Summary

This thesis demonstrates by example how dynamic stochastic control methods may usefully be applied to tackle problems of tree searching and computer game-playing. Underlying this work is the constant tension between what is optimal *in theory* and what is implementable *in practice*. Most of the games studied are solved (under certain conditions) in the sense that a policy is derived which is both optimal and implementable. We examine the various reasons why the general problem of devising an algorithm to play a perfect information game in real time cannot have such a solution, and consider how to respond to this difficulty.

Chapter 1 defines the nature of the problem by introducing the concept of a game tree and explaining the concept of selectivity in game tree search. It then reviews the most important game tree search methods.

Chapter 2 explains what a Markov chain model of computer game-playing might include. It then introduces a much simpler one-player search game and establishes optimal policy under certain conditions. It also contains analysis of a discrete fuel control problem, which was developed as an offshoot of this research.

Chapter 3 details a stochastic satisficing tree search model, and explains the relationship with the standard bandit models. The optimal policy is established, and some important extensions presented.

Chapter 4 considers the difficulties of extending this model to a two-player game tree.

Chapter 5 highlights the connection between the problem of time- and search-control. The chapter then provides an overview of the approaches taken to the problem of time control, before proving an optimal time- and search-control policy for a simple two-player game.

Chapter 6 is the most wide ranging in its scope and is approachable to the artificial intelligence researcher less familiar with the language of dynamic stochastic control. It presents a new selective search algorithm, PCN*, and highlights problems with some of the existing game-playing models.

1 Game Tree Search

A game tree is a tree which represents the current state and possible future states of a game. The nodes correspond to game positions, while the arcs which connect them correspond to moves. Although such a framework allows treatment of very general games, the attention of this work is limited, on grounds of tractability, to one-player games and zero-sum two-player games. The arcs between the positions are implicitly directed, from top to bottom. A game's current position is the root of the game tree, which is indicated in this thesis by a triangle.

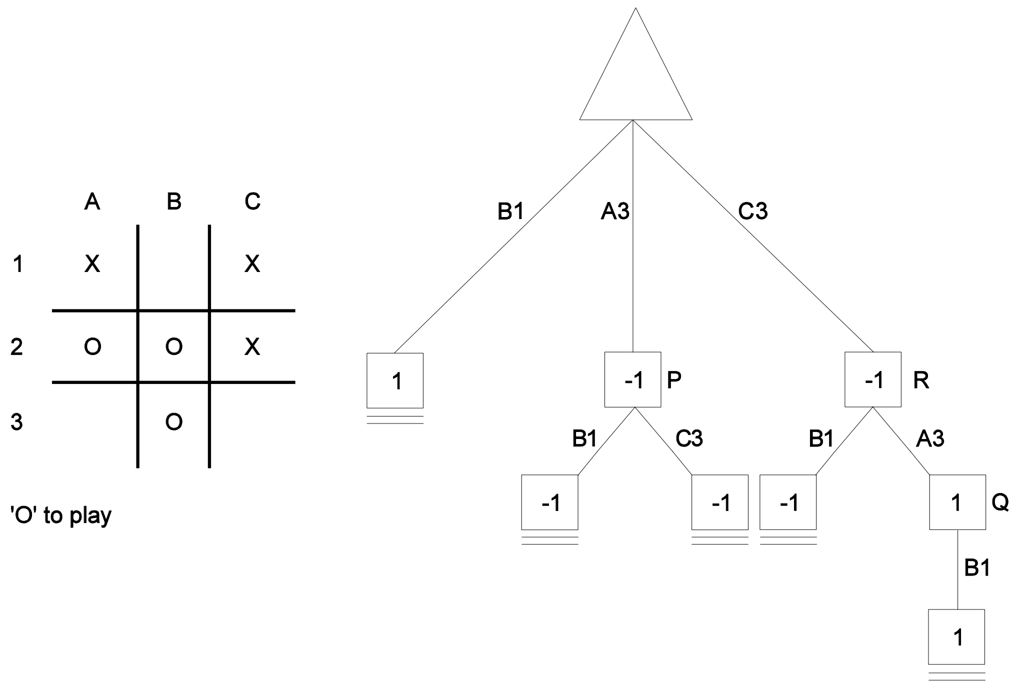

Figure 1: A Noughts & Crosses Position and Corresponding Game Tree

The scalars associated with each node refer to the result of the game. (-1 = Win for 'X', 1 = Win for 'O'). Some scoring function maps the space of positions to a portion of the real line. For two-player games, this is very often finite and centred around 0, the value awarded to games which are drawn. Note that the scoring function is only explicitly defined for terminal positions, where no further play is possible. These positions are the leaves of the game tree, double underlined in Figure 1 on the previous page. It is however possible to give a meaningful value, called the *game theoretic value* or *game theoretic score* to all the other positions, which is the outcome assuming both players play perfectly from that point on.

As an example, consider the consequences if 'O' plays *A3.* Position P is reached, which can be scored as a win for 'X', since all moves available to 'X' give this result. Position Q can be given a value 1 on similar grounds. Once position Q has been given a score, it can be considered to be a leaf of the game tree; when allocating a score to position R, only its immediate descendants need be taken into account:

The result of a game which has reached position R therefore depends upon which of the available moves is played. At this point we make the simplifying assumption (which is probably fairly well-founded in the case of Noughts and Crosses) that both players are aware of the rest of the game tree and have scored all of the possible future positions accordingly. Since player 'X' chooses which move to play, and by assumption is playing rationally, we assume that from position R he will play *B1.* Accordingly, position R

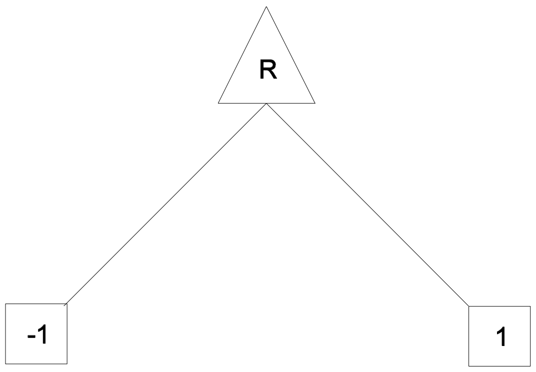

Figure 2: Scoring Position R

is assigned a game theoretic value of -1. Formally, ‘X’ chooses whichever descendant has the *minimum* game theoretic value. On the other hand, ‘O’ prefers the results in the order {win for ‘O’, draw, win for ‘X’}, and so chooses whichever descendant has the *maximum* game theoretic value. This process of scoring nodes from consideration of their daughters’ score is referred to as *backing-up*.

Any two-player zero-sum game of finite length may, in theory at least, be analysed fully in the above fashion. Once the game tree has been deduced, and the leaves scored, the remaining nodes of the graph may be given a game theoretic score by the process just described, termed *minimax* since nodes’ scores are alternately minimised and maximised up the tree. When noughts and crosses is analysed in this fashion, the decision theoretic value of the root of the game tree is shown to be 0, meaning that neither player can force a win. In the last few years, some more interesting games yielded to this approach; Connect Four [1, 86] and Go-Moku [4] have been shown to be wins for whoever starts, while Nine Men's Morris [24] has been shown to be a draw. Analysis of this kind is possible because the first two games have some simplifying features, while the last has a relatively small number of positions. For a large number of games, the sheer size of the game tree means that the amount of computing power required to trace it all the way down to the terminal positions renders such an analysis infeasible with current technology. Table 1 stems from a discussion by Allis, Herschberg and van den Herik [2] of the possibility of analysing games in this fashion. It is that it seems safe to conclude that a full analysis of this sort will never be possible for some games at least, most notably Go, which does not have any simplifying features such as Go-Moku or Connect Four.

A lot of effort has been invested in deducing good game-playing algorithms for these games, especially Chess. Almost all[1] these algorithms work in a similar fashion: they search a certain portion of the game tree and then choose a move based upon this limited evidence. The leaves of the tree searched are not, in general, terminal game positions, so their game theoretic score cannot be precisely determined, but is approximated by what is termed an *evaluation function*. Although the existence and accuracy of such

[1]For games in which the branching factor is so high, and position evaluation stable enough that little can be gained from search, some algorithms do not carry out any search. Backgammon is the classic example — the world champion was defeated as early as 1980 by such a program [13].

Nine Men's Morris ∗	10^{11}
Connect-Four ∗	10^{12}
Kalah(Awari)	10^{12}
Backgammon	10^{19}
Draughts (8x8)	10^{20}
Othello	10^{30}
Bridge	10^{30}
Draughts (10x10)	10^{35}
Go (9x9)	10^{35}
Chinese Chess	10^{45}
Chess	10^{50}
Go-Moku ∗	10^{105}
Renju	10^{105}
Scrabble	10^{150}
Go (19x19)	10^{170}

∗ The game theoretic value of this game is known

Table 1: Estimated Game Tree Sizes of Common Games

a function is vital for the efficacy of the search process, the nature of this function is a highly game-specific matter and not the subject of this text.

The open question which we shall consider is what portion of the game tree should be examined, or, more precisely, *what method should be used to determine which portion of the game tree to search.* This has been the subject of much work by the artificial intelligence community since the question was first raised in 1950 by Shannon [76]. His seminal paper names two main types of search strategies. Type A strategies, which might more descriptively be called *non-selective*, look ahead all possible moves to a fixed depth. Shannon's type B is what is referred to nowadays as the class of *selective* search algorithms. They have the potential to outperform non-selective search, and as Shannon states, to do this they must do the following:

> "1. Examine forceful variations out as far as possible and evaluate only at reasonable positions, where some quasi-stability has been established.
>
> 2 Select the variations to be explored by some process so that the machine does not waste its time in totally pointless variations."

It is a remarkable fact that, whilst the underlying inefficiency of non-selective search is obvious, almost all of the world's strongest game-playing programs are still based, to a greater or lesser extent, on brute force methods that bear more resemblance to Shannon's type A algorithms.

1.1 Non-Selective Search Methods

As the name implies, non-selective search methods spend no time in choosing where to search. This is simultaneously a fundamental flaw and a great strength. It is a fundamental flaw because it means that the vast proportion of their deliberations concern utterly pointless sequences of moves which would not even be considered, far less played, by any human player. It is a strength because it means that all the time available is spent investigating the game tree. One further advantage of non-selective search which is used to great effect is that is that the 'brute force' nature of the search allows for relatively effective parallelisation. By contrast this is not the case for selective search, since (by definition) previous search results are required in order to deduce where further search should be carried out.

A major problem of non-selective searching is the so-called 'horizon effect'. This phenomenon arises from the termination of search and evaluation of a position at a point at which it gives a misleading score. As an example, consider a Chess program which searches to a fixed depth of 5 moves. Naive full-width searching to this depth is likely to lead to a highly implausible fifth move (for example, a queen sacrifice to capture a supported piece). The reason for this is because of the immediate material gain, such a move will lead to a node with the largest positive score, since the search is not deep enough to see that the queen can be immediately recaptured. Just as seriously, if the program is in a position in which search has just revealed that at some future point it will have to incur some loss, it is liable to play any forcing exchanges available, including those which incur a disadvantage, in order to 'postpone the evil hour' and to push the node at which it will incur the loss beyond the search horizon.

An instinctive response to the horizon effect might be to tinker slightly with the position evaluation function to account for recapture, the importance of supporting pieces *en prise* and so on. However, anticipating recapture is very similar to just doing more search, and complicating the position evaluation function generally heralds further problems since every game worthy of serious study has intricacies of its own which are not easily dealt with in this fashion. The fundamental problem remains that positions have varying degrees of stability and so varying degrees of search effort are required in order to deduce position evaluations of comparable accuracy. Any search algorithm which does not model this will suffer from the horizon effect since its search effort will be terminated by some other, essentially arbitrary, criterion.

The original and most primitive non-selective search method is full-width, fixed depth minimax — anticipating *all* possible sequences of moves to some pre-established depth, d. This has complexity $O(b^d)$, where b is the mean branching factor of the game tree, so even with modern computers it is very limited for a game such as Chess (where b is around 35). Many refinements to this basic approach have been developed, to speed up search and to tackle the horizon effect. We now review the most important.

1.1.1 Alpha-Beta Pruning

The most important advance in the area of non-selective search is the development of a technique known as *alpha-beta pruning*, which occurred some time in the early 1960's. A good historical discussion of its derivation and mathematical properties is given by Knuth and Moore [44]. The basic idea behind alpha-beta pruning may perhaps be explained most succinctly by its categorisation as a 'branch and bound' method, although use of that phrase in this context has been the subject of some discussion [48].

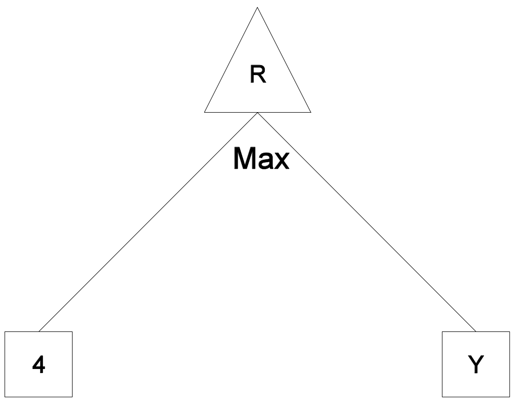

Figure 3: A Lower Bound on Search

The branch and bound principle is illustrated above in Figure 3. If we assume that the left-hand node has been reliably scored as 4, the value of R must therefore be at least 4. The value of Y is therefore only of relevance if it is greater than 4, so it may therefore be possible to search Y less exhaustively without jeopardising the accuracy of the minimax score assigned to R.

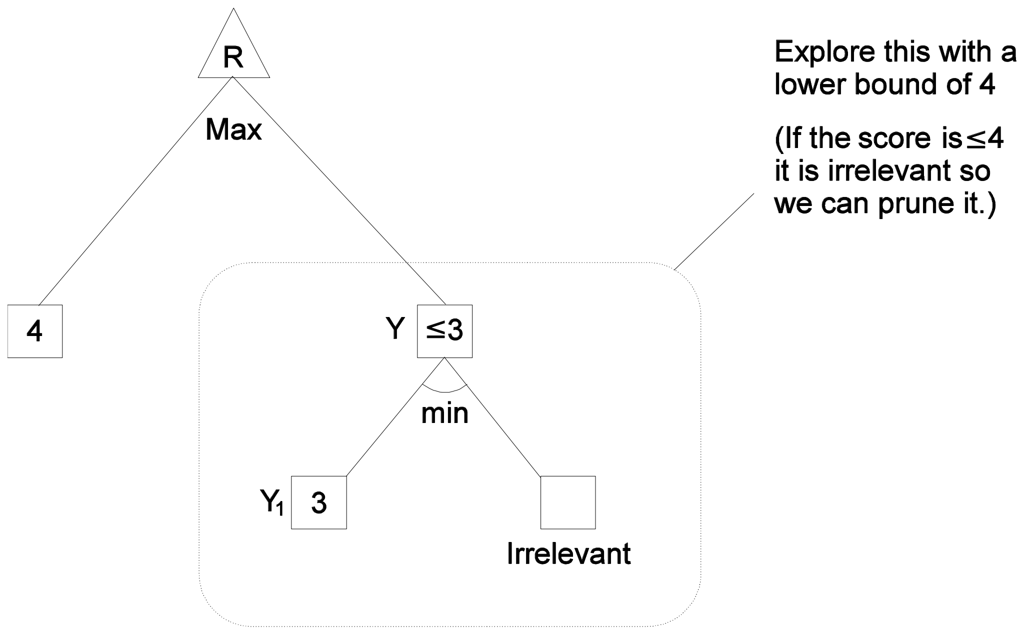
R
Max
4
Y ≤3
min
Y1 3
Irrelevant
Explore this with a lower bound of 4
(If the score is ≤4 it is irrelevant so we can prune it.)

Figure 4: Pruning of a Search Branch

The above figure gives an example of when it is possible to prune a branch of the search tree. If the search of Y_1 shows it to have a score of 3, then the value of Y is at most 3, so the value of R is 4. The values of the nodes $\mathsf{Y}_2 \ldots \mathsf{Y}_\mathsf{N}$ are of no relevance, so they need not be searched. This procedure is referred to as *pruning*, since the $\mathsf{Y}_2 \ldots \mathsf{Y}_\mathsf{N}$ branches are removed from the minimax search tree.

Full alpha-beta pruning is an extension of the procedure above. Instead of a single bound on the value, two bounds are imposed, as shown overleaf in Figure 5. The exact value of Z need only be determined if it is in the interval — or 'window' — $(4, 6)$, assuming that nodes X and Y are scored reliably.

In contrast to the full-width fixed-depth minimax search, the alpha-beta technique is sensitive to the order in which the moves are examined. The

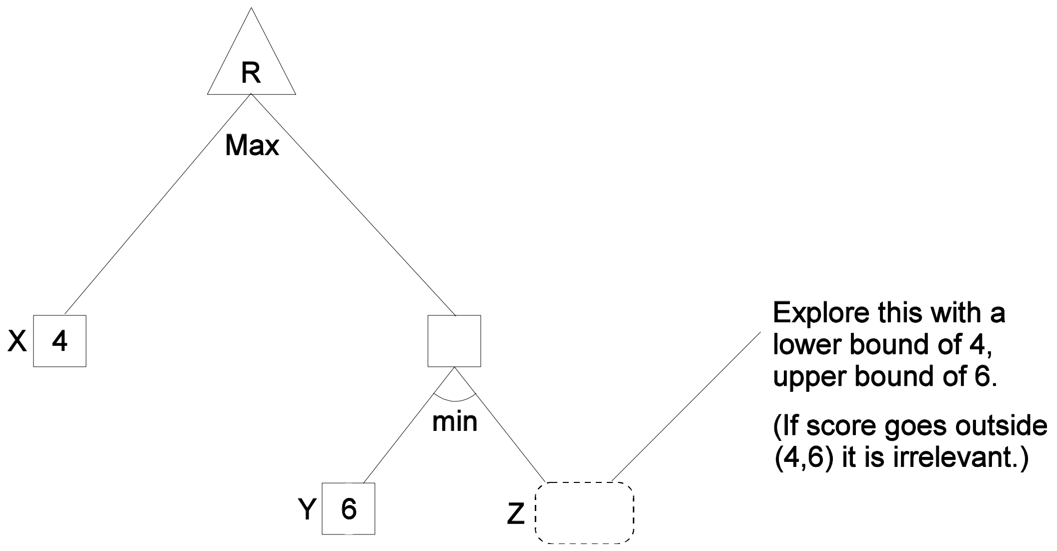
R
Max
X 4
min
Y 6
Z
Explore this with a lower bound of 4, upper bound of 6.
(If score goes outside (4,6) it is irrelevant.)

Figure 5: An Alpha-beta Pruning Window

process of arranging the moves so that the best is looked at first is termed *move pre-ordering*. The 'perfect ordering', resulting in the smallest possible tree, is that which searches the best move first, allowing the strictest bounds to be used to prune the remainder of the tree.

For a game tree of constant branching factor b, a full-width minimax search entails a search of $O(b^d)$ nodes. If we assume a perfect ordering, then alpha-beta pruning requires that $O(b^{d/2})$ nodes be searched if d is even, and $O(b^{(d+1)/2})$ if d is odd, a result first proved by Slagle and Dixon [78]. In their 1969 paper they investigate the performance of dynamic-ordering programs which reorder the moves in an attempt to maximise the extent of the possible pruning. They conclude that:

> "The dynamic-ordering programs are only slightly faster than fixed ordering because the limit of perfect ordering is being approached."

Alpha-beta pruning is very simple to implement and imposes minimal overheads on the time to expand[2] a node, and requires no increase in stack space over simple minimax (d entries). These features, together with the magnitude of the performance gains which it allows (effectively doubling the average depth of a simple minimax search) have caused alpha-beta pruning to become ubiquitous amongst non-selective game-playing programs.

1.1.2 Iterative Deepening

Iterative deepening is a standard feature of alpha-beta search algorithms. The term refers to the iterative increase (by one) of the fixed depth of the alpha-beta pruned search. The first benefit of doing this is that it avoids the issue of how to choose a cut-off depth for the search. If the cut-off depth chosen is too small the search is not sufficiently deep and so moves are unnecessarily overlooked. If a search is started with a cut-off depth which is too great, the search will take too long to complete, and so cannot be completed before the available time is exhausted. This is likely to be even

[2]to generate its children and append them to the search information

worse since the results of a partially completed search are much less reliable than those of a completed search, albeit a shallower one.

The iterative deepening procedure avoids (in a straightforward fashion) the problem of over-estimating the cut-off depth. It initially conducts a search of depth one, then, once this is completed, the depth threshold is increased by one and the procedure is repeated. This procedure ensures that if the search time is exhausted during a search of depth N, there is a completed search of depth $N-1$ available. The obvious disadvantage of this method is that it involves re-searching the nodes at the top of the tree — some of the top level nodes would have been searched N times. This is, however, not such a serious drawback since the cost of the separate searches increases geometrically (on average, by a factor of at least $b^{\frac{1}{2}}$), so that the cost of the last completed search dominates the total time spent searching. Much more important is the fact that the results from each level of search can be used to expedite the next. This is achieved by exploiting the aforementioned sensitivity of alpha-beta to the order in which the moves are searched; since, by and large, a move that is good at depth $N-1$ is likely to be good at depth N, considerable saving can be made by pre-ordering the moves for search at depth N in order of their scores at depth $N-1$.

1.1.3 Null Move Search

This technique arose originally from a need to obtain accurate bounds upon a node's value. Palay [62] carried out a pair of 2-ply searches from a node's position, one for each player, in which two successive moves were made by that player. From this he deduced bounds for his PSVB* algorithm mentioned in Subsection 1.2.5.

The idea was developed by Beal [7, 10] into 'a generalised quiescence search applicable to any minimax problem'. Bounds are obtained in a similar fashion to Palay's method. Rather than being used to guide selective search, they are used to narrow the alpha-beta search window. This method assumes, as does Palay's, that the null move (passing) is not the best move — an assumption that is valid for a great proportion of the time in most standard games.

1.1.4 Singular Extensions

The method of singular extensions, introduced by Anantharaman, Campbell and Hsu [5] is a 'modification of brute force search, that allows extensions to be identified in a dynamic, domain-independent, low-overhead manner'. An extension to a brute force search algorithm is a modification that causes deeper search to be carried out for certain nodes of interest. The word 'dynamic' means that the algorithm makes a choice about which nodes to extend based upon results of the search so far carried out. This is contrasted with other, 'static' extensions, that consider a move in isolation. In the game of Chess, for example, a commonly employed static extension to the search tree is to search more deeply those moves which give check or move out of check. These extensions are successful at increasing performance since they allow more accurate evaluation of some of the terminal nodes of the brute force tree.

Anantharaman, Campbell and Hsu make the point that while search extensions are successful at increasing the performance if applied properly, this is of necessity a domain-specific matter. This is because the choice of nodes at which to conduct a static extension is one that is made by human experts, reflecting the fact that a static extension algorithm encapsulates some knowledge about the game, simple in nature though it may be. Their method does not need game-specific knowledge, since it extends all nodes which are defined as 'singular'. These are positions with a backed-up score which exceeds that of their sisters by at least S. Forced moves are therefore singular, and these are good candidates for deeper search, since they occur during tactical encounters where the position evaluation is likely to be unstable. This method is simple enough not to slow down the search greatly, and so is a practical if rather ad hoc method of adding some degree of selectivity to a brute-force search.

1.1.5 Probcut

This technique, developed by Buro [20] he describes as 'a selective extension of the alpha-beta algorithm'. In a similar fashion to alpha-beta pruning, its function is to prune away uninteresting branches from the search tree, allowing deeper search. In contrast to the alpha-beta technique, it does what is referred to as *forward pruning.* That is, it decides that a branch of the tree is not relevant before it has been searched at all. This can therefore achieve greater performance gains at the expense of possibly pruning away a relevant branch and so deducing a different score from a full-width minimax search.

This method exploits the fact that the evaluation function is fairly stable: to simplify somewhat, if a branch is sufficiently unpromising at depth N it concludes that it is unlikely to be worth searching at depth $N + 1$ and so prunes it at that point. It is assumed that v', the minimax evaluation of a node when searched to depth d', may be used as a predictor for v, the minimax evaluation based upon a search of depth $d > d'$, according to the following simple regression:

$$v = av' + b + N(0, \sigma^2)$$

Once d and d' have been fixed, a data set of positions and search results is collected, and linear regression used to deduce values for a, b and σ. The model is then used to prune branches by ignoring those which have a high probability of falling below the level needed to change the overall minimax value of the search information. After an empirical study into the effectiveness of the procedure in his Othello program, Buro set this cut-off threshold, P, at 0.933.

Buro's claim that Probcut is 'game independent and does not rely on parameters to be chosen by intuition' is difficult to reconcile with the fact that he does not suggest an automated procedure for choosing the search depths d and d' or the cut-off threshold, P. Although he achieved a fairly impressive R^2 statistic of 97%, which provides justification for the use of a normal model, Buro achieved this only by separating the game into 'game phases' (according to the number of moves that had been played). No indication is given about the effectiveness of this approach on other games, or how the 'game phases' might be determined in games other than Othello, of which the great majority have a variable number of moves.

1.1.6 SSS*

This review of non-selective search techniques would not be complete without some attention being given to SSS*, an interesting although rather complicated algorithm due to Stockman [81]. Although not very clear from Stockman's original description of the algorithm, it prunes nodes in a manner which is basically analogous to that of alpha-beta. Rather than doing this in a left to right manner, it keeps an ordered list of the nodes under investigation and so has a greater potential to prune away nodes. The nature of this similarity was first pointed out by Kumar and Kanal [48] some years after Stockman's initial paper.

What *was* clear was the correctness of SSS* — meaning that it deduces the same value as alpha-beta pruning or full-width search — and the fact that whilst alpha-beta looks at nodes pruned by SSS*, the converse is not true. In the light of this it may therefore come as a surprise to learn that SSS* has never been widely implemented by programmers of actual game-playing programs. The reason for this is that the performance gains are at the expense of a large overhead in storage space and bookkeeping costs. Roizen and Pearl [68] proved the two algorithms to be asymptotically equivalent, and reported the ratio of their average complexities for their experiments to be small[3]. The intermediate storage requirement of $b^{d/2}$ nodes therefore renders the original SSS* algorithm greatly inferior to alpha-beta for practical purposes. However, work on understanding SSS* has continued [21, 34, 51], and the investigation of low intermediate storage versions of the original SSS* algorithm remains an active area of research [19, 66].

1.2 Selective Search Methods

All human games players of any skill have an ability to perform selective search. Indeed, the two concepts are virtually synonymous. If asked to explain his thought processes, a human player might begin 'I am investigating what happens if I make move X...'. When asked why *that* particular move, his reply is likely to amount to that fact that he thought it likely to be a good move. In his discussion of type B search strategies, Shannon [76] proposes a simple means of selection akin to a human's use of prior information about the likely merits of the moves available. He suggests 'a function $h(P, M)$,

[3]less than 3

to decide whether a move M in position P is worth exploring', and suggests how such a function might be arrived at for the game of Chess.

This approach to selectivity in searching is a very interesting one, and it is a remarkably underinvestigated area of research. Knuth and Moore mention the idea as an alternative to a fixed-depth cut-off of alpha beta pruning [44], crediting it to R. W. Floyd, although no publication is cited. One possible reason why it has been subject to so little attention[4] is reluctance to work on a system that requires an extra level of domain-specific approximations (move- as well as position-evaluation functions).

We now review the most important methods of selective game tree search. The common aim of all of these methods is to concentrate the search effort upon those branches of the game tree which it is most 'profitable' to consider. To be useful the performance gained from the increased relevance of the nodes examined has to outweigh the performance lost from what we shall refer to as the *meta-calculation* costs, that is, the extra computational burden associated with the selective control mechanism.

1.2.1 Best First Minimax

This technique can be traced back to 1969 when it was mentioned by Nilsson [59] as a special case of the AO* search algorithm. It was rediscovered by Korf in 1987, who initially rejected it on grounds of exponential memory usage, but later began to study its performance on one-player games [46]. It

[4]I am not aware of any research published on this topic.

is based upon what Korf refers to as the *principal variation* of a game tree. This is a path from root down to a leaf every node of which has the same minimax backed-up score. It is a trivial consequence of mimimax backing-up that at least one such path always exists.

The only nodes searched by the best first minimax algorithm are leaf nodes of principal variations. The rationale behind this is that these are the leaves which are most likely to influence the score at root. Korf and Chickering [47] suggest that search be terminated and a move made when the depth of the node being searched exceeds some parameter, the maximum search depth. When making a performance comparison with alpha-beta search, a loose analogy is made between this parameter and the fixed depth cutoff applicable to alpha-beta.

The major drawback of this method is that the benefit achieved by the selectivity is overly dependent upon the accuracy of the position evaluation function. Some appreciation of this may be gained by the observation that best first minimax can be permanently deterred from searching a branch if it is given a very unpromising evaluation. In an extreme case, a branch may remain unexplored beyond depth one, however great the maximum search depth parameter. In an effort to correct this behaviour Korf and Chickering have formulated a hybrid algorithm, termed *best-first extension* search, which initially carries out alpha-beta search to some fixed depth and then spends the remaining time growing the tree by using best first minimax. The resulting algorithm, whilst somewhat ad hoc, outperforms both pure alpha-beta and pure best first minimax, so may be indicative of the improvements to both which are required.

1.2.2 Conspiracy Numbers

Conspiracy number search is an interesting selective search technique originally devised by McAllester [50]. The version described below is influenced by a later paper of Shaeffer [74]. Fundamental to the algorithm is the notion of a *conspiracy*. This is a set of leaves of a game tree, which, if they were all to change their evaluations in a coordinated fashion, could result in changing the value of the root. Nodes P & Q in Figure 6 overleaf, for example, make up a conspiracy since if the scores of both of these nodes were changed to 5 or more, then the value of the root node would be increased.

Nodes T & U are another example of a conspiracy. If they were both to take on a value of 3 or less, then the score of root would be decreased. The *conspiracy number* of a tree is defined as the minimum *size* — i.e. number of nodes — of any conspiracy of the tree. The above tree therefore has conspiracy number of 1, since the single node R is a conspiracy.

The conspiracy numbers algorithm aims to search nodes in a way that increases the conspiracy number of the search information as quickly as possible, by targeting the nodes involved in the smallest conspiracies. The justification for this is that it concentrates search on those parts of the search information which have the greatest influence upon the score of the root node, and so are in some sense the most important to search. The need to carry out

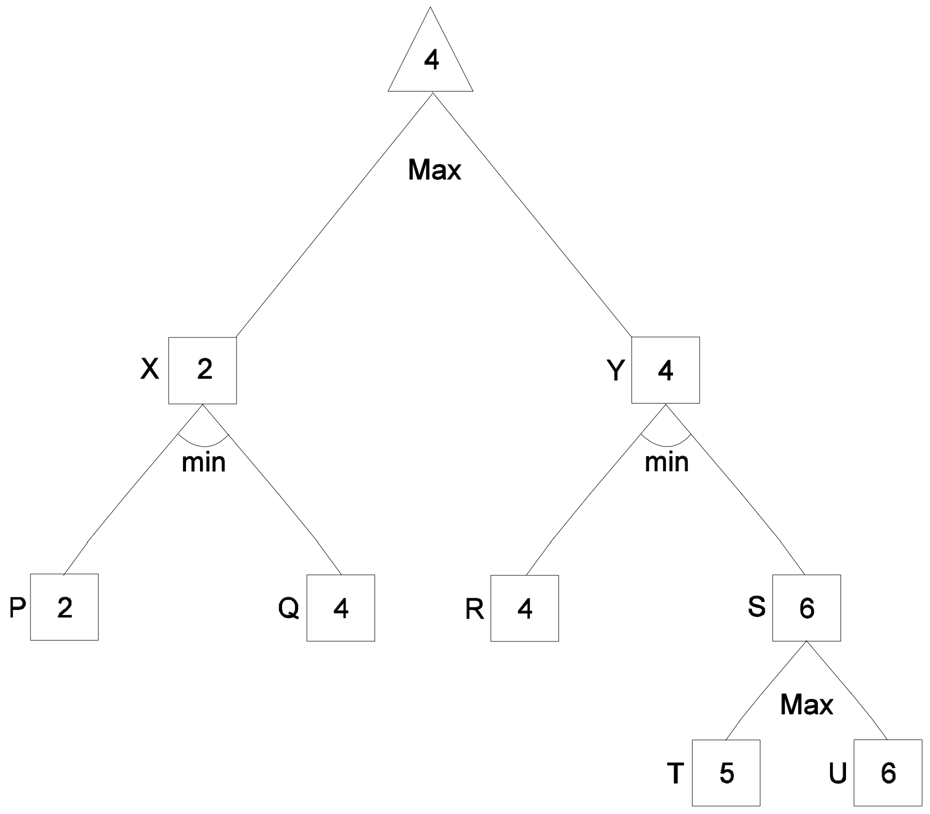

Figure 6: A Graph with Conspiracy Number 1

deep searches of the game tree stems from the imperfection of the evaluation function, and so it is desirable to base a choice of moves upon a tree with a large conspiracy number, since this is to a greater degree 'insulated' from the effects of inaccurate node evaluation function.

1.2.3 B*

An early and important development in the history of non-selective search was Berliner's [12] B* tree search algorithm, published in 1979. The existing algorithm with which it had most in common was the ‘bandwidth’ algorithm devised by Harris [33]. Both of these early selective search algorithms arose from an attempt to overcome the horizon effect. Berliner’s key idea, which he had in 1972 (see [16]), was to score a node not with one, but with *two* scalars. The use of an upper and lower bound on a node’s score allows some assessment to be made of the stability of that node’s evaluation, and hence its relative importance as a subject of further search.

As search progresses, the two bounds on a node’s score gradually converge. Eventually, the lower bound of one move from root is at least as great as the upper bound of all the other possible moves. At this point it becomes possible to conclude — if the bounds are correct — that no further search is required and a best move has been found, even if its exact score has not yet been established. The B* algorithm chooses between the following two strategies:

1. PROVE BEST, which aims to raise the lower bound of the move with the greatest upper bound.

2. DISPROVE REST, which aims to lower the upper bound of one of the other moves.

In practice, the B* algorithm has not found widespread application. The main reason for this is the acknowledged difficulty of deducing functions to generate reliable upper and lower bounds. A later revision of it by Palay [60] deserves mention as a milestone in the history of game tree search; this was the first game-playing algorithm to score each node with a (uniform) probability distribution. In his paper on B*, in which he makes a slight improvement upon Berliner's original formulation, Palay concludes with an insight that was to lead to his later work on a more advanced probabilistic adaptation to the original B* algorithm.

> "It seems quite clear that humans maintain additional information about a node in a search other than one or two values."

1.2.4 PSVB*

Under Berliner's guidance, and building upon B* search, Palay went on to deduce the PSB* [14] and PSVB* [61, 62] algorithms, which score each leaf node with a continuous probability distribution. These are backed up using the probabilistic equivalent of minimax:

$$\begin{aligned}\text{For MAX nodes : } \quad P[\text{Parent} \leq x] &= \prod_i P[\text{Child } i \leq x] \\ \text{For MIN nodes : } \quad P[\text{Parent} \geq x] &= \prod_i P[\text{Child } i \geq x]\end{aligned}$$

The distributions thus given to top level moves are used in an analogous fashion to the upper and lower bounds of the original B* algorithm. The PSB* algorithm attempts to *select* a move which 'dominates' the others available with a certain probability. In the PSVB* algorithm, a *verification* stage was added, in which the opponent's replies to it are examined to establish whether it is indeed the best move. The motivation for the emphasis on finding 'the best' move stems from the analogy with B* and does not appear to have been examined by Palay in his otherwise groundbreaking approach to selective search.

1.2.5 Probabilistic B*

Recently, Berliner and McConnell [16] have built upon the work of Palay to deduce an algorithm that they term 'probability based B*'. In their algorithm, each node is scored with the following triple: (*optimistic*, *best guess*, *pessimistic*) and with the probability distribution *optprob*. The best guess value of a leaf node is the result of a standard fixed depth alpha-beta search. To derive the optimistic values a similar search is carried out in which the opponent's first move is a forced pass. The pessimistic values are derived by the same process, since one player's optimistic value for a node is his opponent's pessimistic value. This procedure has thus overcome the main obstacle to B*'s successful implementation.

Nevertheless, the probabilistic B* algorithm still leaves certain vital theoretical questions unaddressed. For example, on the most fundamental issue of all, how to select the next node to expand, there is little by way of formal justification. One is, however, forced to admire the authors for their candour about this:

> "The expression that computes *TargetVal* is (*optimistic*(2ndBest) + *best guess*(Best))/2. We have no theory for this expression, but it seems to do an excellent job . . . "

1.2.6 MGSS*

Russell and Wefald tackled the problem of how to perform a selective search with a new rigour. They explain it as a problem of 'constrained meta-reasoning' [71], of reasoning about where to search under a time constraint. Their perspective was more theoretical than that of previous selective search authors, discussing the origins of their ideas to an impressive degree of generality. They observe that it is (theoretically) desirable to have a meta-meta-reasoning policy to control the actions of the meta-reasoning policy, and so on. Their conclusion is that, for the general case, the best that can be achieved in practice is a policy which is optimal within a restricted class of policies, which they term *limited rationality*. Their (at times, almost philosophical) text [72] is recommended to the reader who is interested in this problem. As we shall discuss further in Section 6.1 this dilemma cannot be addressed directly by classical dynamic stochastic control.

The approach taken by Russell and Wefald is to break down the meta-calculation into manageably small steps. To decide which of these should be carried out, the 'meta-reasoning' level (i.e. search control policy) estimates the value of carrying out each search step, and selects the one with the greatest expected value. For the sake of tractability, Russell and Wefald [70] assume a search step to be the expansion of a single node of the search information. They limit the control policies considered, thus ensuring the tractability of the decision problem, with the help of the following two assumptions:

Meta-greedy Assumption: This assumes that the meta-meta-reasoning will be carried out by a one-step lookahead policy. Since maximising over all possible *sequences* of computational steps is likely to be infeasible in practice, the maximisation is instead carried out over all possible single computational steps. This is an approximation since it disregards how the computational steps chosen may influence the availability and utility of future computational steps [5].

Single-step Assumption: This allows the value of searching to be efficiently approximated. It evaluates the benefit gained from expanding a node as if no further searches will be carried out. It results in the following formula for $V(S_j)$, the value of expanding node j:

$$V(S_j) = \text{Value of Best Move available after } S_j - \text{Value of current Best Move}$$

The MGSS* algorithm makes both of these assumptions. The idea behind it is to expand at every stage the node with the greatest 'utility'. The utility of an expansion is simply its expected value with a correction to take into account the time cost:

$$U(S_j) = E[V(S_j)] - TC(S_j)$$

[5]Russell and Wefald [70] point out that if no assumption is made about how the computational steps are scored — such as the single-step assumption — then the Meta-greedy assumption need not represent a simplification, since — in theory, if not in practice — the computational steps could be scored so as to take into account their interactions with all possible future computational steps.

The leaves are scored with normal probability distributions. Their mean and a standard deviation are allocated by a position evaluation function trained on a large sample of positions so as to correspond accurately to the change in score when the leaf is further evaluated. Russell and Wefald comment about the assumption of normality that they found some evidence of systematic error, and so suggest that a high performance program might benefit from use of a finite mixture of normal distributions.

1.2.7 BP

The BP game-playing algorithm is a recent method of selective search, which has achieved very promising results against alpha-beta opponents [80]. The recent paper of Baum and Smith [6] contains the following statement about selective game tree search:

> "Our approach of stating a Bayesian model of uncertainty, describing how we estimate utility of expanding given trees within this model, and giving a near linear time algorithm expanding high utility trees can be seen as formalizing this line of research."

This is quite a claim. The BP algorithm has much in common with MGSS*, and in light of this, and the acknowledged influence of Russell and Wefald on its development, we now look at how it differs from MGSS*.

The major operational difference between the two algorithms is that of complexity. While the MGSS* algorithm is O($n^{\frac{3}{2}}$) in the number of nodes of the search tree[6], the BP algorithm requires O($n \log n$). This important saving is achieved at the expense of selectivity by expanding a set of several leaves (called a "gulp") at a time, rather than expanding them individually. This reduces the number of separate control decisions and the calculations necessary to percolate back up to the top of the tree the new information found by the node expansions.

The BP model of utility has much in common with that of the MGSS* algorithm, but is rather more advanced. Denoting by $S()$ the current score, and $S'()$ the score after all the leaves of the tree have been expanded, the expected utility with respect to move i, Q^i, is defined as follows:

$$Q^i = E[S'(\text{Best move})] - S(\text{Move } i)$$

We observe that for the move i^* which is the current best move, Q^{i^*} is a measure of the utility to be gained from expanding all the leaves. Baum and Smith argue that a search which decreases this is useful, since it indicates that this brings closer the point when searching can stop, having "extracted most of the utility in the tree". Conversely, if it increases, then this heralds the prospect of gains with the next expansion, and so such leaves are useful for the point of view of extracting utility from the tree. The measure of relevance of a leaf that they use is therefore defined to be the expected absolute length of step in Q^{i^*}, referred to as Q Step Size, QSS.

[6]This estimation is due to Baum and Smith [6].

The BP algorithm proceeds by expanding those leaves with the highest QSS all in one go (gulp), and then evaluating the consequences. The performance of BP is strongly dependent upon the 'gulp size' parameter, which fixes the proportion of leaves to expand each time. The major advantage of the use of QSS as a measure of utility over Russell and Wefald's suggestion is that it caters for interactions (conspiracies) between the leaves.

Another advance on MGSS* is that BP uses discrete probability distributions to score the nodes. By making an appropriate choice of masspoints, this allows for essentially unrestricted modelling of distributions. The scores are percolated up the tree with the formulae mentioned in Subsection 1.2.4 and first applied to game-tree search by Palay. Baum and Smith [6] report the results of an interesting experiment into the significance of this. In a series of Othello games between two otherwise identical alpha-beta programs with an evaluation function that returned a probability distribution, the one which used probabilistic percolation was shown to beat the one which treated the expectation of the value returned as a static score and then used standard minimax.

1.3 Statistical Models

A game is completely determined by a set of rules relating to the board, the pieces and so on. These define the legal moves from every state, and also govern the evaluation of terminal game tree nodes by defining how games are scored. The game tree can be calculated from these rules. In this work, I have avoided any study of real games played for amusement by humans, since, even if the rules of the game are very simple, such games tend to have complicated game trees[7].

Instead we concentrate on artificially created games, the rules of which refer directly to the game tree; there being no actual ‘game’ or ‘pieces’, but just the game tree itself. The complications of real games confuse the comparison of search algorithms, and lean heavily towards empirical as opposed to theoretical justification. Many evaluation functions have been the subject of extensive research and, for popular games at least, are of not insignificant commercial value. Authors are therefore understandably reluctant to publish them, a step which is necessary if proper comparisons are to be made between different search algorithms.

Even within one domain, empirical evidence about the efficacy of search methods must be interpreted with care, since the strength of any game-playing program can be increased by countless adaptations useful to that particular game (opening books and endgame databases being perhaps the two most universal). These are of themselves of great statistical interest, but are not directly relevant to searching, and so are outside the scope of this text.

[7]Some would argue that for a game to be of interest to humans, it must have both of these properties. The game of Go is the example *par excellence* of this duality.

1.4 Summary

The mainstream of game tree searching has deviated surprisingly little from the non-selective paradigm first suggested by Shannon in 1950. The discovery of the alpha-beta pruning technique described in Subsection 1.1.1 may have been the earliest refinement to minimax search, and is certainly the most important. Iteratively deepened alpha-beta search is currently almost ubiquitous among the top game-playing programs. A myriad of other refinements and new techniques have been suggested to search game trees, few of which seem to have found broad acceptance. Almost invariably, empirical work done supporting each new technique has shown it is superior in some way to previously described methods. However, the results are typically limited to one or maybe two different games. More seriously still, there has, typically, been very little done by way of derivation or other explanation of the algorithms' origins, other than in the broadest terms. Although it is understandable that research has been empirically driven to such an extent, the lack of a theoretical basis to much of the work that has been carried out is most unfortunate, since such studies can give only scant indication of the wider applicability of the methods presented, and provide no basis for a wider theory of game tree search.

Of all the refinements so far suggested to improve alpha-beta, the Probcut method described in Subsection 1.1.5 is arguably the most successful. It presents an interesting halfway house between standard 'brute force' techniques and real selective searching. It is instructive to consider the similarities between Probcut and the latest methods of selective search. They all carry out selection based upon a probabilistically derived ('trained') evaluation function. Probcut has much less refined selectivity but because of this is much faster than proper selective algorithms. Probcut is only applied below a certain depth, and so is 'bolted-on' to a standard full-width search tree. In this way it is similar to the *best-first extension* search described in Subsection 1.2.1. The general finding that the performance of selective search algorithms can be improved by prepending full-width search of a relatively small depth is intuitively reasonable, since if it is clear from the outset of searching that *whatever search reveals*, a particular set of moves is very likely to be worthy of investigation, then the most effective way to search them will be in a fast (i.e. nonselective) manner. Put another way, selective searching is only an advantage for nodes where there is doubt over their relevance. Nodes which are very close to root therefore do not merit selective searching, since it is *a priori* very likely that they will be worthy of search.

Shannon [76] himself admitted that the nonselective search strategy outlined in his seminal paper has 'certain basic weaknesses', which is still the case, in spite all the various refinements suggested. With the advance of computing technology, old constraints on memory and C.P.U. usage are receding somewhat in importance, and so sheer speed alone is less important than it was as a criterion for an algorithm's performance. The implications of this are that more work must be done on the theoretical underpinnings of game-playing algorithms. This realisation no doubt goes part way to explain the increased attention being paid to the topic of selective search. The authors of the MGSS* and BP algorithms deserve special mention for their success at addressing the greater theoretical challenges posed by selective search algorithms. Whilst both algorithms, inevitably, resort to some simplification, it is commendable that they explain and explicitly state their assumptions.

This thesis tackles the problem of game tree search by drawing upon the methods of dynamic stochastic control to apply a new rigour to the area of game tree searching. It is therefore based upon simplified mathematical models rather than real games, and so presents theoretical rather than empirical results. Whilst this is a disappointment in the sense that the results derived are of less immediate practical use, it is a necessary price to pay for the increased generality of the results provided hereafter. The primary goal of this thesis is to inspire a more systematic study of game-playing and game tree search by both statisticians and artificial intelligence experts alike.