allow for chance alternatives in the games examined in this chapter. It is straightforward to handle chance naively: evaluating a chance vertex by solving each successor and taking the expected value in accordance with the chance distribution. However, as we noted in section 3.10, it appears that the only approach to efficiently handling games with chance is to use a test-like approach, similar to that of SCOUT. Such an approach appears impractical for this class of games when $N > 2$ or when payoffs are not constant-sum (this is discussed in section 4.2.4). In light of this, we consider only games without chance in this chapter.

N-player games of perfect information have sequential equilibria (i.e., profiles which, when restricted to any subgame, are also equilibria in the subgame). These equilibria can be determined by using back-induction. Hence, we can extend the payoff function u to the internal vertices in the tree by back-induction (as we did in chapter 3). However, there need not be a unique sequential equilibrium. Non-uniqueness can occur whenever a player receives the same payoff from two different moves. Unlike in a two-player zero-sum game, the payoffs of other players may differ in this case. The assumption of rationality does not specify how the other players will behave when given alternatives towards which they are indifferent. In the absence of additional information, we should assume that the other players are equally likely to play any move which gives them the same payoff. We defer discussion of this complication to section 4.5, since it is related to the issue of searching approximation games using static evaluation. For now, we presume that all the players have a total ordering over outcomes. As such, we assume that if two payoffs have the same value in one coordinate, they must have the same value in all coordinates. This is a sufficient condition for there being a unique payoff value for the game.

We present a naive algorithm to solve an N-player game of perfect information (and compute the payoff value for the root):

```
Function Naive-Solution(vertex V)
(It returns a payoff vector and a principal variation).

if V ∈ τ return (u(V), Λ)
else
  (v,q) ← Naive-Solution(V_1)
  p ← (1,q)
  for i = 2 to c(V)
    (w,q) ← Naive-Solution(V_i)
    if w_a(V) > v_a(V)
      v ← w
      p ← (i,q)
  return (v,p)
```

If different payoffs may have the same payoff in one coordinate, then this algorithm computes only one sequential equilibrium.

4.1 Developing a More Efficient Algorithm

4.1.1 Bounds on Payoff Sums

In order to determine a more efficient algorithm for solving such games, we deal with those games that have the additional property of having (known) bounds on payoff sums. All games have such bounds, since their game trees are finite (and have real valued payoffs).[1] However, if the only way to determine the bounds is to examine all the terminal vertices, then one cannot use this information to expedite search.

In the most general form, we have for each player i, payoffs in the interval $[a_i, b_i]$. We can apply a positive affine transformation to each player's utilities (since utility functions have ordinal range, and ordinals are defined only up to such transformations). So, without loss of generality, we can take the interval to be $[0, b_i]$. We consider bounds on the sum of payoffs. In the simplest case we have real numbers m, M such that for each terminal vertex V, $m \leq \sum_{i=1}^{N} u_i(V) \leq M$. More generally, we have bounds m_I, M_I for each subset of players, I such that for each terminal V, $m_I \leq \sum_{i \in I} u_i(V) \leq M_I$.

In addition to this, if a player i can receive a zero payoff, we need to know the payoff vector z_i for this payoff (it is unique, by our earlier assumption). If player i cannot receive a zero payoff, we set $z_i = 0$ by convention. If the value of z_i is not known, we use a positive affine transform to rescale the player i's payoffs to fit within the interval $[\epsilon, b_i]$ and take $z_i = 0$ (since player i can no longer receive a zero payoff). Such rescaling requires (slightly) different bounds. ϵ should be sufficiently small that performance is not changed (it certainly should be smaller than the smallest difference between (rescaled) payoff values for any player).

[1] Numerous authors make the assumption that the value of a win in a two-player zero-sum game is ∞, for example, Knuth and Moore[KM75]. Since we are dealing with utility functions this assumption seems unwarranted.

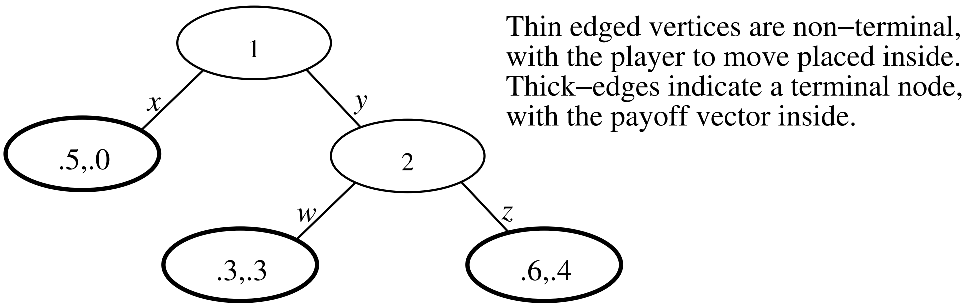


Figure 4-1: An Example of How the Lack of Total Opposition Allows Mutual Benefit

We initially consider a simpler case, where $b_i = 1$ for each player i, and we have only global m and M values (i.e., bounds on the sum of payoffs of all the players).

4.1.2 Possibilities for Pruning

The major factor that distinguishes such games from two-player constant-sum (i.e., zero-sum) games is the lack of total opposition. In particular, it is possible for a sequence of players to all profit by selecting a given sequence of moves. For example, consider the two player game in figure 4-1 where the payoff bounds are $m = .5$ and $M = 1$.

Note that player 1 prefers move y to x, and that player 2 prefers move z to w, so both players prefer the variation (y, z) over either of the other variations.

This property reduces the possibilities for pruning in the search. Indeed, if $M = N$ and for all i $b_i = 1$, but no player ever receives a payoff of 1, there can be no pruning: any move could lead to an outcome superior for all players to any other outcome. Also, unlike the two player case, in a constant-sum game (e.g., $m = 1$ and $M = 1$), if $N > 2$ the game does not have total opposition; two players can make a "deal" to profit at the expense of the other player(s).

4.1.3 A Branch and Bound Algorithm

A slight improvement on the naive algorithm is to employ a branch and bound technique:

```
Function Branch-and-Bound(vertex V, bound ψ)
(It returns a payoff vector.)

if V ∈ τ return u(V)
else
  let p ≡ a(V)
  v ← z_p
  for i = 1 to c(V)
    w ← Branch-and-Bound(V_i, M − v_p)
    if w_p > v_p
      if v_p ≥ ψ
        return v
      v ← w
  return v
```

As in the background chapter, we do not return a principal variation in this and subsequent algorithms on the topic, for it is simple but notationally inconvenient.

One can use this algorithm to solve for the game rooted at V, by computing Branch-and-Bound$(V, 1)$. This will compute the same solution as the naive algorithm above.

Theorem 4.1.1 *Branch-and-Bound*$(V, b_{a(V)}) = u(V)$

Proof: For $p = a(V)$, we claim the following conditions on the value returned hold:

$$\begin{array}{rcll} \text{Branch-and-Bound}(V, \psi) & = & \text{u}(V) & \text{if } u(V)_p < \psi \\ \text{Branch-and-Bound}(V, \psi)_p & \geq & b & \text{if } u(V)_p \geq \psi \end{array}$$

These condition hold when V is terminal. Let us verify these conditions by induction on tree height. We consider an internal vertex. At the beginning of the ith iteration of the `for` loop, we have the invariant conditions

$$v_p = \max\left(0, u_p\left(V_1\right), \ldots, u_p\left(V_{i-1}\right)\right)$$

And for $i > 1$,

$$v = u(V_{\operatorname{argmax}_{j \in \{1,\ldots,i-1\}} u_p(V_j)})$$

To verify that these are invariant, let us consider the behavior when examining the ith successor. If $u_p(V_i) > v_p$, then $u_{a(V_i)}(V_i) < M - v_p$, and

$$\text{Branch-and-Bound}(V_i, M - v_p) = u(V_i)$$

by the conditions above. If $\max\left(0, u_p\left(V_1\right), \ldots, u_p\left(V_i\right)\right) \geq b$, then $u_p(V) \geq b$ and search is cutoff, returning a payoff vector with the appropriate constraint (and the `for` loop does not continue for this vertex). Otherwise,

$$v_p = \max\left(0, u_p\left(V_1\right), \ldots, u_p\left(V_i\right)\right)$$

remains true (whether $u_p\left(V_i\right) > v_p$ or not). If $v_p = 0$, then $v = z_i$ and in the other case v

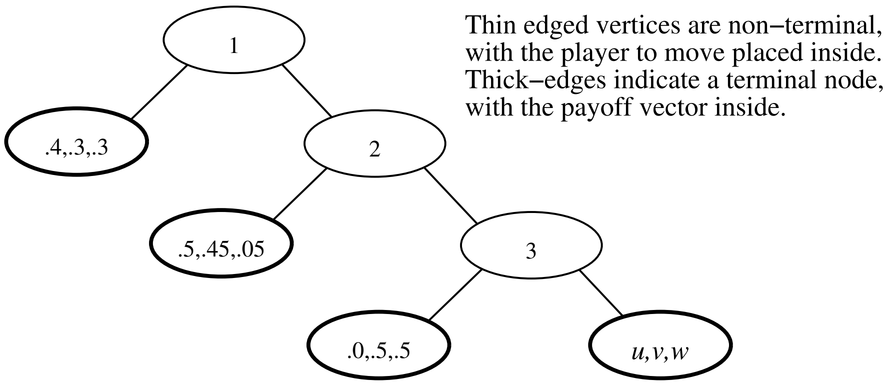


Figure 4-2: Refutation In N-player Games

is straightforwardly maintained. So the invariant on v holds in all cases.

If the algorithm searches all the successors without a cutoff, then it returns v, which is exactly $u(V)$ by the invariant condition. So the given conditions holds by induction. $\square$

Note that we can employ more general subset bounds information for Branch-and-Bound by replacing $M - v_p$ with $M_{\{p,a(V_i)\}} - v_p$.

One cannot go further and directly extend the approach of α-β. Suppose one were to pass a vector of lower bounds (one for each player), α, in a fashion similar to α-β (i.e., passing α', where $\alpha'_i = \alpha_i$ for $i \neq p$, and $\alpha'_p = \max(\alpha_p, v_p)$). If $u(V)_p > M - \sum_{i \neq p} \alpha_i$, then it is true that *some* player will not wish to make a move that leads to V (since not all players can achieve their floor value). So the given position will not be reached by best play. However, it is of critical importance to determine which move will not be made.[2] Consider the case in figure 4-2.

For this game, $m = 1$ and $M = 1$. Now, after examining the first successor to the vertex at which player three moves, we know that either player 1 or player 2 will not make a move that leads to that vertex, for $.5 > 1 - (.4 + .45)$. However, the values u, v, w can have an impact on the game. If $v > .45$ or $w < .5$, then the solution to this game is $(.4, .3, .3)$. If $v < .45$ and $w > .5$, the solution is $(.5, .45, .05)$.

4.2 Evaluation Using Deep Cutoff Information

We can attempt to use deep cutoff information by searching a vertex's first move, and if its predecessor's floor payoff is not met on this move, then we can evaluate subsequent moves, but checking only whether both floors are met; if we discover a move whose payoff meets both floors, then some moves may need to be re-searched. However, if no such move is discovered, the predecessor's move that led to this position is refuted. We now present a

[2]Even in the two player case this can occur if the payoff sum is not constant.

generalization of this concept, including a provision for passing a ceiling for one player, so that if this position ensures him this ceiling, further information is not required.

This concept is implemented in a pair of mutually recursive functions, Unimodal-Eval and Bimodal-Eval, that are used to solve games using deep cutoff information. The search is somewhat like that performed by α-β with windowing. That is, we provide various bounds on the payoff values when searching, and allow cutoffs by assuming these hold. If they do not, we sometimes need to re-search. In a sense, this is a method of testing values but it is much closer to the kind of testing that is done by a windowing procedure than that done in SCOUT since it evaluates for payoffs within a range of values. This solution algorithm is expected to perform quite well on best-ordered trees (i.e., trees wherein moves are ordered so that the first move gives a player the maximum payoff value he can receive at the vertex). We supply the most general bounds formulation with bounds on each subset of players. We discuss how to handle bounds (including the case of having only global bounds) below.

Also, for simplicity, we assume that there is a payoff 0 (so for each i, $z_i = 0$). If this is not true (e.g., if $m > 0$), then it is necessary to use a positive affine transformation to rescale the payoffs for each player, i, to fit within $[\epsilon, b_i]$ (with an appropriate adjustment of bounds), as discussed earlier. We write e_k to denote an N-dimensional vector x such that $x_i = \delta_{ik}$. Both Unimodal-Eval and Bimodal-Eval return a payoff vector and a boolean, where the boolean indicates whether a cutoff was made or not (i.e., that the payoff information might not be exact).

Function Unimodal-Eval(vertex V, lower bounds α, upper bound β,
player j)
(It returns a payoff vector and a boolean.)

if $V \in \tau$ then return $(u(V), \texttt{false})$
else
  let $p \equiv a(V)$
  $f \leftarrow \alpha_p$; $c \leftarrow \max_{I \supset \{p\}} \left(M_I - \sum_{i \in I - \{p\}} \alpha_i \right)$; info $\leftarrow$ false
  if $j = p$ then $c \leftarrow \min(c, \beta)$
  else $f \leftarrow \max(f, m_{\{j,p\}} - \beta)$
  $\alpha^a \leftarrow \alpha$; $\alpha^a_p \leftarrow f$; $\alpha^i \leftarrow f e_p$; $l \leftarrow \Lambda$; ancestral $\leftarrow$ false
  for $i = 1$ to $c(V)$
    if ancestral then $(w, x) \leftarrow$ Unimodal-Eval$(V_i, \alpha^a, \beta, j)$
    else $(w, x) \leftarrow$ Bimodal-Eval(V_i, α^i, c, p)
    if $w_p > f$ then
      if $w_p \geq c$ then return $(0, \texttt{true})$
      info $\leftarrow$ true ; $v \leftarrow w$; $f \leftarrow v_p$
      if $\beta > w_j$ and for each $i \in \{1, \ldots, N\}, \alpha_i < w_i$ then
        ancestral $\leftarrow$ false ; $\alpha^i_p \leftarrow f$
      else
        ancestral $\leftarrow$ true ; $\alpha^a_p \leftarrow f$
    else if ancestral and x then adjoin i to the end of l
  if ¬ancestral then
    if ¬info then return $(0, \texttt{true})$
    $\alpha^i_p \leftarrow f - \epsilon$
    for i in l
      $(w, x) \leftarrow$ Bimodal-Eval(V_i, α^i, f, p)
      if $w_p \geq f$ then return $(0, \texttt{true})$
    return $(v, \texttt{false})$
  else
    if $l = \Lambda$ then return $(v, \texttt{false})$
    else return $(0, \texttt{true})$

Function Bimodal-Eval(vertex V, lower bounds α, upper bound β,
player j)
(It returns a payoff vector and a boolean.)

if $V \in \tau$ then return $(u(V),$ false$)$
else
  let $p \equiv a(V)$
  $f \leftarrow \alpha_p$; $c \leftarrow \max_{I \supset \{p\}} \left(M_I - \sum_{i \in I - \{p\}} \alpha_i \right)$; info $\leftarrow$ false
  $v \leftarrow 0$; $\psi \leftarrow 0$
  search-type $\leftarrow$ "open"
  if $j = p$ then
    if $c < \beta$ then
      $c \leftarrow \beta$; $\psi \leftarrow \beta e_j$
    search-type $\leftarrow$ "low"
  else if $f \leq m_{\{j,p\}} - \beta$ then
    $f \leftarrow m_{\{j,p\}} - \beta$; $v \leftarrow \beta e_j$
  $\alpha^a \leftarrow \alpha$; $\alpha^a_p \leftarrow f$; $\alpha^i \leftarrow f e_p$; $l \leftarrow \Lambda$; ancestral $\leftarrow$ false
  for $i = 1$ to $c(V)$
    if ancestral then
      if search-type="low" then $(w, x) \leftarrow$ Bimodal-Eval(V_i, α^a, c, p)
      else $(w, x) \leftarrow$ Unimodal-Eval$(V_i, \alpha^i, \beta, j)$
    else $(w, x) \leftarrow$ Bimodal-Eval(V_i, α^i, c, p)
    if $w_p > f$ then
      if $w_p \geq c$ then return $(\psi,$ true$)$
      info $\leftarrow$ true ; $v \leftarrow w$; $f \leftarrow w_p$
      $\alpha^i_p \leftarrow f$; $\alpha^a_p \leftarrow f$
      update search type (described later)
    else if ancestral and x then adjoin i to the end of l
  if search-type="high" then $\delta \leftarrow \beta e_j$
  else $\delta \leftarrow 0$
  if $\neg$ancestral then
    if $\neg$info then return $(v,$ true$)$
    $\alpha^i_p \leftarrow f - \epsilon$
    for i in l
      $(w, x) \leftarrow$ Bimodal-Eval(V_i, α^i, f, p)
      if $w_p \geq f$ then return $(\delta,$ true$)$
    return $(v,$ false$)$
  else
    if $l = \Lambda$ then return $(v,$ false$)$
    else return $(\delta,$ true$)$

where `update search type` means:

```
if search-type="open" then
  if w_j ≤ α_j then
    search-type ← "low"
    ancestral ← true
  else if w_j ≥ β then
    search-type ← "high"
    ancestral ← true
else if search-type="low" then
  ancestral ← ¬∀i ∈ {1,...,N} α_i < w_i
else if j ≠ p and f > M_{j,p} − β
  l ← Λ
  search-type ← "low"
  ancestral ← ¬∀i ∈ {1,...,N} α_i < w_i
else
  ancestral ← w_j ≥ β
```

4.2.1 Description and Proof of Correctness

Informally, Unimodal-Eval returns a cutoff only if one of the player's floors (α) is not met, or if player j's value is at least β. If it doesn't return a cutoff, it returns the payoff value. Bimodal-Eval returns a high cutoff (βe_j) only if player j's value is at least β. It returns a low cutoff (0) only if one of the player's floors (α) is not met. If it doesn't return a cutoff, it returns the payoff value. Bimodal-Eval may be allowed to return a low or a high cutoff, in which case it could return either (or an exact value). Both evaluation functions can return exact values even if they are permitted to return a cutoff. So calling Bimodal-Eval$(V, 0, b_{a(V)}, a(V))$ will always return the payoff value of V.

We will now prove this description by induction. In the base case, each algorithm returns an exact value, so the statement is true.

Unimodal-Eval: For Unimodal-Eval, the initial setting of f is such that if the player p does not receive at least this value, then a cutoff must result (either because $u_p(V) < \alpha_p$ or because $u_p(V) < f$ implies $u_j(V) \geq \beta$). The initial value for c is such that if the player receives this value, then a cutoff can immediately be returned (either because some player cannot receive his payoff floor or because $j = p$ and the required β bound has been met). The flag `info` indicates whether any moves currently searched give the player a better payoff than the initial value of f, i.e., whether f is the initial value or the value to the player of some move.

Unimodal-Eval performs two kinds of search, determined by the truth of `ancestral`. If `ancestral` is set, then the value, v, of the best of the moves that have currently been searched (and that did not return a cutoff, i.e., those moves not stored in l) is such that it would generate a cutoff if $u(V) = v$. Those moves in l all would generate a cutoff if best (and might give the player a better payoff than v_p).

When `ancestral` holds, the program searches moves to determine whether they could be the best and if so, would not permit a cutoff to be returned. This is done by calling Unimodal-Eval with the inherited cutoff information (but using a payoff floor of f for p). The point is that unless such a move has a value for p of at least f and the conditions for a cutoff at this vertex do not hold for it, it either is not the best move or it permits a cutoff.

Otherwise, moves are searched bimodally to determine payoff values within the window (f, c) (of values for p).[3] Those moves with value at most f for the player are of no interest; if such a move is the best then a cutoff can be returned (and we need not determine which is best). Moves with a value of at least c (for the player) will generate an immediate cutoff, as explained earlier.

We consider the case where a new best known move is discovered. If the value for p is sufficient to cause a cutoff, a cutoff is immediately returned. Otherwise, the move could not have been caused by a cutoff (a cutoff either returns ce_p, which suffices to cause a cutoff, or it returns a value of 0, which cannot be the best). In this case, v, f and player p's coordinate in the relevant α are set. Also, `ancestral` is set according to whether a cutoff would be permitted if this were the best move available.

If `ancestral` is true and the move was cutoff, the move is put on the list l so that any moves that might be better than the best known move are stored.

Let us consider how the algorithm behaves once all the moves have been searched, if an early cutoff was not made. If `ancestral` is true, the best move must generate a cutoff (since the only moves that might give a better payoff than v are in l, and if one is better it would also generate a cutoff). If l is empty, the best value is v (since no moves might produce a better payoff) and this is returned. Otherwise, a cutoff is returned.

We consider the case where `ancestral` does not hold. If `info` does not hold, then no moves had a value (for the player) of at least f (its initial value) and a cutoff can be returned. Otherwise, the moves in l are all checked to determine whether they achieve a better payoff than the best known move, by calling Bimodal-Eval with a payoff window of $(f - \epsilon, f)$. Moves with value below f (for the player) are of no interest since the best move not in l gives the player a payoff of f. If any move has a value of at least f an immediate cutoff can be returned, since the best move must be in l in this case. If none of the moves in l were better, then v is the value of the game (since `info` does not hold, v is the payoff for the best move).

Bimodal-Eval: Bimodal-Eval is slightly more complicated, but similar. In this algorithm, there is a need to discriminate between high and low cutoffs. c is set to a value that will guarantee a cutoff; if $j = p$ and $\beta = c$, it is a high cutoff else it is a low one. ψ stores the cutoff value that should be returned if $u_p(V) \geq c$. f is set to a value below which a cutoff will automatically be returned, if $j \neq p$ and $f = m_{\{j,p\}} - \beta$ then it is a high cutoff, else it is a low one. v is initialized to the cutoff value we need to return if no move has a value of at least f. `search-type` determines whether the evaluation function is trying to establish a high cutoff (if equal to `"high"`), a low cutoff (if equal to `"low"`) or has not yet decided which (if equal to `"open"`). `ancestral` can be true only if `search-type`$\neq$ `"open"`,

[3] Note that when Bimodal-Eval is called using such a window (i.e., the only lower bound is that for the player for whom the upper bound is passed), its cutoffs are never ambiguous.

and it indicates whether or not the relevant type of cutoff would be generated by the best known move (with value v). l stores those moves that generate the relevant type of cutoff (which might be better for p than v). The flag `info` indicates whether any moves currently searched give the player a better payoff than the initial value of f, as in Unimodal-Eval.

During the loop, if `ancestral` holds then the relevant type of cutoff is tested. For a high cutoff, only p's floor and the upper bound for j, β, are passed to Unimodal-Eval. For a low cutoff, all the floors and p's ceiling (c) are passed to Bimodal-Eval. If `ancestral` is not set, the window (f, c) (on p's value) is tested.

If a move is best and exceeds c, then a cutoff (of the type stored in ψ) is immediately returned. Otherwise, v, f and the player's coordinates in the α's that are passed to children are set. Also, the search type is set. If it was open and v would allow one type of cutoff (i.e., if we had $u(V) = v$, we could return that type of cutoff), we set `search-type` for this cutoff (giving preference to low cutoffs), and `ancestral` is set to true. If the search type was high, and the value for p is sufficient to guarantee that a high cutoff cannot occur, then the search type is set to low, l is emptied (because any move that allowed a high cutoff could not be the best) and `ancestral` is set according to whether v would permit a (low) cutoff. Otherwise, the search type remains the same and `ancestral` is set according to whether v (if the value for V) would permit the appropriate cutoff.

At the end, if `ancestral` holds, a cutoff (of the appropriate type) can be generated, though if $l = \Lambda$ then v is the exact value and this is returned instead. If `info` is false, then the default value of v (a cutoff) can be returned (i.e., no move met the initial floor f). Otherwise, the moves in l must be re-searched using the window $(f - \epsilon, f)$. If any have a value of at least f, then the cutoff (determined by `search-type`) is returned immediately. If none of the moves has a value of at least f, then v is the exact value and it is returned.

4.2.2 Alternatives in the algorithm

One possible improvement in the algorithm is to return a reference value v. Whenever a cutoff occurs, the largest payoff value that the player to move is guaranteed to obtain is also returned (if f has not increased above its initial setting, then this is 0). If the position needs to be re-searched, the reference value might be sufficient to prevent this. If this minimum value guarantees that the player to move will not receive her floor payoff, then the position can be skipped. If the subgame must be re-searched, then $v - \epsilon$ can be passed as (an additional) lower bound for the player who moves at the subgame. Since this lower bound is guaranteed to be met, the algorithm will still work correctly.

There is also the possibility of being more aggressive and immediately re-searching stored moves if `ancestral` becomes false. That is, whenever `ancestral` becomes false but l is non-null, one can search moves in l using a window of $(f - \epsilon, f)$. If a move does not have a value of at least f, then it can be removed from l (it will not be the best). As soon as one does, then `ancestral` can be set to true (since the best move must be in l) and this search can stop. For Bimodal-Eval, if l becomes empty then the search type should be reset to open. If the move ordering is quite good, re-searching moves immediately in this manner is probably advantageous.

It also seems that the algorithm's performance could be improved by devising a better approach for setting `search-type` during a bimodal search. This is related to another possible improvement. It may be possible to incorporate more deep cutoff information, by passing upper bounds for more than one player (possibly in a chain-like fashion, so that unless we can exclude one upper bound from being met, the others are not considered). However, our simulation results (presented in section 4.6) suggest that the lower bound information gives almost all the benefit in this algorithm. To test this, we used a slightly simpler algorithm, Lower-Eval that uses only lower bounds information. This algorithm is similar to Unimodal-Eval, except that it passes $\beta = b_j$ and it always calls Unimodal-Eval instead of Bimodal-Eval. Obviously, this is best done by not passing β and j and removing steps that are applicable only when the upper bound passed is not b_j. The simulation results are contained in section 4.6.

4.2.3 Handling Subset Bounds

Let us consider the issue of how to handle bounds. First, we note that if only global bounds (m and M) are available, then $m_{\{i,j\}}$ can be replaced by $m - \sum_{k \in \{1,\ldots,N\}-\{i,j\}} b_k$ (this lower bound on the sum of payoffs for i and j is implied by the lower bound on the sum of all the players' payoffs), and $\max_{I \supseteq \{p\}} M_I - \sum_{i \in I-\{p\}} \alpha_i$ can be replaced by $M - \sum_{i \in \{1,\ldots,p-1,p+1,\ldots,N\}} \alpha_i$ (simply taking the only known bound).

We now consider how to efficiently compute the various bounds given for a game for which all the subset bounds are available. The algorithm uses lower bounds information only for doubletons, so any additional lower bounds on sets are useful only for setting such bounds. This can be done in the same way as we set $m_{\{i,j\}}$ from m: we compute

$$m_{\{i,j\}} = \max_{I \supseteq \{i,j\}} m_I - \sum_{k \in I-\{i,j\}} b_k$$

This computation can be done once, before we begin to search the tree. If we set bounds dynamically, lower bounds should be computed only for doubletons, to avoid the need to perform such computations (so that computation of constraints on the lower bounds for larger sets is simplified to deal only with doubletons). Dynamic setting of bounds can be done in a manner that is like non-speculative forward pruning and is discussed in section 4.4.4. For taking the maximum of sets of upper bounds containing a given player, we exclude any sets containing a player i for whom $\alpha_i = 0$, since we should always have $M_J \geq M_I$ for $J \supseteq I$. Now, if N is small, examining the $2^{N-1} - 1$ constraints that strictly contain a given player will not be costly. However, if this examination takes too long, then one must either appeal to some kind of regularity in the bounds structure (so that it is possible to determine what subset will minimize the ceiling) or use a heuristic approach (picking a subset that generates a small, but possibly not minimal, ceiling).

However, we do not expect that most games would actually have 2^N subset bounds that are not trivially inferrable. One needs to store M_I only if, for each superset $J \supset I$, $M_J > M_I$. And one needs to consider only the stored upper bounds, since the other bounds provide no information. We expect that in most practical situations, constraints would lead to upper bounds and would create a number of upper bounds that is a polynomial in N. Also, we speculate that the bounds would have a fairly regular structure, so that one can easily characterize ranges of lower bounds for which different bounds will produce a minimal ceiling.

4.2.4 Further Issues

Note that this algorithm uses depth-first search, allowing for minimal space overhead and quick computation in actual use.

This algorithm is directional on best-ordered trees. The reason is that a move can be re-searched only if, after it is searched, another move is determined to be the best of the moves searched thus far. This is because moves can be re-searched only if placed in l and after searching all the moves, `ancestral` is false. But moves can be placed in l only if `ancestral` is true, and `ancestral`, once true, can become false only if a better move is found. But in a best-ordered tree, the first move searched is the best. If it returns a cutoff, then no further moves are searched. If it does not meet the floor, f, then no move will. Finally, if its exact value is returned, no move will produce a better value for the player. So in all cases, no moves will ever be re-searched. Hence the algorithm must be directional on best-ordered trees. We conjecture that this algorithm is the same as α-β in the case of a two-player constant-sum game.

The performance of the algorithm could be improved by storing a complete game-tree, to expedite re-search. This information can be used to prevent re-search of leaves, and (like reference values) to determine that certain interior vertices do not need to be searched at all. It is also possible that a best-first algorithm, analogous to SSS*, could be employed, although doing a best-first search for various projections of a vector function is much more complicated than doing a best-first search for a scalar function.

One could make an extension of SCOUT to this more general class of perfect information games, but we believe that this would not be an efficient algorithm. One generally needs evaluative information and can not merely test for values. The lack of direct opposition destroys the ability to test one player's score for a given value without obtaining actual values for the other players, i.e., effectively evaluating certain positions.

4.3 Asymptotic Analysis

We now will compute an upper bound on the best case performance of both Branch-and-Bound and evaluation using Bimodal-Eval and Unimodal-Eval. We analyze the case where we have only the global bounds of $M = 1$ and any m (naturally, $0 \leq m \leq M$) and the upper bound on any player's payoff is 1. We also take the tree to be a uniform tree of depth d and branching factor b. We further assume that players never have two moves in a row.[4]

For the best case, we analyze a best-ordered tree. As noted earlier, the (deep cutoff) algorithm performs directionally in this case. In fact, the case we exhibit can be optimally searched by Branch-and-Bound (although this case is extremely unlikely, even for best-ordered trees).

Let us define the payoffs by forward induction. Naturally, for a best-ordered tree, the first child of a vertex has the same payoff value as the vertex itself, and the subsequent children have no larger a payoff value for the player than the payoff for the vertex. There are three types of nodes, only two of which are searched: those that have all their children

[4] One can make an equivalent game tree like this for any perfect information game by absorbing the moves at successors where the player moves again into the parent vertex, though this will alter branching factors.

explored (type 1 nodes), those that have only their first child explored (type 2 nodes) and those that are not searched (type 3 nodes). We shall first define the types of nodes and then prove that these properties hold.

We define the root to be of type 1. We also define the first child of a type 1 node to be of type 1, and the remaining children of a type 1 node to be of type 2. We define the first child of a type 2 node to be of type 1 and the remaining children thereof to be of type 3. All the children of a type 3 node are of type 3.

The payoff for every node (of type 1 or 2) will be of the form

$$\frac{1-w}{N}(1,\ldots,1) + we_k \text{ for } 0 \leq w \leq 1$$

For the root, we take $w = 0$. A type 1 node's value will be the same as that of its parent. For a type 2 node, V, let its parent's payoff value have parameters w', k'. Then for $k = a(V)$ and any w such that $\frac{N(1-w')-1}{N} < w < 1$, the type 2 node will allow a shallow cutoff after exploring its first successor. To avoid concerns about payoffs that are the same in some, but not all, coordinates, we pick w values that have not already been used (using any order consistent with the forward induction) to ensure unique payoffs. This can be done because we always have a range of values available. Type 3 nodes can have any payoff value (as long as they obey the restriction that payoff values cannot be the same in some, but not all, coordinates).

Let us verify the properties of the nodes. For both algorithms, the first children of any node will be searched. Since type 1 nodes are all of this kind, these will all be searched. Now, for Branch-and-Bound, we easily see that type 2 nodes will have only their first child searched: the payoff value for the player moving at such nodes was constructed to generate an immediate cutoff (i.e., if the parent of V's parameter was w', then a cutoff ensues if $u_p(V) > \frac{N(1-w')-1}{N}$, and $u_p(V) \geq w$, for these nodes by construction). So the type 3 nodes will not be searched.

For the deep cutoff algorithm, we note that both types of nodes are always given their parent's lower bound, so type 2 nodes will (again) always generate a cutoff after searching their first move (since the search starts with `ancestral` false, the only bounds passed are a window (f, c) for the player's own payoffs). So the collection of type 1 and 2 nodes is a superset of the nodes searched.

Let us compute an upper bound on the best-case performance. Let $f(n)$ be the number of terminal evaluations performed when examining a type 1 node that roots a subgame of depth n, and $g(n)$ be the like cost for a type 2 node. Then we have, for $n > 0$:

$$\begin{aligned} f(n) &= f(n-1) + (b-1)g(n-1) \\ g(n) &= f(n-1) \end{aligned}$$

Eliminating g, we obtain

$$\begin{aligned} f(0) &= 1 \\ f(1) &= b \end{aligned}$$

$$f(n) = f(n-1) + (b-1)f(n-2) \text{ for } n > 1$$

Solving this recurrence yields:

$$\begin{aligned} f(n) &= \frac{(2b-1+\sqrt{4b-3})(1/2+1/2\sqrt{4b-3})^n}{2\sqrt{4b-3}} + \\ &\quad \frac{(2b-1-\sqrt{4b-3})(1/2-1/2\sqrt{4b-3})^n}{2\sqrt{4b-3}} \\ &\leq \frac{2b-1+\sqrt{4b-3}}{\sqrt{4b-3}}(1/2+1/2\sqrt{4b-3})^n \\ f(n) &\geq (1/2+1/2\sqrt{4b-3})^n \end{aligned}$$

$$\text{So, } R_{\text{Bimodal-Eval}} \leq 1/2 + \sqrt{b - 3/4}$$

$$\text{Also, } R_{\text{Branch-and-Bound}} = 1/2 + \sqrt{b - 3/4}$$

This is very close to the best case time for 2ZPI games of $R = \sqrt{b}$, achieved by α-β and SCOUT.

We do not supply a proof that this is the best case behavior for the algorithms. For Branch-and-Bound it is a best-case, for it is impossible to have a child node cutoff search early and to immediately cutoff search at a node after this, since the child cuts off search only if a move is worse for the player at its parent than a previously explored sibling.

For $N = 2$ and $m = M = 1$ it is not a best case (as conjectured above, this search is the same as an α-β search). Indeed whenever there are lower bounds $m_{j,p}$ better performance can result (for $N = 2$ simply having a global m suffices, though if $m < M$ fewer cutoffs can occur).

This best-case analysis is not very enlightening, because direct refutations require fairly extreme payoffs. As in our example to upper bound the best case, having many direct refutations requires almost every node's payoff give almost all the utility available to a single player. Indeed, this best case can not be achieved for arbitrary game trees even if one reorders the branches.

However, when direct refutations do not occur, our more sophisticated algorithm is able to perform significantly better by use of deep cutoff information. So in order to better gauge the efficacy of the various algorithms, we supply simulation results for best-ordered, semi-ordered and independent, random uniform (and semi-uniform) trees in section 4.6.

4.4 Optimizations

We discuss this topic briefly here; the extension of such techniques from 2ZPI games is more straightforward than for simultaneous play games. As usual, approximation games and iterative deepening can and usually should be used. Storing the principal variation is again a good idea in this context (for move ordering).

4.4.1 Transposition Tables

Transposition tables need to store bimodal and unimodal evaluation information. We propose to store information only for the deepest evaluation done. If the minimax value was determined, it (alone) is stored. Otherwise, the lowest ceiling value (c) that achieved a cut-off for the node is stored, along with any bimodal β values that generated high cutoffs. As usual, the best move at the level is stored, so that if the hashed information is not sufficient to prevent search, the efficiency of search is improved.

Because in the proposed algorithm a failure to establish conditions can require re-search, transposition tables are a fairly important device for reducing the cost of such re-search.

Heuristic evaluations are employed in an analogous fashion to that used in 2ZPI games (except that static evaluation must produce a vector, giving the expected payoff for each player). However, questions about modelling opponents as rational have an impact on this. We discuss this matter in section 4.5.

4.4.2 Quiescence Search

One can also use non-standard measures of depth for N-player games of perfect information. Static feature-oriented extensions are straightforward. Singular extensions can be extended to this case. The concept of singularity is changed so that if a player's payoff for one move is greater than that for all others by a singular margin, then that move is singular. The concept of fail-high singular is extended so that if a player's payoff for a move is greater than the value (to him) of the principal variation by a fail-high singular margin and all the other moves give a value less than that of the principal variation, this move is fail-high singular.

4.4.3 Windowing

Windowing search can also be used for this class of games. We present a simple windowing technique that changes search only at the root. If the root player ($a(A)$) is p, and the previous iteration's value was v, calling bimodal evaluation with $\alpha = (v_p - \lambda)e_p$, $j = p$, $\beta = v_p + \lambda$ allows setting a window $(v_p - \lambda, v_p + \lambda)$. λ is chosen so that failures generally do not occur. If the window fails high, one simply researches with the window $(v_p + \lambda, b_p)$. If it fails low, one researches with the window $(0, v_p - \lambda)$. One can improve on this scheme, by handling high failures by resetting the window about player p's score to $(v_p + \lambda, v_p + \lambda + \epsilon)$ and continuing to search the moves after the first move that failed high. If another move fails high, one then researches the first two (fail high) moves and all subsequent moves using $(v_p + \lambda + \epsilon, b_p)$.

4.4.4 Non-Speculative Forward Pruning

Non-speculative forward pruning is also promising for this class of games. One can make a straightforward extension of the usual application of non-speculative forward pruning, by allowing immediate cutoffs if the value of a position (when searched to the required depth) can be bounded to guarantee a cutoff will occur. In addition to this, one can also dynamically configure upper and lower bounds on subsets by use of a like technique: using information about the heuristic function to bound the payoff available to sets of players. For example, if only two players (j and p) can control some objective that yields a utility of x (either in the next few moves, or in any successor of this game state), then $m_{j,p} \geq x$. Also, $M_{\{1,\ldots,N\}-\{j,p\}} \leq M - x$.

Unlike in simultaneous play games (and like in 2ZPI games), heuristic information is used only for move ordering—it is not used to set bounds. As such, doing a shallow (depth $\pi(d)$) search to estimate position values does not seem to be as important for this class of games. However, this algorithm's performance will definitely be degraded more than α-β's when move ordering is inferior. Hence, more time should be spent ordering moves, and it might be advantageous to use some kind of shallower search (especially in Bimodal-Eval, to prevent testing for a kind of bound that can't be met).

4.5 Rationality Issues

Even in the special case of two-player zero-sum games, there are a number of problems with assuming the rationality of one's opponent. If one has encountered past play that is not rational, the wisdom of maintaining an assumption of rationality is questionable. Moreover, if one does not consider the actual game tree, but rather an approximation thereto, it seems even more unusual to assume the other player(s) are not only rational, but are using this same approximation (and the same approach of searching an approximation). As we noted in chapter 3, the approximation approach also suffers because if play reaches a vertex that was searched less deeply, the other players will be able to examine the situation more closely.

Despite these considerations, in a two-player zero-sum game, there is a certain safety and conservatism in assuming rationality. The player's actions are such that she expects to obtain a minimum value, according to her own model of the situation, no matter what her opponent does.

However, when the game is not two-player and zero-sum, this approach cannot be defended on safety grounds. If some of other players do not "see" a complicated sequence of play that is in their interest (and in the deliberating player's), they may select a line that is far worse for the player. Conversely, they may see a complication that escapes the agent's deliberation. Also, they may simply regard different positions as preferable (for example, they might have a different heuristic estimate of value). Playing against opponents who have different estimations of position values is related to the question of how to play a game of incomplete information, since one does not know what the other players will estimate their payoffs to be.

One approach to these problems is to play a maximin strategy to secure the maximum payoff that the player can ensure by his play. This would be done by assuming that one's opponents are all total adversaries, i.e., one treats the situation as a two-player zero-sum game, with all the other players choosing like a single opponent. However, in most contexts this would lead to abysmal play. It also ignores the important fact that one's opponents have their own goals. Any other approach, however, relies on the other players exhibiting certain behaviors.

While using a maximin strategy is too extreme, we believe that incorporating a degree of risk-aversion into the deliberation is beneficial. A good approach for this is to perturb one's heuristic estimates so that other players' payoffs are reduced by a monotone function of the player's own payoff. For example, one might subtract a small constant times the player's payoff from the payoffs of the other players. This approach will result in a certain degree of risk-aversion because opponents are assumed to harm the player in close decisions. The constant should be set to be large enough to avoid problems from discrepancy in evaluation and differing amounts of foresight, but small enough to avoid missing major opportunities.

A departure from a model of one's opponents as rational is to attempt to construct more flexible models of the other players. This is discussed immediately below.

Finally, we revisit the question of what players will do when faced with moves that have exactly the same payoff to them. If simply assuming that the first encountered move is chosen by the player creates problems, we need to use a risk-averse approach or employ more sophisticated opponent models.

4.5.1 Opponent Modelling

There are two basic sources of uncertainty in the deliberations of a game-playing agent (in addition to any external chance factors in the game). One is related to the fact we are using approximations. As we discussed in section 2.2.4.2, there are several proposals for explicitly representing this uncertainty. The biggest problem with employing such approaches is the need to extend from probability of victory to expected values. The other source of uncertainty is caused by the dispositions and decision procedures of the other agents.

The most natural manner of capturing both kinds of uncertainty is to employ probabilistic models of opponent behavior. These models can incorporate certain (perhaps limited) presumptions of rationality. We concentrate on the uncertainty caused by opponent behavior in this section, though it is probably possible to combine this with the uncertainty based on opponents having additional information when moving. Such a combination would likely rely on ideas that probabilistically account for the uncertainty, such as those mentioned in section 2.2.4.2.

In principle, then, we would like to have a model of behavior that assumes the moves by other players are probabilistically determined. This model would incorporate the additional information that can be considered when selecting additional moves later in the game and also uncertainty about how the agent will actually select moves. Such a model needs to be able to capture observed behavior, and also account for behavior that satisfies certain kinds of rationality.

This approach seems to have the most potential, but it also seems difficult to do properly. In its most ambitious form, one would use various rules (inferred from past play) to generate probability distributions for opponent play. One would then select the move that has the highest expected value. While it might be more difficult to tractably search using arbitrary combinations, the biggest problem is to construct a reasonable framework for learning and modelling choices.

Various researchers have examined the issue of how one might use probability distributions to construct models of opponent reasoning.

Rosenschein and Breese, in [RB89], discuss various kinds of rationality assumptions that an opponent might exhibit and suggest forming modelling opponents by use of a probability distribution over the various types. Gmytrasiewicz *et al.*, in [GDW91], likewise deal with modelling other players' choices probabilistically, and explicitly allow for recursive consideration (so that in modelling players choice's, one may assume they employ a model of the other players).

However, current research in this area has dealt with games in strategic form and therefore does not handle questions of limited lookahead, or learning from specific moves played. Also, analysis of strategic forms for perfect information games takes super-exponential time—for a uniform tree with breadth b and depth d (without chance vertices), there are $1+b+\ldots+b^{d-1}$ interior nodes (and equally many information sets). This means that there are $b^{\frac{b^d-1}{b-1}}$ pure strategy profiles.

We believe that any attempt to extend concepts of opponent modelling to larger interactions would require metareasoning models of one's opponents deliberations. Further research in opponent modelling promises to offer useful alternatives to an assumption of rationality.

Situations involving communication and alliance are largely beyond the scope of this thesis; we only sketch possible approaches to these issues. A practical approach to communication (i.e., one that does not require the program to comprehend a large subset of a natural language) requires its incorporation into the game's framework. That is, one allows moves that communicate various concepts in a fairly simple language. However, in order to use communication effectively, the program must be able to incorporate changes in the (subjective) model of one's opponent suggested by statements like "I will not make move x if you make move y." To properly handle alliances, a combination of communication and opponent modelling is needed.

4.5.2 Evaluating the Rationality Assumption

In the absence of a well-defined theory for modelling opponents, we appear to need to use the rationality assumption (possibly modified to be risk-aversion).

One must judge the rationality assumption as one judges any heuristic: does it appear to produce strong approximate solutions or does it oversimplify and generate poor results? This is a question that needs to be settled by empirical experience. The author believes that there are certainly interactions for which this assumption will prove acceptable. Indeed, past experience with chess and other 2ZPI games suggests that reasonably good selections can be made without subtle probabilistic analysis and opponent modelling. We believe that the structure of perfect information games seems to place a great deal more importance on consideration of consequences and less importance on questions of modelling opponent analysis. Admittedly, one expects that those interactions in which communication is limited are probably the most amenable to good solutions using the assumption of rationality.

4.6 Simulation Results

We present statistics for the number of terminal nodes evaluated for random test trees of several different kinds. We tested uniform trees of breadth b and depth d, with various values for N. In all the trees players made moves in a fixed sequence (so at each vertex on level n, player $n \bmod N$ moved).

We used uniform distributions over payoffs. We always set the bound $M = 1$, and we tested for various values of m. We denote our distributions by U_m, so U_m is uniformly distributed over

$$\left\{ x \in [0,1]^N | m \leq \sum_{i=1}^{N} x_i \leq 1 \right\}$$

We employed four kinds of move ordering in the trees. All four were generated using a "base" random tree, whose leaves have values assigned by the appropriate distribution. One kind of tree we tested was random, so that this is all that we did to create the tree.

We also constructed ordered trees, by reordering moves in descending order of payoffs for the player to move. Finally, we constructed semi-ordered trees. This used an error parameter η, and the moves at each vertex were ordered by descending order of the product of their (backed-up) payoff for the player to move and η^{1-2r}, for r a random number drawn from a uniform distribution on $[0,1]$. We always used $\eta \geq 1$. $\eta = 1$ gives perfect ordering, and larger values give more scrambled orderings. We tested semi-ordered trees with $\eta = 2$ and those with $\eta = 4$. In the limit as η tends to ∞, the ordering is completely random. So we denote random orderings by $\eta = \infty$.

We did not insist that payoff vectors which agreed in one coordinate had to agree in all coordinates, although the probability of having such vectors is quite low. However, the efficiency of search is not appreciably altered if we do have such vectors; we simply compute only one of the sequential equilibria.

We define $I_A(N, F, \eta, b, d)$ to be the expected number of terminal evaluations for algorithm A search a uniform N-player game tree of breadth b and depth d whose terminal values are drawn from the joint distribution F, and with move ordering dictated by the parameter η. We define

$$R_A(N, F, \eta, b) = \lim_{d \to \infty} \left(I_A(N, F, \eta, b, d)^{1/d} \right)$$

4.6.1 Critique of the Model

As in 2ZPI games, the analysis of models with independent random leaf values is a useful tool for approximating behavior on real games, but the results produced may differ significantly from those in real games in which dependencies between the values of terminal is an important feature. It would be worthwhile to investigate what dependencies among leaf values improve or hinder the performance of our algorithm. Another area worthy of future consideration is the impact on performance produced by having only a few possible payoff vectors. We conjecture that this will improve performance, just as searching two-player zero-sum random trees with a discrete distribution of terminal values has a better asymptotic branching factor ($b^{1/2}$) than those with a continuous distribution ($R^*(b)$) as noted by Pearl ([Pea84]).

4.6.2 Results for Trees With Global Bounds and More Than Two Players

We tested performance for $m = 0, 1$, $N = 3, 4, 5$, $b = 2, 3, 6$ and for depths ranging from N to $\log(20000)/\log(b)$ (i.e., $4.3/\log(b)$). For each combination of parameters, we collected the mean number of terminal evaluations, and the deviation in the number of terminal evaluations for 200 trees. We omit the results for the case $N = 5$, for these are similar to the results for $N = 4$ (for each combination of parameters and algorithms, the number of leaves examined was slightly greater in this case).

4.6.2.1 Simulation Data

We present the simulation results.

Algorithm	Search Depth (d)	Number of Players (N)					
		3			4		
		Branching Factor (b)			Branching Factor (b)		
		2	3	6	2	3	6
Branch and Bound	3	2.89	2.86	2.79			
	4	3.89	3.85	3.78	3.99	3.98	3.97
	5	4.86	4.82	4.76	4.98	4.98	4.97
	6	5.84	5.82		5.99	5.98	
	7	6.85	6.82		6.99	6.98	
	8	7.83	7.82		7.99	7.98	
	9	8.83	8.82		8.99	8.98	
	10	9.83			9.99		
	11	10.83			10.99		
	12	11.83			11.99		
	13	12.83			12.99		
	14	13.83			13.99		
Lower-Eval	3	2.79	2.67	2.52			
	4	3.70	3.54	3.35	3.87	3.76	3.59
	5	4.59	4.38	4.12	4.81	4.67	4.46
	6	5.49	5.22		5.76	5.57	
	7	6.39	6.07		6.69	6.47	
	8	7.27	6.91		7.66	7.40	
	9	8.18	7.76		8.62	8.31	
	10	9.08			9.56		
	11	9.98			10.51		
	12	10.87			11.48		
	13	11.76			12.42		
	14	12.67			13.37		
Bimodal and Unimodal	3	2.79	2.67	2.52			
	4	3.70	3.54	3.35	3.87	3.76	3.59
	5	4.59	4.38	4.12	4.81	4.67	4.46
	6	5.49	5.22		5.76	5.57	
	7	6.39	6.06		6.69	6.47	
	8	7.26	6.90		7.66	7.40	
	9	8.17	7.74		8.61	8.31	
	10	9.07			9.56		
	11	9.97			10.51		
	12	10.86			11.47		
	13	11.75			12.42		
	14	12.65			13.37		

Figure 4-3: $\log_b$ of Mean Leaves Examined for $m = 1$, $\eta = 1$

Algorithm	Search Depth (d)	Number of Players (N): 3, Branching Factor (b): 2	3	6	Number of Players (N): 4, Branching Factor (b): 2	3	6
Branch and Bound	3	2.97	2.96	2.94			
	4	3.97	3.95	3.93	4.00	4.00	3.99
	5	4.96	4.94	4.93	5.00	5.00	4.99
	6	5.95	5.95		6.00	6.00	
	7	6.96	6.95		7.00	7.00	
	8	7.96	7.95		8.00	8.00	
	9	8.95	8.95		9.00	9.00	
	10	9.95			10.00		
	11	10.95			11.00		
	12	11.95			12.00		
	13	12.95			13.00		
	14	13.95			14.00		
Lower-Eval	3	2.94	2.91	2.88			
	4	3.92	3.88	3.86	3.97	3.97	3.96
	5	4.89	4.83	4.81	4.96	4.95	4.96
	6	5.84	5.80		5.94	5.94	
	7	6.82	6.76		6.92	6.92	
	8	7.78	7.72		7.92	7.91	
	9	8.75	8.66		8.90	8.91	
	10	9.73			9.89		
	11	10.70			10.88		
	12	11.65			11.86		
	13	12.63			12.85		
	14	13.60			13.84		
Bimodal and Unimodal	3	2.94	2.91	2.85			
	4	3.92	3.86	3.80	3.98	3.95	3.91
	5	4.89	4.81	4.75	4.97	4.93	4.88
	6	5.84	5.77		5.95	5.91	
	7	6.82	6.72		6.94	6.89	
	8	7.78	7.68		7.94	7.88	
	9	8.75	8.62		8.92	8.86	
	10	9.72			9.91		
	11	10.69			10.91		
	12	11.64			11.89		
	13	12.62			12.88		
	14	13.59			13.88		

Figure 4-4: $\log_b$ of Mean Leaves Examined for $m = 1$, $\eta = \infty$

Algorithm	Search Depth (d)	Number of Players (N): 3 — Branching Factor (b) 2	3	6	Number of Players (N): 4 — Branching Factor (b) 2	3	6
Branch and Bound	3	2.91	2.88	2.83			
	4	3.90	3.88	3.82	3.99	3.99	3.97
	5	4.89	4.86	4.80	4.99	4.99	4.97
	6	5.88	5.85		5.99	5.99	
	7	6.88	6.86		6.99	6.99	
	8	7.86	7.85		7.99	7.99	
	9	8.86	8.85		8.99	8.99	
	10	9.87			9.99		
	11	10.86			10.99		
	12	11.86			11.99		
	13	12.86			12.99		
	14	13.87			13.99		
Lower-Eval	3	2.83	2.73	2.62			
	4	3.76	3.65	3.50	3.90	3.81	3.68
	5	4.69	4.54	4.38	4.85	4.74	4.60
	6	5.62	5.45		5.82	5.68	
	7	6.56	6.36		6.78	6.62	
	8	7.47	7.27		7.75	7.57	
	9	8.43	8.20		8.71	8.52	
	10	9.36			9.68		
	11	10.31			10.64		
	12	11.25			11.63		
	13	12.20			12.59		
	14	13.14			13.57		
Bimodal and Unimodal	3	2.83	2.73	2.61			
	4	3.77	3.64	3.48	3.90	3.82	3.68
	5	4.69	4.53	4.35	4.86	4.76	4.59
	6	5.62	5.43		5.83	5.70	
	7	6.56	6.34		6.79	6.63	
	8	7.46	7.25		7.77	7.58	
	9	8.42	8.17		8.74	8.54	
	10	9.35			9.71		
	11	10.30			10.68		
	12	11.24			11.67		
	13	12.19			12.64		
	14	13.13			13.62		

Figure 4-5: $\log_b$ of Mean Leaves Examined for $m = 1$, $\eta = 2$

Algorithm	Search Depth (d)	Number of Players (N) 3, Branching Factor (b) 2	3	6	4, Branching Factor (b) 2	3	6
Branch and Bound	3	2.93	2.91	2.86			
	4	3.93	3.90	3.86	3.99	3.99	3.98
	5	4.91	4.89	4.84	4.99	4.99	4.98
	6	5.90	5.88		5.99	5.99	
	7	6.90	6.89		6.99	6.99	
	8	7.89	7.89		7.99	7.99	
	9	8.89	8.88		8.99	8.99	
	10	9.90			9.99		
	11	10.89			10.99		
	12	11.89			11.99		
	13	12.89			12.99		
	14	13.90			13.99		
Lower-Eval	3	2.86	2.78	2.70			
	4	3.82	3.72	3.62	3.92	3.86	3.77
	5	4.75	4.64	4.52	4.88	4.81	4.71
	6	5.69	5.57		5.86	5.76	
	7	6.65	6.51		6.82	6.72	
	8	7.59	7.43		7.81	7.68	
	9	8.55	8.38		8.78	8.66	
	10	9.49			9.75		
	11	10.46			10.73		
	12	11.42			11.72		
	13	12.37			12.69		
	14	13.32			13.67		
Bimodal and Unimodal	3	2.86	2.78	2.68			
	4	3.82	3.71	3.58	3.93	3.86	3.75
	5	4.75	4.63	4.47	4.89	4.81	4.68
	6	5.69	5.55		5.88	5.77	
	7	6.65	6.49		6.84	6.73	
	8	7.58	7.40		7.83	7.69	
	9	8.54	8.34		8.80	8.66	
	10	9.48			9.79		
	11	10.45			10.77		
	12	11.41			11.76		
	13	12.36			12.74		
	14	13.31			13.72		

Figure 4-6: $\log_b$ of Mean Leaves Examined for $m = 1$, $\eta = 4$

Algorithm	Search Depth (d)	Number of Players (N): 3, Branching Factor (b)			Number of Players (N): 4, Branching Factor (b)		
		2	3	6	2	3	6
	3	2.97	2.97	2.93			
	4	3.97	3.96	3.93	4.00	4.00	3.99
	5	4.96	4.96	4.93	5.00	5.00	4.99
Branch	6	5.96	5.95		6.00	6.00	
and	7	6.97	6.95		7.00	7.00	
Bound	8	7.96	7.95		8.00	8.00	
	9	8.96	8.95		9.00	9.00	
	10	9.96			10.00		
	11	10.96			11.00		
	12	11.96			12.00		
	13	12.96			13.00		
	14	13.96			14.00		
	3	2.90	2.83	2.66			
	4	3.84	3.72	3.52	3.94	3.85	3.69
	5	4.79	4.61	4.35	4.90	4.79	4.58
Lower-	6	5.71	5.49		5.88	5.72	
Eval	7	6.67	6.36		6.84	6.63	
	8	7.58	7.23		7.79	7.57	
	9	8.51	8.10		8.76	8.49	
	10	9.41			9.72		
	11	10.34			10.69		
	12	11.28			11.65		
	13	12.18			12.61		
	14	13.09			13.58		
	3	2.90	2.83	2.66			
	4	3.84	3.72	3.52	3.94	3.85	3.69
	5	4.79	4.61	4.34	4.90	4.79	4.58
Bimodal	6	5.70	5.48		5.88	5.72	
and	7	6.66	6.35		6.84	6.63	
Unimodal	8	7.57	7.21		7.79	7.56	
	9	8.50	8.07		8.76	8.48	
	10	9.39			9.72		
	11	10.32			10.69		
	12	11.26			11.64		
	13	12.16			12.61		
	14	13.07			13.57		

Figure 4-7: $\log_b$ of Mean Leaves Examined for $m = 0$, $\eta = 1$

Algorithm	Search Depth (d)	$N=3$, $b=2$	$N=3$, $b=3$	$N=3$, $b=6$	$N=4$, $b=2$	$N=4$, $b=3$	$N=4$, $b=6$
Branch and Bound	3	2.99	2.99	2.98			
	4	3.99	3.99	3.98	4.00	4.00	4.00
	5	4.99	4.99	4.98	5.00	5.00	5.00
	6	5.99	5.99		6.00	6.00	
	7	6.99	6.99		7.00	7.00	
	8	7.99	7.99		8.00	8.00	
	9	8.99	8.99		9.00	9.00	
	10	9.99			10.00		
	11	10.99			11.00		
	12	11.99			12.00		
	13	12.99			13.00		
	14	13.99			14.00		
Lower-Eval	3	2.98	2.97	2.95			
	4	3.96	3.95	3.96	3.99	3.99	3.99
	5	4.95	4.95	4.93	4.98	4.99	5.01
	6	5.93	5.92		5.97	5.99	
	7	6.92	6.90		6.97	6.98	
	8	7.90	7.87		7.96	7.98	
	9	8.88	8.85		8.96	8.97	
	10	9.86			9.94		
	11	10.83			10.94		
	12	11.81			11.92		
	13	12.79			12.93		
	14	13.77			13.91		
Bimodal and Unimodal	3	2.98	2.96	2.92			
	4	3.96	3.93	3.90	3.99	3.98	3.94
	5	4.95	4.92	4.86	4.99	4.97	4.94
	6	5.93	5.88		5.98	5.96	
	7	6.92	6.86		6.97	6.94	
	8	7.90	7.82		7.97	7.93	
	9	8.88	8.79		8.96	8.92	
	10	9.86			9.95		
	11	10.84			10.96		
	12	11.81			11.94		
	13	12.79			12.94		
	14	13.77			13.93		

Figure 4-8: $\log_b$ of Mean Leaves Examined for $m = 0$, $\eta = \infty$

Algorithm	Search Depth (d)	Number of Players (N)					
		3			4		
		Branching Factor (b)			Branching Factor (b)		
		2	3	6	2	3	6
Branch and Bound	3	2.98	2.97	2.94			
	4	3.98	3.97	3.94	4.00	4.00	3.99
	5	4.97	4.96	4.94	5.00	5.00	4.99
	6	5.97	5.96		6.00	6.00	
	7	6.96	6.96		7.00	7.00	
	8	7.96	7.96		8.00	8.00	
	9	8.97	8.96		9.00	9.00	
	10	9.96			10.00		
	11	10.97			11.00		
	12	11.96			12.00		
	13	12.97			13.00		
	14	13.97			14.00		
Lower-Eval	3	2.93	2.87	2.78			
	4	3.88	3.80	3.68	3.95	3.89	3.78
	5	4.84	4.72	4.59	4.93	4.85	4.71
	6	5.79	5.66		5.90	5.80	
	7	6.74	6.59		6.87	6.76	
	8	7.69	7.54		7.86	7.72	
	9	8.66	8.48		8.83	8.68	
	10	9.61			9.81		
	11	10.59			10.78		
	12	11.54			11.76		
	13	12.50			12.74		
	14	13.45			13.72		
Bimodal and Unimodal	3	2.93	2.86	2.76			
	4	3.88	3.79	3.65	3.95	3.89	3.76
	5	4.84	4.71	4.55	4.93	4.85	4.69
	6	5.79	5.64		5.91	5.80	
	7	6.74	6.56		6.88	6.76	
	8	7.68	7.51		7.87	7.72	
	9	8.66	8.44		8.85	8.68	
	10	9.61			9.83		
	11	10.58			10.81		
	12	11.54			11.78		
	13	12.50			12.77		
	14	13.45			13.75		

Figure 4-9: $\log_b$ of Mean Leaves Examined for $m = 0$, $\eta = 2$

Algorithm	Search Depth (d)	$N=3$, $b=2$	$N=3$, $b=3$	$N=3$, $b=6$	$N=4$, $b=2$	$N=4$, $b=3$	$N=4$, $b=6$
Branch and Bound	3	2.98	2.98	2.95			
	4	3.98	3.97	3.95	4.00	4.00	3.99
	5	4.98	4.97	4.95	5.00	5.00	4.99
	6	5.98	5.97		6.00	6.00	
	7	6.97	6.97		7.00	7.00	
	8	7.97	7.97		8.00	8.00	
	9	8.97	8.97		9.00	9.00	
	10	9.97			10.00		
	11	10.97			11.00		
	12	11.97			12.00		
	13	12.97			13.00		
	14	13.97			14.00		
Lower-Eval	3	2.94	2.90	2.84			
	4	3.91	3.86	3.78	3.97	3.92	3.85
	5	4.87	4.80	4.72	4.94	4.90	4.82
	6	5.85	5.76		5.92	5.87	
	7	6.80	6.71		6.90	6.84	
	8	7.76	7.67		7.89	7.81	
	9	8.75	8.64		8.87	8.80	
	10	9.71			9.86		
	11	10.70			10.84		
	12	11.65			11.83		
	13	12.62			12.81		
	14	13.59			13.79		
Bimodal and Unimodal	3	2.94	2.89	2.82			
	4	3.92	3.84	3.74	3.97	3.92	3.82
	5	4.87	4.78	4.66	4.95	4.90	4.77
	6	5.85	5.74		5.93	5.86	
	7	6.80	6.68		6.91	6.83	
	8	7.76	7.63		7.91	7.80	
	9	8.75	8.59		8.89	8.78	
	10	9.71			9.88		
	11	10.70			10.87		
	12	11.65			11.85		
	13	12.62			12.84		
	14	13.59			13.83		

Figure 4-10: $\log_b$ of Mean Leaves Examined for $m = 0$, $\eta = 4$

N	η	Aggregate Lower Bound (m)					
		1			0		
		Branching Factor (b)			Branching Factor (b)		
		2	3	6	2	3	6
3	1	.89	.84	.78	.91	.87	.83
3	2	.94	.91	.87	.95	.94	.90
3	4	.95	.92	.89	.97	.96	.92
3	∞	.97	.94	.93	.98	.97	.96
4	1	.95	.92	.86	.96	.92	.89
4	2	.98	.96	.91	.98	.96	.93
4	4	.99	.97	.92	.99	.98	.95
4	∞	.99	.98	.97	.99	.99	.99

Figure 4-11: Estimated Values for Asymptotic Cost Function h

4.6.2.2 Analysis of Simulation Results

We concentrate on the performance of Bimodal-Unimodal, as the results indicate that it is the most promising of the algorithms. In the following, we interpolate from our results for η and m, and extrapolate from those for N.

Performance of Bimodal-Unimodal Evaluation: We conjecture that there is a function h such that

$$I_{\text{Bimodal-Unimodal}}(N, U_m, \eta, b, d) \approx b^{dh(N,U_m,\eta)}$$

Accordingly, we conjecture that

$$R_{\text{Bimodal-Unimodal}}(N, U_m, \eta, b) = h(N, U_m, \eta)b$$

Figure 4-11 contains estimated values for $h(N, U_m, \eta, b)$. These values were obtained by examining the behavior of

$$\log_b\left(I_{\text{Bimodal-Unimodal}}(N, U_m, \eta, b, d+1)\right) - \log_b\left(I_{\text{Bimodal-Unimodal}}(N, U_m, \eta, b, d)\right)$$

From this we infer that increasing values of b reduce h, but at a diminishing rate. Larger values of N and smaller values of m increase h fairly significantly. Increasing values of η increase h, though at $\eta = 2$, most of the increase has already occurred.

Naturally, increasing N or reducing m increase h, because in both cases it is harder to generate a cutoff, since there is less constraint on the sum of payoffs for any fixed-sized set. Likewise, increasing η increases h, since Bimodal-Unimodal is designed to work best for good ordering. Figures 4-12 through 4-17 display the behavior for different orderings and different players and distributions, respectively.

For small values of d, the rate of increase (with respect to d) of

$$\log_b(I_{\text{Bimodal-Unimodal}}(N, U_m, \eta, b, d))$$

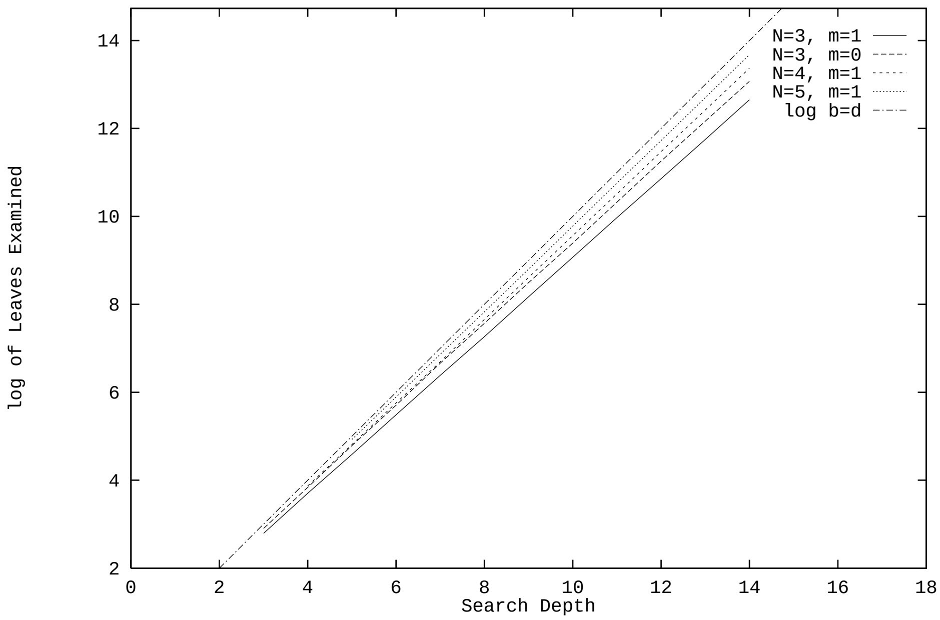


Figure 4-12: Comparison by Players and Distributions, $b = 2$

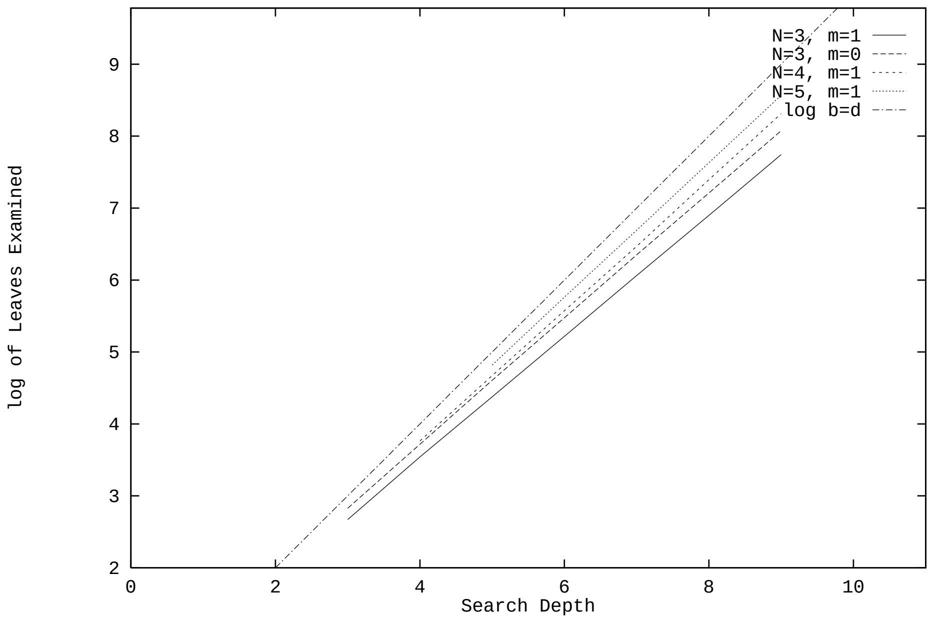


Figure 4-13: Comparison by Players and Distributions, $b = 3$

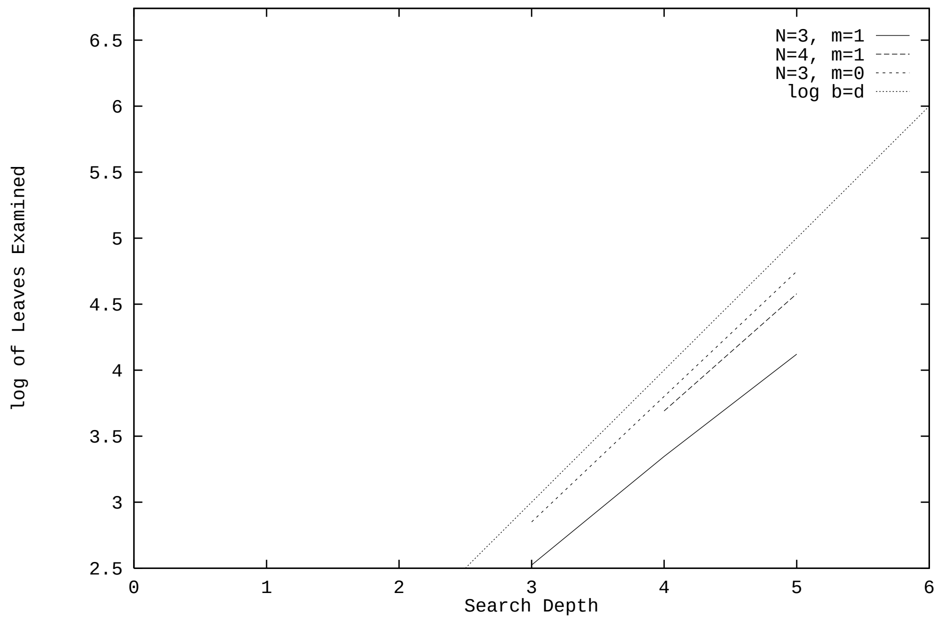


Figure 4-14: Comparison by Players and Distributions, $b = 6$

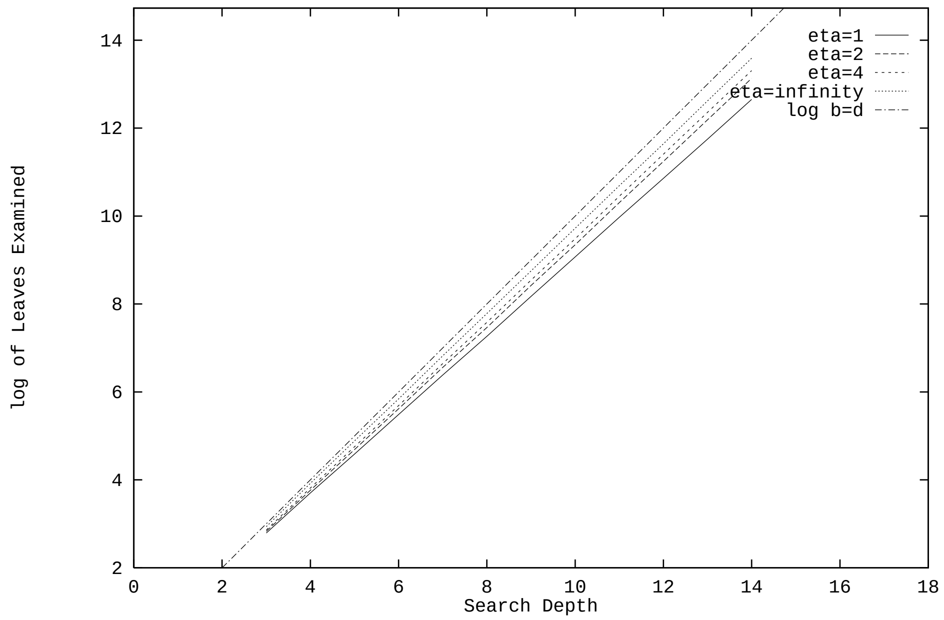


Figure 4-15: Comparison of Performance for Different Orderings, $b = 2$, $N = 3$, $m = 1$

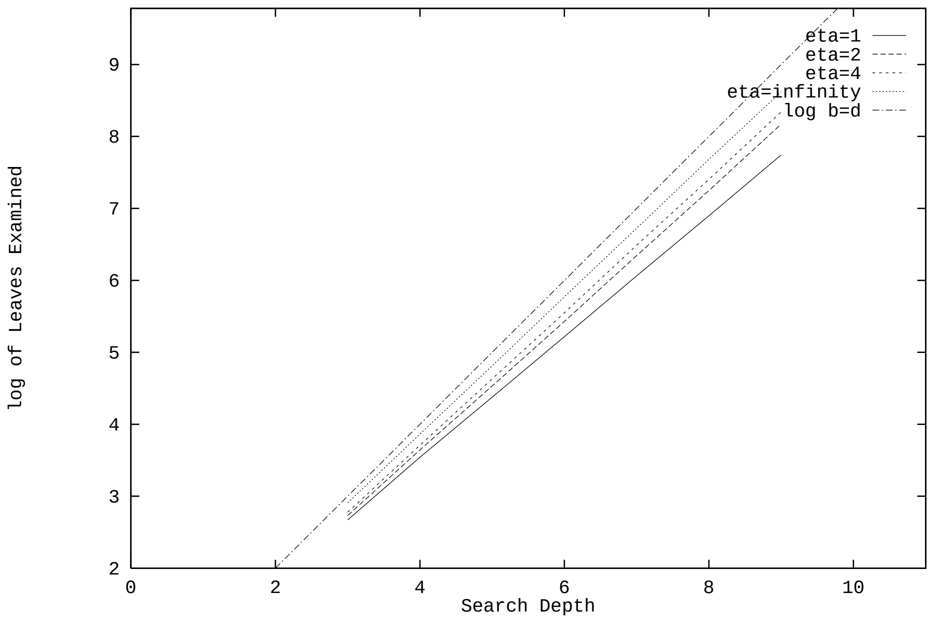


Figure 4-16: Comparison of Performance for Different Orderings, $b = 3$, $N = 3$, $m = 1$

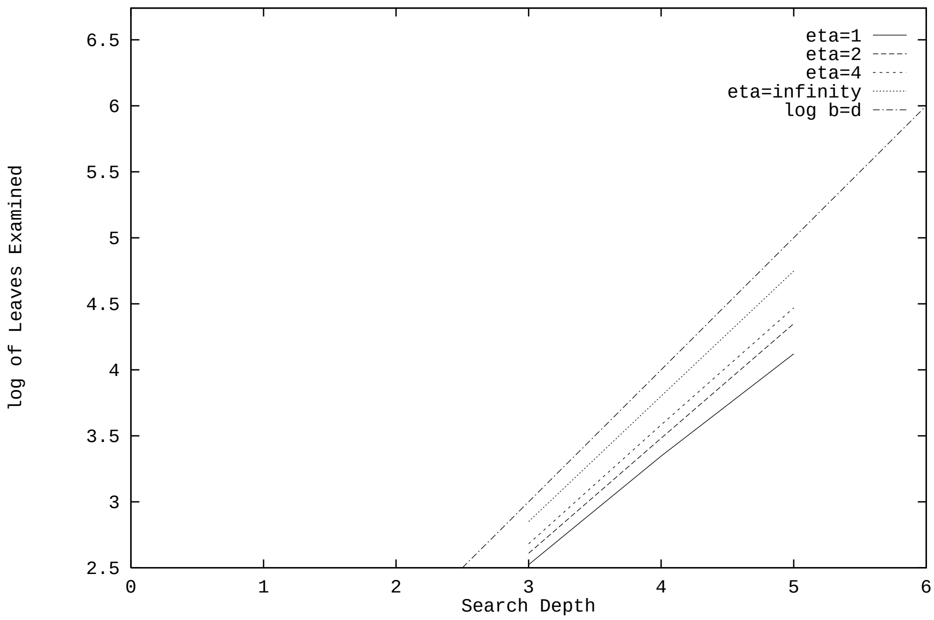


Figure 4-17: Comparison of Performance for Different Orderings, $b = 6$, $N = 3$, $m = 1$

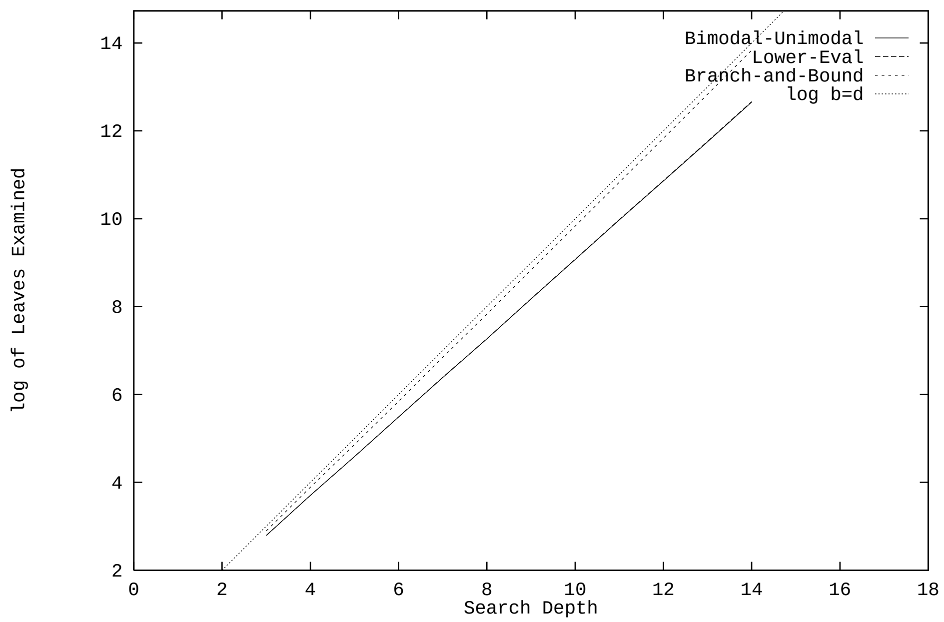


Figure 4-18: Comparison of Algorithm Performance for $\eta = 1$, $b = 2$, $m = 1$, $N = 3$. The curves for Bimodal-Unimodal and Lower-Eval overlay each other, appearing as the bottom line.

is a bit larger. We conjecture that this behavior is caused by the improved opportunities for passing non-zero lower bounds information to children in deeper trees. Once the depth exceeds the number of players by one or two, this improvement seems to stop.

Also, the data shows that $\log_b$ of nodes expanded generally increases more when depth is incremented to an odd number than an even one. This is like the best-case behavior of α-β: $I_{\alpha\text{-}\beta}(b,d) = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$.

4.6.2.3 Performance of Other Algorithms

We compare the performance of Bimodal-Unimodal with those of Branch-and-Bound and Lower-Eval.

Performance of Branch-and-Bound: Branch-and-Bound searches slightly fewer nodes than a naive algorithm. For the tested distributions of leaf value, number of players, ordering of moves, and branching factor, it appears that the fraction of leaf nodes examined, as a function of depth, is nearly a constant fraction of the total. Moreover, it seems that g does not depend appreciably on b. So we conjecture that there is some function $g(N, U_m, \eta)$, with $0 < g < 1$ such that

$$I_{\text{Branch-and-Bound}}(N, U_m, \eta, b, d) \approx g(N, U_m, \eta) b^d$$

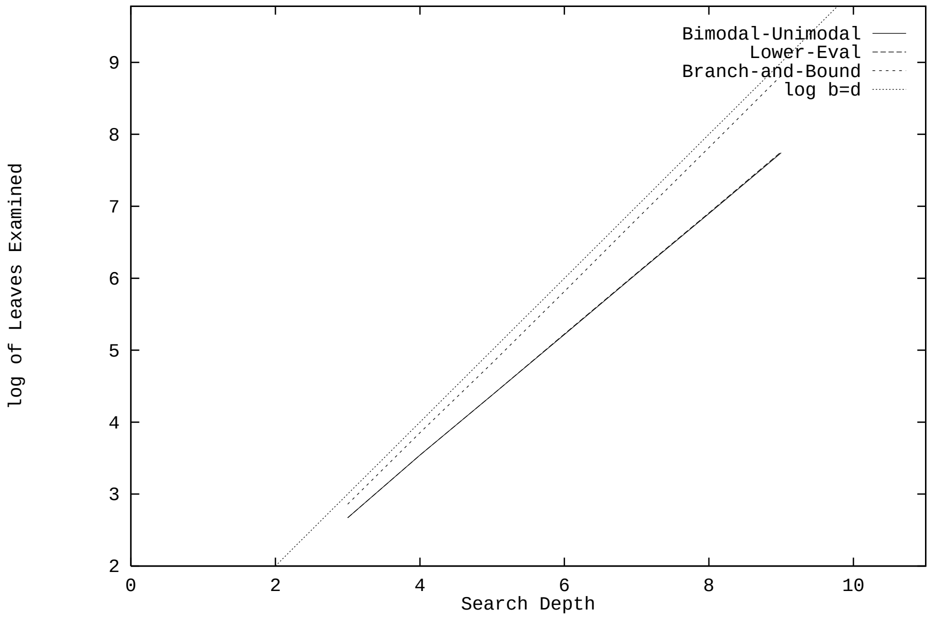


Figure 4-19: Comparison of Algorithm Performance for $\eta = 1$, $b = 3$, $m = 1$, $N = 3$. The curves for Bimodal-Unimodal and Lower-Eval overlay each other, appearing as the bottom line.

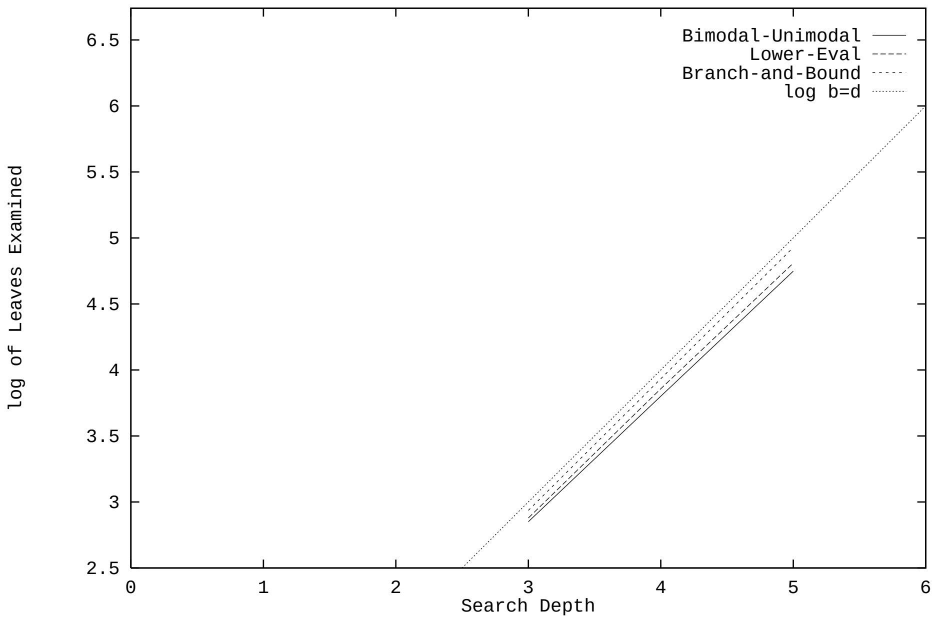


Figure 4-20: Comparison of Algorithm Performance for $\eta = 1$, $b = 6$, $m = 1$, $N = 3$.

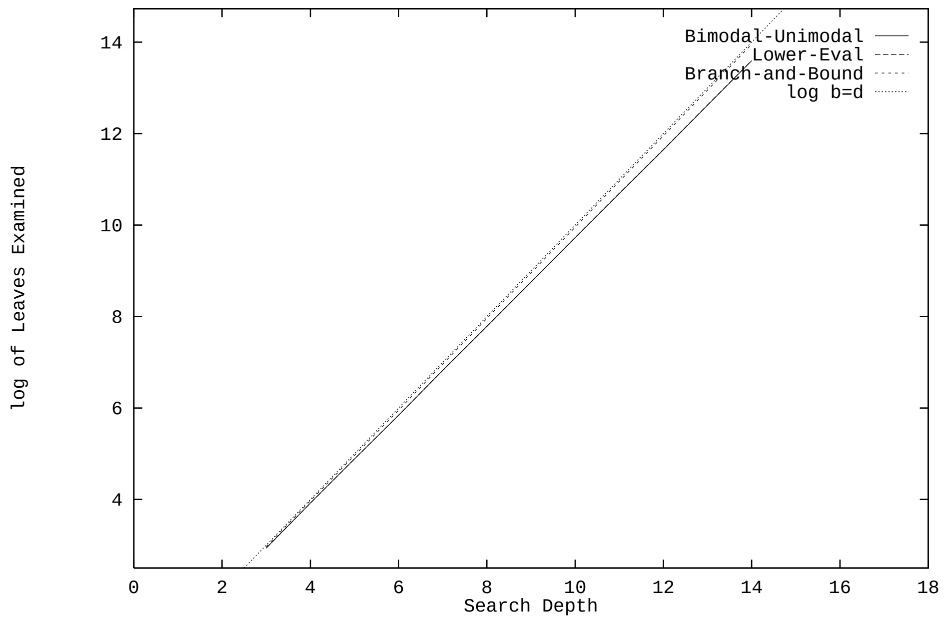


Figure 4-21: Comparison of Algorithm Performance for $\eta = \infty$, $b = 2$, $m = 1$, $N = 3$. The curves for Bimodal-Unimodal and Lower-Eval overlay each other, and correspond to the bottom line. Those for Branch-and-Bound and $\log(b) = d$ are close to each other, above the curves for Bimodal-Unimodal and Lower-Eval.

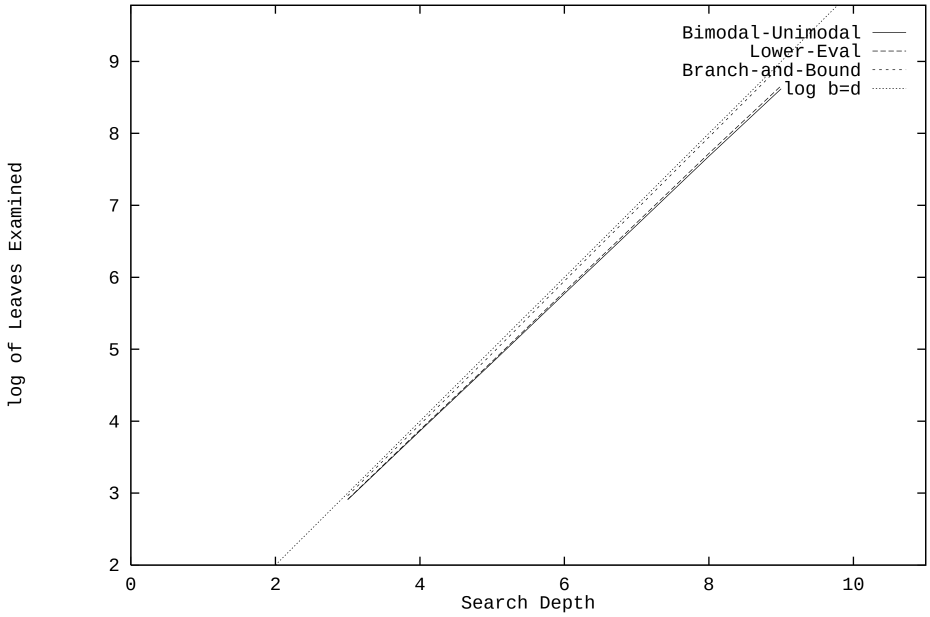


Figure 4-22: Comparison of Algorithm Performance for $\eta = \infty$, $b = 3$, $m = 1$, $N = 3$. The curves for Bimodal-Unimodal and Lower-Eval are close to each other. Those for Branch-and-Bound and $\log(b) = d$ are close to each other, above the curves for Bimodal-Unimodal and Lower-Eval.

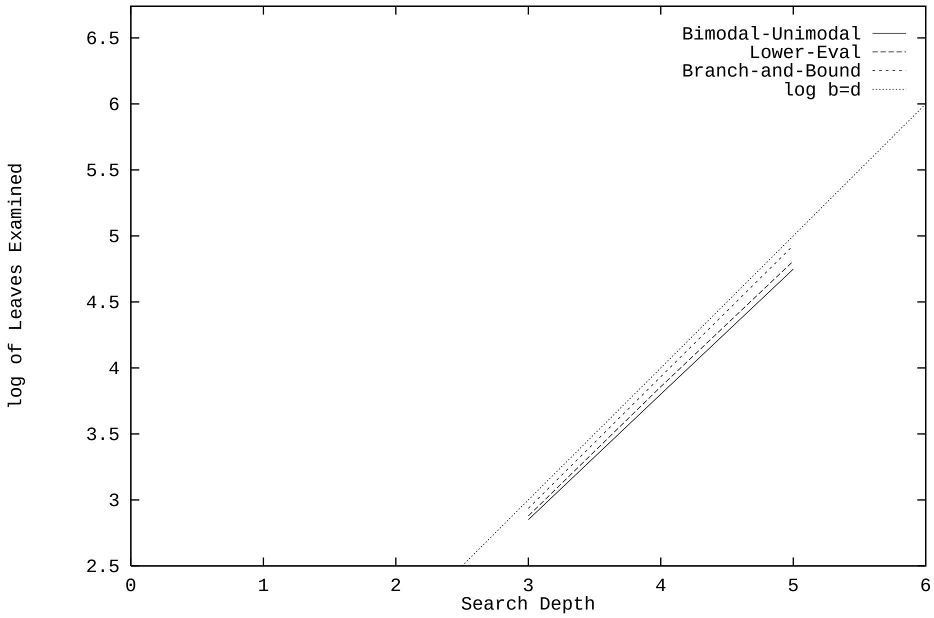


Figure 4-23: Comparison of Algorithm Performance for $\eta = \infty$, $b = 6$, $m = 1$, $N = 3$

Accordingly, we conjecture that

$$R_{\text{Branch-and-Bound}}(N, U_m, \eta, b) = b$$

As N increases, g rapidly approaches 1. Also, g is inversely related to m. Finally, g is increased with larger values of η.

The results strongly support the conclusion that Bimodal-Unimodal is superior to Branch-and-Bound.

4.6.2.4 A Comparison of Bimodal-Unimodal and Lower-Eval

In some circumstances Lower-Eval performs approximately as well as Bimodal-Unimodal. Its performances degrades relative to that of Bimodal-Unimodal as η increases. In trees with more terminal nodes (i.e., deeper trees or those with larger branching factors), Lower-Eval also performs worse. Lower-Eval's effective branching factor appears to slowly increase (with d); at least it increases relative to that of Bimodal-Unimodal. As b increases, this effect is more pronounced. For small trees with $\eta = 2, 4$, Lower-Eval actually performed better than Bimodal-Unimodal.

For larger N and for smaller m, Lower-Eval is more noticeably inferior to Bimodal-Unimodal.

4.6.3 Two-Player Non-Constant-Sum Game Trees

We tested these trees for $\eta = 1$ and for $\eta = \infty$. We tested the (uniform) distributions U_m, for $m = 0$, $m = .8$ and $m = .98$.

Algorithm	Search Depth (d)	Degree of Ordering (η): 1, Branching Factor (b): 2	3	6	∞, Branching Factor (b): 2	3	6
Branch and Bound	2	1.59	1.48	1.36	1.87	1.84	1.78
	3	2.34	2.20	2.09	2.79	2.70	2.65
	4	3.03	2.81	2.62	3.69	3.57	3.49
	5	3.75	3.47		4.56	4.44	
	6	4.45	4.13		5.47	5.34	
	7	5.15	4.81		6.37	6.22	
	8	5.89			7.27		
	9	6.60			8.20		
	10	7.30			9.09		
	11	8.04			10.01		
Lower-Eval	2	1.59	1.48	1.36	1.87	1.84	1.78
	3	2.34	2.20	2.09	2.79	2.70	2.65
	4	3.03	2.81	2.62	3.69	3.57	3.49
	5	3.75	3.47		4.56	4.44	
	6	4.45	4.13		5.47	5.34	
	7	5.15	4.81		6.36	6.22	
	8	5.89			7.26		
	9	6.60			8.20		
	10	7.30			9.09		
	11	8.04			10.01		
Bimodal and Unimodal	2	1.59	1.48	1.36	1.87	1.84	1.78
	3	2.34	2.20	2.09	2.79	2.70	2.65
	4	2.84	2.64	2.46	3.63	3.48	3.41
	5	3.54	3.30		4.44	4.31	
	6	4.01	3.77		5.24	5.12	
	7	4.64	4.44		6.08	5.93	
	8	5.19			6.88		
	9	5.82			7.74		
	10	6.34			8.53		
	11	7.03			9.37		

Figure 4-24: $\log_b$ of Mean Leaves Examined for $m = .98$, $N = 2$

Algorithm	Search Depth (d)	Degree of Ordering (η) 1, Branching Factor (b) 2	3	6	∞, Branching Factor (b) 2	3	6
Branch and Bound	2	1.69	1.60	1.55	1.91	1.87	1.85
	3	2.47	2.41	2.33	2.84	2.79	2.75
	4	3.26	3.13	3.04	3.76	3.72	3.65
	5	4.06	3.91		4.72	4.62	
	6	4.90	4.68		5.65	5.54	
	7	5.70	5.44		6.58	6.48	
	8	6.48			7.51		
	9	7.33			8.49		
	10	8.13			9.40		
	11	8.93			10.33		
Lower-Eval	2	1.69	1.60	1.55	1.91	1.87	1.85
	3	2.47	2.41	2.33	2.84	2.80	2.79
	4	3.26	3.12	3.03	3.76	3.72	3.68
	5	4.06	3.90		4.71	4.63	
	6	4.90	4.67		5.64	5.54	
	7	5.69	5.44		6.56	6.48	
	8	6.48			7.49		
	9	7.33			8.46		
	10	8.13			9.37		
	11	8.92			10.29		
Bimodal and Unimodal	2	1.69	1.60	1.55	1.91	1.87	1.85
	3	2.47	2.41	2.33	2.84	2.80	2.79
	4	3.17	3.05	2.95	3.73	3.71	3.66
	5	3.98	3.83		4.68	4.61	
	6	4.77	4.56		5.59	5.51	
	7	5.55	5.30		6.51	6.44	
	8	6.32			7.43		
	9	7.15			8.39		
	10	7.93			9.29		
	11	8.69			10.21		

Figure 4-25: $\log_b$ of Mean Leaves Examined for $m = .8$, $N = 2$

		Degree of Ordering (η)					
		1			∞		
Algorithm	Search Depth (d)	Branching Factor (b)			Branching Factor (b)		
		2	3	6	2	3	6
Branch and Bound	2	1.86	1.79	1.71	1.95	1.95	1.90
	3	2.71	2.60	2.46	2.91	2.89	2.80
	4	3.60	3.45	3.19	3.87	3.80	3.73
	5	4.44	4.18		4.81	4.74	
	6	5.29	4.98		5.80	5.66	
	7	6.14	5.77		6.72	6.59	
	8	6.94			7.67		
	9	7.78			8.62		
	10	8.63			9.56		
	11	9.41			10.51		
Lower-Eval	2	1.86	1.79	1.71	1.95	1.95	1.90
	3	2.71	2.60	2.46	2.91	2.90	2.86
	4	3.59	3.44	3.18	3.86	3.82	3.77
	5	4.44	4.17		4.80	4.76	
	6	5.29	4.96		5.78	5.69	
	7	6.13	5.76		6.70	6.62	
	8	6.93			7.64		
	9	7.77			8.58		
	10	8.62			9.53		
	11	9.39			10.46		
Bimodal and Unimodal	2	1.86	1.79	1.71	1.95	1.95	1.90
	3	2.71	2.60	2.46	2.91	2.90	2.86
	4	3.58	3.41	3.12	3.86	3.82	3.76
	5	4.40	4.12		4.80	4.75	
	6	5.24	4.88		5.76	5.67	
	7	6.05	5.65		6.68	6.60	
	8	6.83			7.61		
	9	7.65			8.54		
	10	8.49			9.48		
	11	9.24			10.41		

Figure 4-26: $\log_b$ of Mean Leaves Examined for $m = 0$, $N = 2$

Search Depth (d)	Degree of Ordering (η): 1			Degree of Ordering (η): ∞		
	Branching Factor (b)			Branching Factor (b)		
	2	3	6	2	3	6
3	2.84	2.73	2.56	2.96	2.94	2.88
4	3.74	3.58	3.40	3.94	3.89	3.82
5	4.64	4.44	4.22	4.88	4.84	4.78
6	5.52	5.26		5.87	5.79	
7	6.42	6.10		6.84	6.75	
8	7.33			7.80		
9	8.22			8.77		
10	9.12			9.74		
11	10.00			10.71		

Figure 4-27: $\log_b$ of Mean Leaves Examined with Additional Bounds Information

The branching factor in this case is much smaller than b; for $m = .98$ the algorithm performs virtually identically to α-β. Indeed, in the best case α-β examines $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ terminal nodes. Bimodal-Unimodal exhibits a corresponding disparity in the rate of increase between even and odd levels. For larger m, the disparity is most noticeable.

For comparison with α-β, note that $\log_2(2^{\lceil 11/2 \rceil} + 2^{\lfloor 11/2 \rfloor} - 1) = 6.57$, $\log_3(3^{\lceil 7/2 \rceil} + 3^{\lfloor 7/2 \rfloor} - 1) = 4.26$, and $\log_6(6^{\lceil 4/2 \rceil} + 6^{\lfloor 4/2 \rfloor} - 1) = 2.38$. This illustrates how close the performance of Bimodal-Unimodal is to that of α-β, when the game is nearly constant-sum.

For these trees, we also note that the costs increase for smaller values of m or b, and for larger values of η.

4.6.3.1 Three-Player Game Trees With Additional Bounds

We employed lower bounds on doubletons, so for each pair of players $i \neq j$, we had $m_{\{i,j\}} = k$, for $0 \leq k \leq .5$, in addition to having $m = M = 1$. We created a random tree using a uniform distribution on the set of joint distributions that satisfies these constraints.

We tested for Bimodal-Unimodal only, because the other algorithms cannot use the additional lower bound information (so their performance will be the same as without such bounds information).

These results indicate that additional bounds information does not provide a significant improvement in the efficiency of our algorithm, although it may be that a uniform distribution of values that obeys the pairwise bound constraint generates worse prospects for cutoffs without the use of such bounds information.

4.6.4 Conclusions

The results suggest that certain kinds of games can be handled quite efficiently by Bimodal-Unimodal. For three-player games with large branching factors ($b > 15$), and good move ordering, but using only global bounds information, Bimodal-Unimodal should be approximately half-way between the efficiency of α-β and naive evaluation: in this case we expect $R \approx b^{3/4}$.

4.6.5 Preliminary Results from Playing Abalone

We implemented our algorithm to play Three-Player Abalone, using Michael Frank's Abalone program. This implementation was preliminary: we did not have the time to implement good move ordering and a strong heuristic playing function or any optimizations. However, even in this fairly primitive state, we were able to obtain a significant improvement in searching efficiency (nearly halving the time required to search to depth three). The program used the assumption that the other players would play rationally, and it appeared to make good moves, even though it used a simple heuristic scoring function. This offers some support to our assessment that assuming rationality would allow for strong play in practice.

Chapter 5

Conclusion

Our research has taken many of the techniques from traditional computational game theory, and successfully extended these to a wider class of games.

5.1 Results Achieved

Our most important results are algorithms that efficiently solve extended classes of games—two-player zero-sum simultaneous play games (including games with chance) and N-player games of perfect information (with bounds on payoff sums). In this thesis, we have presented these algorithms and proved the correctness of the methods these algorithms use to improve performance.

We analyzed the best-case asymptotic complexity of the algorithms we have provided. In the best case, for both classes of games (without chance), our algorithms permitted us to search approximately twice as deeply as a naive algorithm would. For games of perfect information involving chance, in the best case, our algorithm reduced the effective branching factors for player moves to the square root of the number of moves available, though the effective branching factor for chance alternatives was not reduced.

We also have provided empirical evidence of the efficiency of these algorithms. The algorithms for two-player zero-sum simultaneous play games were able to achieve nearly best-case performance in the political simulation game we devised. The algorithm for N-player games of perfect information exhibited a reduced branching factor that also suggested this algorithm represents a significant improvement over the naive solution algorithm.

We also extended various additional techniques from 2ZPI games to our more general classes of games. We have described how to extend the basic heuristic technique of solving approximation games (using heuristic evaluation functions). We also have described how to extend many of the optimizations that have been used in 2ZPI games, such as iterative deepening, transposition tables and singular extensions.

5.1.1 Two-Player Zero-Sum Simultaneous Play Games

In two-player zero-sum games with simultaneous play, we have described two promising evaluation algorithms: the substructure evaluation algorithm and the dominating evaluation algorithm. The substructure evaluation algorithm has a better best case performance and is more robust (not requiring domination of moves). Because it appears practical to achieve nearly best case performance, it is not surprising that substructure evaluation appears to be the superior algorithm.

In addition to extending techniques from 2ZPI games, we solved some unique problems. The most interesting of these is the question of how to set bounds for the value of a linear combination of values each of which must be (recursively) tested to allow a high probability of success. This led us to develop methods for testing a bound on linear combinations of values, when computing the values was more expensive than testing these values. This line of consideration was also naturally extended to allow for efficiently solving two-player, zero-sum games of simultaneous play that involve chance. This result for chance is also relevant to two-player zero-sum games of perfect information.

To implement the algorithms for solving two-player zero-sum simultaneous play games we needed to supply a number of sub-algorithms. In accordance with this need, we considered various possible policies. The discussion of these was more speculative, relying more on intuition and experience with specific games of this kind. We collected statistics from our case study to strengthen some of our assumptions. However, we believe that the specific policies chosen will exert a second-order influence on the performance of the algorithms. The policies we implemented did allow our program that played the political simulation to achieve an impressive performance improvement over naive evaluation. For different games, some of these policies may need to be changed and more detailed analysis of some policies may be required.

5.1.2 N-Player Perfect Information Games

Our other major extension to the class of games amenable to efficient solution involved games of perfect information in which there can be more than two players, and the sum of payoffs is not constant, but merely bounded. We have provided an algorithm that significantly improves performance over naive evaluation. Our simulation results suggest that the effective branching factor is reduced significantly for these games. Preliminary results from play of Abalone indicate that the algorithm also significantly improves search in real games.

We have also extended concepts from traditional computational game theory to this class of games. We believe that considering opponent models that incorporate resource limitations and use of a system that employs a more sophisticated approach to backing up values should improve quality of play. However, we believe that the basic approach of assuming rational play (possibly using a modification to allow for risk aversion) will produce good results. Preliminary results from play of Abalone support this conjecture.

5.2 Future Directions

Our presentation of algorithms to efficiently solve larger classes of games is a useful step in fulfilling the objectives of studying computer play of games. It allows for better models of strategic interactions to be simulated on computers, both for prescriptive and descriptive approaches.

There are important areas for future work that can improve upon the possibilities for computational game theory. Naturally, advances in many areas of AI research can be profitably incorporated. Advances in making computers plan and reason strategically are important and extend to the more general classes of games we have considered as well as for 2ZPI games. Other issues that are applicable to computational game theory extend to our more general games, such as incorporating metareasoning and handling search depth pathology. Learning and agent modelling are two additional topics from which advances can be incorporated. Naturally, many other fields in AI have a bearing on computational game theory (and conversely, computational game theory can be used to test ideas that are applicable to many fields in AI).

5.2.1 Model Games

One specific area in AI that seems especially important when dealing with complicated strategic interactions is use of hierarchical problem solving. Our idea is to employ simplified "model" games. These can be used to obtain strategic insight, or to examine specific aspects of an interaction. This search can produce a solution that is directly translated into the more detailed game domain. Another approach is to use the search to set higher-level objectives, so that one changes the heuristic evaluation of positions to favor actions that implement the overall strategy determined by solving the model game. One can use solutions from model games to allocate resources to various "tasks" and search games for each task separately, using the constraint limitations from the overall allocation. This approach allows for a dramatic reduction in complexity, by considering the different actions independently. Conversely, one can use model games to search separate aspects of an interaction and combine the distinct recommendations. In the context of the game Go, it has been suggested that this approach might be useful, for considering play in a given section of the game board.

Using this approach seems especially appropriate for real-world interactions; parlor games are already simple, and it is harder to construct models that abstract away certain features but still bear a close resemblance. Let us conclude this idea by presenting an example of how one might use model games.

Suppose we are modelling an industrial game, in which several competing firms are interacting. To select a move, one might use a model game in which the resources of a firm are broken down into large categories, and into highly discrete units. One uses a simple approximation of the overall outcomes of various interactions, and solves for a good overall allocation of resources. After determining the best overall allocation, one might solve a second model game to refine the overall resource distribution. In either case, after finishing with the (strategic) model game, one has an allocation of resources for various tasks (like marketing a category of products, or investing in new facilities).

The next phase has one search several subordinate games, using the resource allocations that each player is expected to make for the given kinds of resources. This involves using several (tactical) model games to capture the specific interactions for a particular kind of allocation. It seems best to search less deeply in these games, since it is necessary to have a good degree of confidence in the overall allocation strategies (i.e., including allocations to be made in subsequent moves) that will be employed.

5.2.2 Handling More General Games

The other major area for future research is specific to games. This is to consider how to handle a wider class of games. There are two obstacles for such research. From a game theoretic point of view, there is no agreement on how rational players will play a game. Several equilibrium refinements have been proposed, which specify additional criteria for a solution profile, beyond its simply being an equilibrium. Harsanyi and Selten (in [HS88]) have proposed a general theory of equilibrium selection that prescribes a unique solution for "rational" players in playing a game, and Güth and Kalkofen (in [GK89]) have proposed another. However, these proposals are not generally accepted in the game theory community.

Nevertheless, it is certainly worthwhile examining how to efficiently implement various equilibrium refinements (returning all the strategy profiles that meet the refinement's additional criteria). For descriptive uses, computing all the profiles may suffice. For prescriptive uses, one might use ad-hoc procedures specific to the domain to select among profiles if multiple profiles exist.

We believe that the evaluation techniques developed for two-player zero-sum simultaneous play games can be usefully extended to handling equilibria in more general classes of games. The ideas underlying both of the evaluation algorithms should be applicable. The substructure evaluation algorithm essentially tests strategies to determine whether they are best replies. Testing strategies for domination and to determine whether they are best replies are universally applicable when computing equilibria. Accordingly, these techniques promise to be useful in extending to computing equilibria and refinements for more general games. However, additional questions will arise, such as testing for more stringent conditions or handling tests when there is not a unique solution value for a game. Another concern is the complexity of solving general games. If a refinement requires us to employ strategic forms, and we cannot handle this implicitly by using behavior strategies (strategies at a single information set), then the super-exponential complexity for games with even small extensive forms would make the problem beyond reasonable hope of computation.

For the case of general two-player games in extensive form, Wilson[Wil72] described an algorithm for computing equilibrium points. His approach employs a procedure for determining best replies using the extensive form (without explicitly representing all of a player's pure strategies). We believe it is possible to employ a testing approach similar to that used in substructure evaluation in conjunction with this algorithm to efficiently compute solutions to quite general classes of two-player zero-sum games. This is definitely an area worthy of further investigation.

The last area for future research is being pursued already within the game theory community. This is the question of how one might define a jointly prescriptive-descriptive theory of play of games by resource-limited rational agents (see, e.g., Aumann [Aum92]). This question is also being pursued within the distributed AI community, in the context of modelling opponent behaviors.

5.2.3 Further Applications

We believe that further investigating the behavior of our algorithms on various games will prove fruitful. One promising area is to investigate performance on games that are patterned on games that have been used in the social sciences, but providing more detail in the game.

We also believe that further testing our algorithms on games will prove advantageous. Providing more detailed testing on Abalone, and on other multiple-player games should give better insight into how our algorithm for N-player perfect games performs. The CSCOUT algorithm appears promising for allowing extended look-ahead in a game like backgammon (if nearly best case performance is obtained, we expect to allow 50% deeper searches than naive searching allows).

5.3 Summary

In this thesis, we have demonstrated the capability to extend computational game theory to larger classes of games. The extensions we have presented have the potential to allow more complicated and interesting interactions to be efficiently analyzed. We believe that this thesis represents an important contribution to the problem of computing actions during strategic interactions.

Bibliography

[ACH90] T. S. Anantharaman, M. S. Campbell, and F.-H. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–110, April 1990.

[AD83] S. G. Akl and R. J. Doran. A comparison of parallel implementations of the alpha-beta and scout tree search algorithms using the game of checkers. In M. A. Bramer, editor, *Computer game-playing*. Ellis Horwood Limited, Chichester, England, 1983.

[Aum92] R. Aumann. Perspectives on bounded rationality. In Y. Moses, editor, *Proceedings of the Fourth Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 108–117, 1992.

[Bau92] E. B. Baum. On optimal game tree propagation for imperfect players. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 507–512. American Association for Artificial Intelligence, 1992.

[Bea90] D. F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, April 1990.

[Bel70] E. J. Beltrami. *An Algorithmic Approach to Nonlinear Analysis and Optimization*. Academic Press, New York, New York, 1970.

[Ber79] Hans Berliner. The B^* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[Bod92] R. J. Bodkin. A corrected proof that scout is asymptotically optimal. In preparation, 1992.

[Chv83] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, New York, 1983.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, 1963.

[Fre83] P. W. Frey. The α-β algorithm: incremental updating, well-behaved evaluation functions, and non-speculative forward pruning. In M. A. Bramer, editor, *Computer game-playing*, chapter 21, pages 285–289. Ellis Horwood Limited, Chichester, England, 1983.

[FT91] D. Fudenberg and J. Tirole. *Game Theory.* MIT Press, Cambridge, MA, 1991.

[FU92] T. E. Fawcett and P. E. Utgoff. Automatic feature generation for problem solving systems. COINS Technical Report 92-9, Computer and Information Science Department, University of Massachusetts, Amherst, January 1992.

[GDW91] P. Gmytrasiewicz, E. H. Durfee, and D. K. Wehe. A decision-theoretic approach to coordinating multiagent interactions. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 62–74, August 1991.

[Gil72] J. J. Gillogly. The technology chess program. *Artificial Intelligence*, 3:145–163, 1972.

[GK89] W. Güth and B. Kalkofen. *Unique Solutions for Strategic Games.* Springer-Verlag, New York, New York, 1989.

[HACN90] F.-H. Hsu, T. S. Anantharaman, M. S. Campbell, and A. Nowatzyk. Deep thought. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 55–78. Springer-Verlag, New York, New York, 1990.

[Har67] J. Harsanyi. Games with incomplete information played by bayesian players. *Management Science*, 14:159–182,320–334,486–502, 1967-8.

[HM89] O. Hansson and A. Mayer. Decision-theoretic control of search in BPS. In *AAAI Spring Symposium on AI and Limited Rationality*, pages 59–63, 1989.

[HS88] John C. Harsanyi and Reinhard Selten. *A General Theory of Equilibrium Selection in Games.* MIT Press, 1988.

[Kar84] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[KM75] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[Lev87] D. Levy, editor. *Computer Games.* Springer-Verlag, New York, New York, 1987.

[LM90] K.-F. Lee and S. Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43:21–36, 1990.

[LR92] R. Levy and J. S. Rosenschein. A game theoretic approach to distributed artificial intelligence and the pursuit problem. In Y. Demazeau and E. Werner, editors, *Decentralized Artifical Intelligence III.* Elsevier Science Publishers B.V./North-Holland, 1992.

[MC82] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[McA88] D. A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[McC90] J. McCarthy. Chess as the drosophila of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, chapter 14, pages 227–237. Springer-Verlag, 1990.

[MOS86] T. A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor tree-search experiments. In D. F. Beal, editor, *Advances in Computer Chess 4*, pages 37–51. Pergamon Press, Oxford, 1986.

[New89] M. Newborn. Computer chess: Ten years of significant progress. In *Advances in Computers*, pages 197–250. Academic Press, 1989.

[Pal83] A. Palay. *Searching with probabilities.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1983.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[RB89] J. Rosenschein and J. Breese. Communication-free interactions among rational agents: A probabilistic approach. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence*, volume 2, pages 99–118. Pitman Publishing, London, England, 1989.

[Rei83] A. Reinefeld. An improvement of the scout tree search algorithm. In *ICCA Journal*, volume 6, pages 4–14, 1983.

[Riv88] R. L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34:77–96, 1988.

[RSM85] A. Reinefeld, J. Schaeffer, and T. A. Marsland. Information acquisition in minimal window search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1040–1043, August 1985.

[Rus90] S. Russell. Fine-grained decision-theoretic search control. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 436–442, 1990.

[RW89] S. Russell and E. Wefald. On optimal game-tree search using rational metareasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 334–340, 1989.

[SA77] D. Slate and L. Atkin. Chess 4.5: The northwestern university chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

[Sch83] J. Schaeffer. The history heuristic. *ICCA Journal*, 6(3):16–19, 1983.

[Sch90] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43:67–84, 1990.

[Sha88] C. E. Shannon. Programming a computer for playing chess. In D. Levy, editor, *Computer Chess Compendium*, pages 2–27. Springer-Verlag, 1988.

[Shu82] M. Shubik. *Game Theory in the Social Sciences*. MIT Press, Cambridge, MA, 1982.

[Tar83] M. Tarsi. Optimal search on some game trees. *Journal of the ACM*, 30(3):389–396, 1983.

[vNM44] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, 1944.

[Wil72] R. Wilson. Computing equilibria of two-person games from the extensive form. *Management Science*, 18(7):448–460, 1972.

[Ye91] Y. Ye. An $O(n^3L)$ potential reduction algorithm for linear programming. *Mathematical Programming*, 50:239–258, 1991.