Dynamic Stochastic Control:
A New Approach to
Tree Search & Game-Playing

Robin J. G. Upton[†]

Ph.D. Thesis

Statistics Department[‡]

University of Warwick, UK

23 April 1998

Ecclesiastes 3:6 – A time to search and a time to give up.

MSC Classifications: 90D40, 93E20, 90D50, 90D43, 90C40, 90D05, 90D15, 90D60, 93E05, 90D10, 68T99, 90C39

Keywords: Tree Searching, Game-playing, Game-theoretic Models, Optimal Stochastic Control, Artificial Intelligence, Fuel Control Problems, Dynamic Programming

[†]www.RobinUpton.com [‡]www.warwick.ac.uk/statsdept

In Memoriam Paul Hobley.

Science is all about looking for unifying theories even if they are not there.

P. Hobley (23/4/93)

Contents

List of Figures

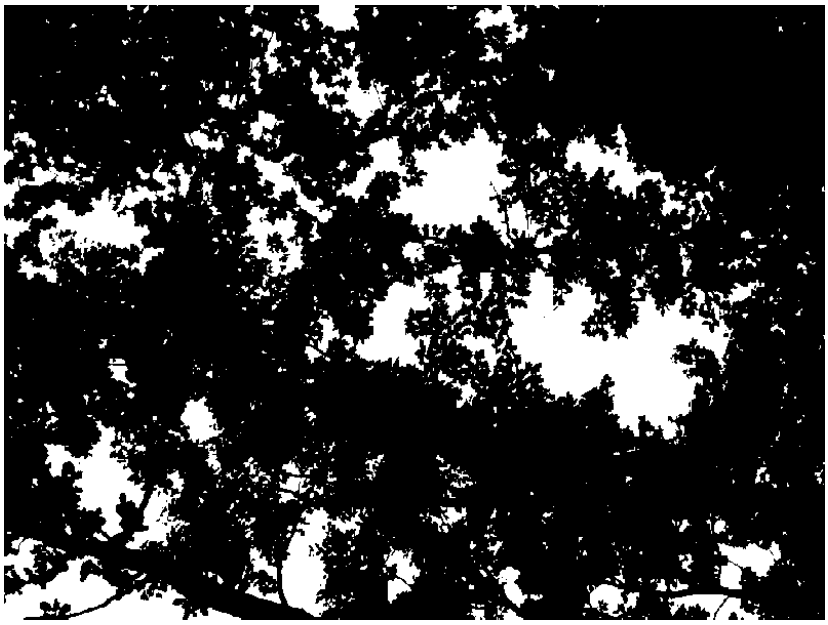
List of Tables

Acknowledgements

I am grateful to the Warwick Statistics Department staff, particularly Saul Jacka, my supervisor. Richard Charlton computerised the figures for me. Ivan Lai & Graham Upton helped me with proof-reading. My examiners, Jim Smith, and especially Kevin Glazebrook increased the accuracy and readability of this work with their constructive criticism.

Thanks are due entirely to God.

This work was supported by EPSRC grant #94003733.



Declaration

Except where otherwise indicated, all the work presented in this thesis is, to the best of my knowledge, original and unpublished.

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.

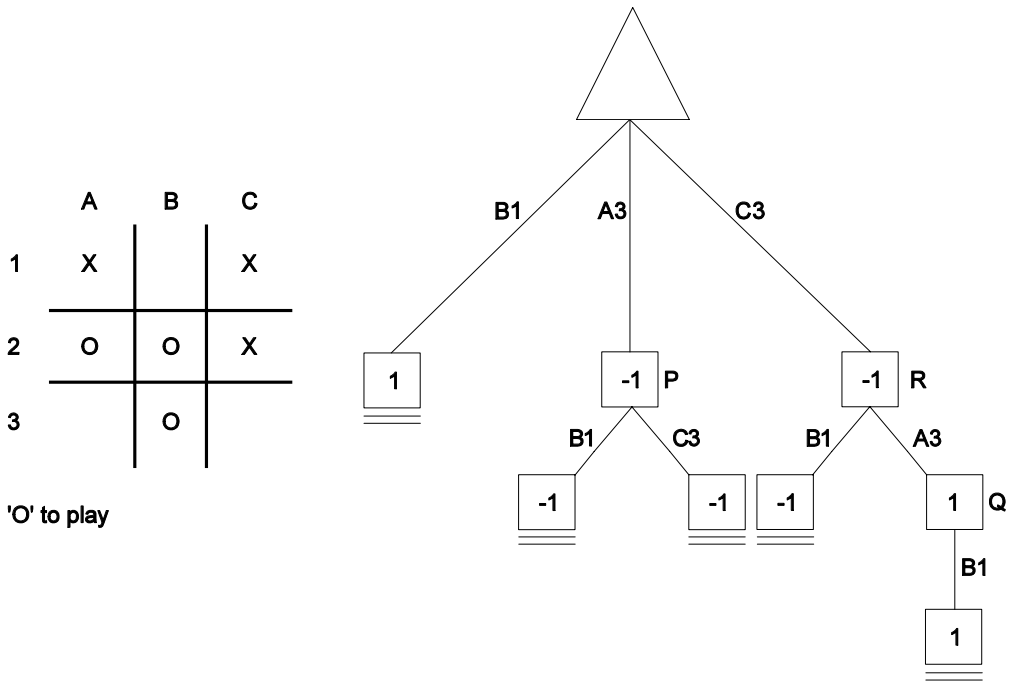


Figure 1: A Noughts & Crosses Position and Corresponding Game Tree

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

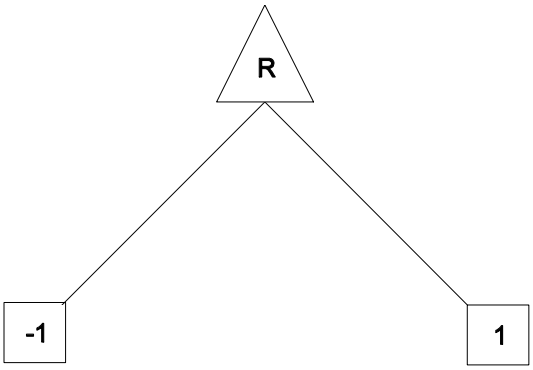


Figure 2: Scoring Position R

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].

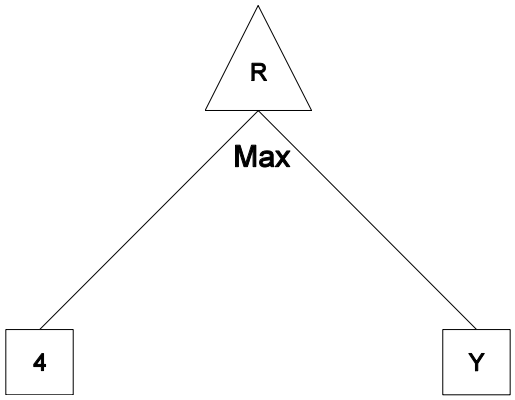


Figure 3: A Lower Bound on Search

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.

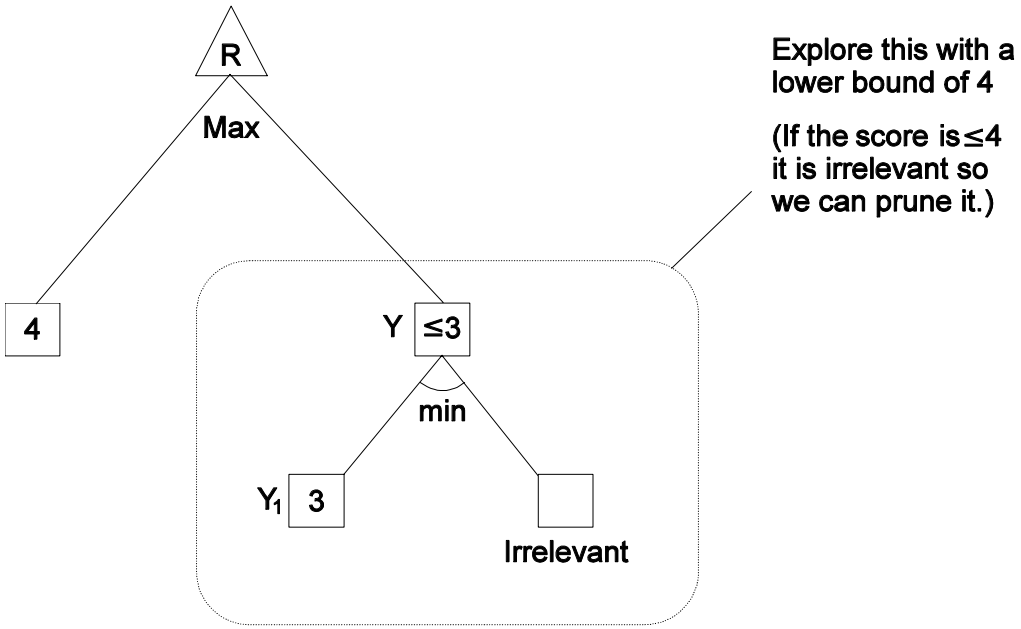


Figure 4: Pruning of a Search Branch

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

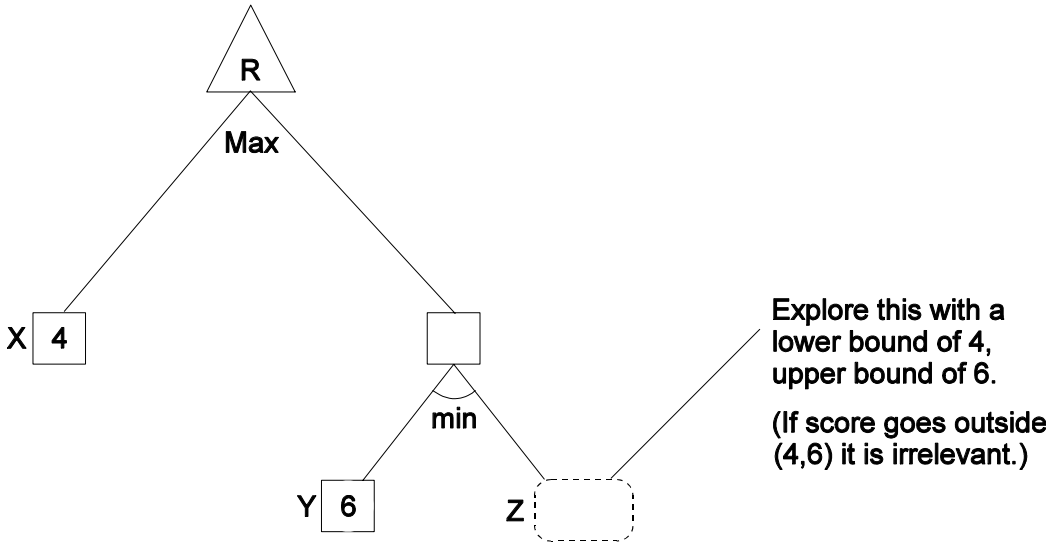


Figure 5: An Alpha-beta Pruning Window

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

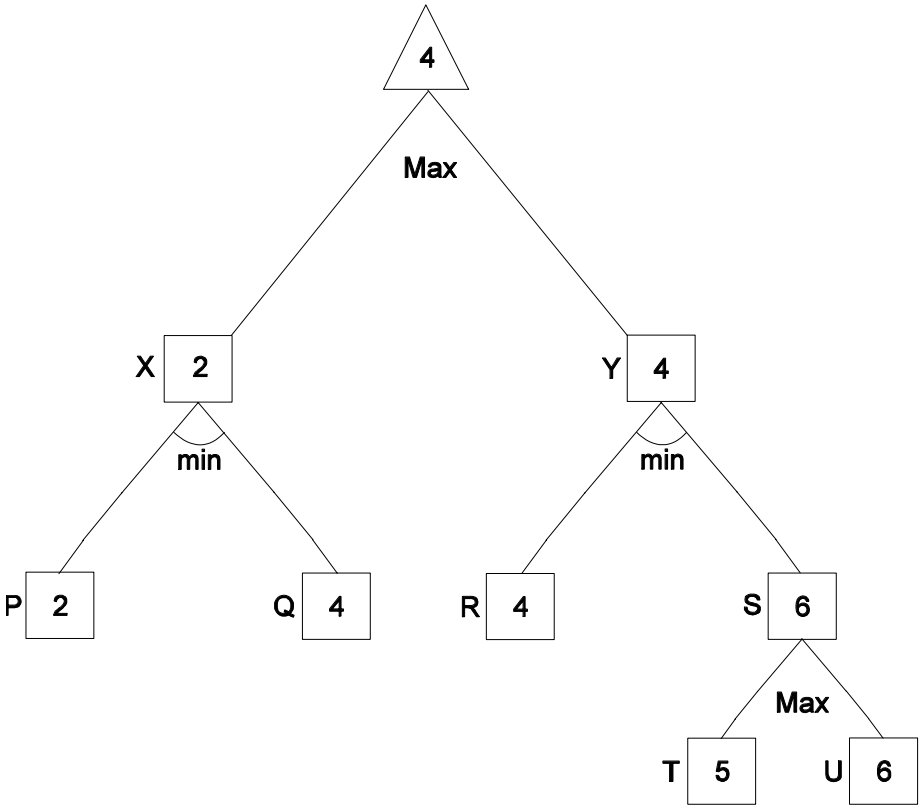


Figure 6: A Graph with Conspiracy Number 1

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:

$V(S_j) =$ Value of Best Move available after $S_j -$ 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 similar-

ities 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.

2 Markov Chains

For a fully satisfactory model of a game being played by two agents, the statespace must be rich enough to represent not merely the state of the game[8], but also the amount of time left and a representation of the 'mind' — be it human or silicon — of each of the two agents (i.e. which portion of the game tree they have investigated, and how they have evaluated it). Human gameplayers have comparatively little difficulty taking into account their opponent's 'state of mind'. Bluffs, trick plays and psychological ploys are common currency among human game players, as is playing on the opponent's weaknesses. All these are strategems which involve not merely the state of the game itself, but also inferred knowledge about the opponent's state of mind.

When in a position which it has already established to be a game-theoretic draw, Schaeffer's draughts program, Chinook, chooses the line of play it estimates to be the most difficult for its opponent [75], so as to maximise its chance of winning through a blunder on its opponent's part — an idea he first suggested in [74]. This is a very primitive step towards the goal of properly modelling the opponent's deliberations, but is unmistakably a step in that direction.

As far as I am aware, no work has yet been published about a game-playing program, either actual or virtual, which involves modelling its oppo-

[8] typically the board position and whose move it is

nent's model[9]. Such meta-modelling would seem to lie beyond the realms of what it is currently useful to implement in a game-playing program. However, its inclusion in a program would have useful consequences, such as allowing the program to bluff and to exploit its opponent's known weaknesses. It is therefore my confident prediction that its importance will become apparent as computing power increases, and a point will be reached at which its implementation becomes desirable. This view is broadly supported by Berliner, Goetsch, Campbell and Ebeling [15] who conclude (for the game of Chess) that as computing power increases, so does the need to deploy it with greater selectivity.

Note that even a meta-model is an approximation. In fact it is impossible to derive a completely satisfactory model of the state of the game, because of the need to have a full model of the opponent's state of mind; since we must assume that the opponent is in turn is modelling our state of mind, we need to model his model of our model, leading to an infinite recursion. There is an obvious parallel here with Russell and Wefald's comments about how the need to optimally control meta-calculations incurs meta-meta-calculations, and so on. Their response to this recursive dilemma was to adopt the meta-greedy assumption, which requires the meta-meta-calculations to be of a particularly simple nature. Once this issue eventually becomes relevant to the playing strength of programs, a compromise of this kind seems likely to

[9]The closest reference I have found to such an idea is Korf's [45] study of the alpha-beta technique for players who use different evaluation functions.

be an adequate solution for many games[10].

2.1 Modelling Game Trees

We now consider the essential properties of a game tree. For the purposes of developing a game-independent model of the game-playing process, a flexible if simple mathematical model is to be preferred to a carefully carried out empirical study of some specific game or games. In choosing a model, a balance must be struck between simplicity on the one hand and descriptive power on the other.

As a first concession to simplicity, we assume the game tree to be, in fact a proper *tree*, as opposed to a DAG (directed acyclic graph). This is a standard assumption. Various authors have achieved minor speed-ups in their domain specific game-playing programs by the incorporation of a 'transposition table'[11], capitalising upon the fact that certain sequences of moves can be interchanged and still lead to the same position. Although no systematic study of this has yet been carried out, it seems likely that this is a second order consideration the importance of which does not justify its inclusion in the model at this stage.

The essential property of game trees, which underpins all search algo-

[10] As programs become more advanced, it will be interesting to see how the importance of such meta-modelling will vary depending upon the game involved. It seems likely to be of greatest importance for games of imperfect information, such as Bridge or Scrabble.

[11] A hash table of previously searched positions.

rithms, is that there is a correlation between evaluations of nodes at different points on the tree. Specifically, a node with a high score is likely to have, by and large, descendants which also have high scores. This is fundamental if meaningful information is to be inferred from partially explored game trees.

One simple method in which a tree with this property may be defined is by directly relating the scores of the parents and the children. Two important ways in which this can be done are deducing the parents' scores from those of their children ('upward percolation'), or generating the scores of the child nodes from the score of their parents ('downward percolation'). The former approach is, in one sense, the most natural, since it reflects how game theoretic values are actually defined. However, the consideration of future searches requires analysis of the possible children given the parent node, rather than the other way round, so this is not a convenient mathematical formulation to analyse[12]. Accordingly, most of the models we consider will assume downward percolation of node scores. We shall use a simple additive error structure: the difference between a node's score and its child's score is a random variable of known expectation.

[12]Some very rudimentary models with upward percolation of scores have been constructed, to analyse the minimax backing-up procedure. This approach was first applied by Nau [55, 54, 56, 57], and subsequently used by Pearl [63, 64] and Beal [7, 8].

2.2 A One-player Search Model

A one-player model allows for very substantial simplifications on the general framework set out above, since it avoids the difficult issue of modelling the opponent. The state is made up of the *game tree*, G, the remaining *search time*, τ, and the subtree of G which has been searched, termed the search *information*, L. Since $G \backslash L$ is unknown we exclude it from the state. The control decision to be made is a choice between moving and increasing the search information by spending one unit of search time to expand a leaf of the information.

Consider a one-player game in which on h occasions, a choice must be made between two alternative moves. With the added condition that each attainable position may be reached by a unique sequence of moves, the corresponding game tree is a binary tree of height h. Each one of the 2^h possible different terminal positions has a score associated with it. The player's goal is to terminate the game at a node with as great a score as possible.

At the outset, the only score that is known to the player is the score at root, which we assume without loss of generality to be 0. Upon making a move, the player becomes aware of the score associated with the node at which they arrive. The player has a limited amount of time, τ, to spend conducting an exploratory search of the game tree. The only nodes which may be searched are daughters of a node the value of which is already known. Therefore, the actions available are *making a move* and *expanding a leaf of the search information*.

The effects of making a move are shown below. Making a move rules out half of the end positions which are currently possible, while moving closer to the other half. It therefore corresponds to discarding half of the game tree, while the root of the remaining half becomes the new overall root.

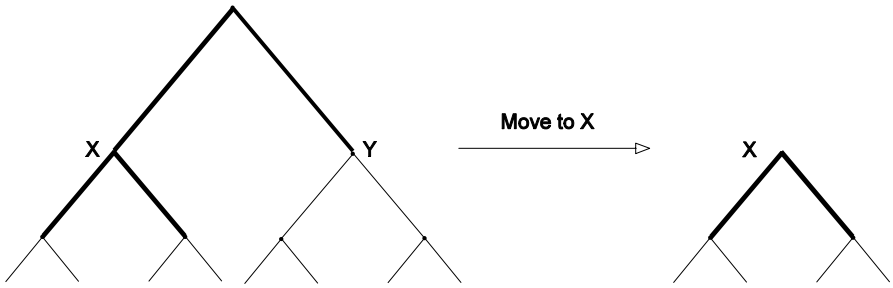


Figure 7: Making a Move

The distribution of scores among the nodes of the game tree is random, but the nature of the uncertainty is known *a priori*. There are in fact game-specific patterns of correlations between node scores at different heights – as exploited in the game of Othello by the Probcut model of Section 1.1.5. However, at this point we make the simplifying assumption that the game tree is a Markov field[13]. Specifically, the model assumes that the score associated with a parent node is always the mean of its two daughters, so one is better by a random variable, Y, and one is worse by the same amount. Since this is a one player game, there is an optimal policy amongst the class of non-randomised policies. In our discussion of one-player games we shall therefore

[13]That is, conditional upon knowledge of the score associated with its parent, a node's score is independent of the scores of any of its other ancestors.

only consider deterministic policies from now on, so in ensuing analyses of one-player games, any policies mentioned may be assumed non-randomised.

We now make another observation about the optimal policy. Let us define the set of policies, S, to be those which only ever make a move down the game tree if it is part of an uninterrupted sequence of moves to an unsearched node or to the end of the tree. The below result implies that there is an optimum policy in S.

Theorem 2.1 *For every policy, $\pi \notin S$, there is a policy $\pi' \in S$ which obtains the same payoff.*

Proof: Consider a policy $\pi \notin S$. There must therefore be states in which it performs actions $\{m_1, \ \ldots \ m_n, \ s\}$ where $m_1, \ \ldots \ m_n$ are a sequence of move actions that do not reach an unsearched node, and s is the immediately subsequent search action. Define π' to mimic π except when π performs such a sequence. When this occurs, let π' take actions $\{s, \ m_1, \ \ldots \ m_n\}$. Since the action sequence $\{m_1, \ \ldots \ m_n\}$ does not move to an unsearched node, it yields no information, so policy π', which obtains the same payoff as π, is admissible. ■

This result is in line with intuition: since searching reveals information without restricting future choices whilst moving to an already searched node does exactly the opposite, it is to be expected that the optimal policy will search first and move later. The case of moving to an unsearched node is different, since it reveals that node's score and so makes it cheaper to search that node's descendants.

Policies in S never move down the tree unless as part of a sequence of moves that terminates either at the bottom of the tree or at an unsearched node. That is, they take an action from the following set: {*Move beyond* i, *Expand* i, *Move to* e *and terminate*}, where i is a leaf of the search information, and e is a leaf of the search information with height 0.

The conditional independence of a node's score given the score of its parent means that the scores of the leaves of the search information comprise a sufficient statistic for inference of the unknown values. The leaves are therefore the only nodes of the search information which are relevant to future search. Hence, the tree of search information may be represented by a vector of leaves. The statespace may therefore be summarised by *the search time left*, τ, and *the leaves of the search information*, $\mathbf{L}$. Suppose the game starts at the root of a binary tree which has height h. We shall address the problem of how best to spend a number $\tau \leq h$ units of search time.

As demonstrated above, for the purposes of deducing optimal policy the search information may be summarised by its leaves, so, if it has N leaves, then it may be written $(L_1 \ \ldots \ L_N) \equiv \mathbf{L}$. A leaf i of height h_i and score v_i shall be denoted $L_i = (h_i, v_i)$. Every nonterminal node L_i has two daughters, which we denote $L_{(i,1)}$ and $L_{(i,2)}$. Since they are one level further down the tree:

$$h_{(i,1)} = h_{(i,2)} = h_i - 1$$

Each node's Y_i value is an independent and identically distributed symmetric random variable. We shall use κ to denote $E[|Y_i|]$, the expected difference

between a node's score and those of its daughters. The scores of $L_{(i,1)}$ and $L_{(i,2)}$ satisfy the following:

$$v_{(i,1)} = v_i + Y_i \qquad\qquad v_{(i,2)} = v_i - Y_i$$

Define $V_{M_i}(\mathbf{L}, \tau)$ to be the value of the game $(\mathbf{L}, \tau)$ subject to the constraint that the next action taken will be to *move* beyond leaf L_i, or, if L_i is a terminal node, to terminate:

$$V_{M_i}(\mathbf{L}, \tau) = \begin{cases} E[V(L_{(i,1)}, \tau)] & | \quad h_i > 0 \\ v_i & | \quad h_i = 0 \end{cases}$$

Further define $V_{S_i}(\mathbf{L}, \tau)$, for $\tau > 0$, the value of the game $(\mathbf{L}, \tau)$ subject to the constraint that the next action taken will be to *search* beyond leaf L_i, or, if L_i is a terminal node, to discard a unit of time:

$$V_{S_i}(\mathbf{L}, \tau) = \begin{cases} E[V(L_1 \ldots L_{i-1}, L_{(i,1)}, L_{(i,2)}, L_{i+1} \ldots L_N, \tau - 1)] & | \quad h_i > 0 \\ E[V(\mathbf{L}, \tau - 1)] & | \quad h_i = 0 \end{cases}$$

2.2.1 Optimal Policy for $\tau \leq h$

Lemmas 2.2, 2.3 and 2.4 all use the following induction hypothesis. The last of these will show that if it is true for $\tau = N$, then it is true for $\tau = N + 1$. We first note that it is true for $\tau = 0$.

$$V(\mathbf{L}, \tau) \leq \max_j \{v_j\} + \tau\kappa \tag{1}$$

Lemma 2.2 *If inequality (1) holds for $\tau = N$, then for any i:*

$$V_{S_i}(\mathbf{L}, N) \leq \max_j \{v_j\} + N\kappa \tag{2}$$

Proof: For any i:

$$\begin{aligned}
V_{S_i}(\mathbf{L}, N+1) &= E\left[V(L_1 \ldots L_{i-1}, L_{(i,1)}, L_{(i,2)}, L_{i+1}, \ldots L_N, N)\right] \\
&\leq E\left[\max_{j \in \{1 \ldots i-1,(i,1),(i,2),i+1 \ldots N\}} \{v_j\}\right] + N\kappa \,\text{from (1)} \\
&\leq E\left[\max\left\{\max_{j \in \{v_{(i,1)}, v_{(i,2)}\}} \{v_j\}, \max_{j \neq i}\{v_j\}\right\}\right] + N\kappa \\
&\leq E\left[\max\left\{v_i + Y, v_i - Y, \max_{j \neq i}\{v_j\}\right\}\right] + N\kappa \\
&\leq E\left[\max\left\{v_i + |Y|, \max_{j \neq i}\{v_j\}\right\}\right] + N\kappa \\
&\leq \max_j\{v_j\} + (N+1)\kappa \qquad \text{since E[|Y|]} = \kappa
\end{aligned}$$

■

Lemma 2.3 *If inequality (1) holds for $\tau = N$, then for any i:*

$$V_{M_i}(\mathbf{L}, N+1) \leq \max_j\{v_j\} + (N+1)\kappa \tag{3}$$

Proof: We prove this by induction on H, the number of search units required to completely explore the search information. The lemma holds if $H = 0$, since $V_{M_i}(\mathbf{L}, N+1) = \max_j\{v_j\}$. So, for the remaining cases where $H > 0$,

for any i:

$$\begin{aligned} V_{M_i}(\mathbf{L}, N+1) &= E[V(L_{(i,1)}, N+1)] \\ &= E[V((h_{(i,1)}, v_{(i,1)}), N+1)] \end{aligned}$$

We observe that the search information, $\mathbf{L}$, only contains a single leaf and so the node that decides the terminal payoff must be a direct descendant of the node $L_{(i,1)} = (h_{(i,1)}, v_{(i,1)})$. The additive structure of the tree therefore implies that $V(\mathbf{L})$ is linear in $v_{(i,1)}$. Hence:

$$\begin{aligned} V_{M_i}(\mathbf{L}, N+1) &= V((h_{(i,1)}, E[v_{(i,1)}]), N+1) \\ &= V((h_i - 1, v_i), N+1) \\ &= \max\{V_{M_1}((h_i - 1, v_i), N+1), V_{S_1}((h_i - 1, v_i), N+1)\} \\ &\leq \max\{V_{M_1}((h_i - 1, v_i), N+1), v_i + (N+1)\kappa\} \text{ from (2)} \end{aligned}$$

The result is proved with the observation that since the search information $(h_i - 1, v_i)$ has a lower H value than $\mathbf{L}$, equation (3) implies by induction on H:

$$V_{M_i}((h_i - 1, v_i), N+1) \leq v_i + (N+1)\kappa$$

■

Lemma 2.4

$$\forall \tau : \quad V(\mathbf{L}, \tau) \leq \max_j\{v_j\} + \tau\kappa$$

Proof: Lemmas 2.2 and 2.3 show that if (1) is true for $\tau = N$, then (2) and (3) also hold, so

$$\max\left\{\max_i\{V_{M_i}(\mathbf{L}, N+1)\}, \max_i\{V_{S_i}(\mathbf{L}, N+1)\}\right\} \leq \max_j\{v_j\} + (N+1)\kappa$$

The optimality equation for V is:

$$V(\mathbf{L}, N+1) = \max\left\{\max_i\{V_{M_i}(\mathbf{L}, N+1)\}, \max_i\{V_{S_i}(\mathbf{L}, N+1)\}\right\}$$

Combining these we see if that inequality (1) is true for $\tau = N$ then this implies that $V(\mathbf{L}, N+1) \leq \max_j\{v_j\} + (N+1)\kappa$, which is inequality (1) for $\tau = N+1$ so the result is proved by induction on τ. ■

Theorem 2.5 *For $\tau \leq h$, the optimal payoff is $\tau\kappa$. Policy π^*, defined below, achieves this payoff and so is therefore optimal.*

Policy π^* is to pick a leaf L_i which satisfies $v_i = \max_{1\leq j\leq N}\{v_j\}$ and
if $\tau > 0$ then *Search* it,
if $\tau = 0$ and $h_i > 0$ then *Move* to a random daughter of it,
if $\tau = 0$ and $h_i = 0$ then move to it.

Proof: The payoff from playing policy π^*, V_{π^*}, from a state where $\min_i\{h_i\} \geq \tau$ is given by:

$$V_{\pi^*}(\mathbf{L}, \tau) = \max_i\{v_i\} + \tau\kappa$$

The value of the game played under the optimum policy, V, is an upper bound on the value of the game played under any other policy, so combining this with Lemma 2.4, we see:

$$V_{\pi^*}(\mathbf{L}, \tau) \leq V(\mathbf{L}, \tau) \leq \max_i\{v_i\} + \tau\kappa = V_{\pi^*}(\mathbf{L}, \tau)$$

■

2.2.2 Optimal Policy for $\tau > h$

The optimal policy for $\tau > h$ depends upon the properties of the distribution $P()$. At one extreme, if Y takes only the values -1 and 1, the paths of the game tree represent all possible courses of a 1-dimensional random walk of length h. In this case, the set of terminal node values is completely determined, and $\tau = h$ is sufficient time to discover the path to the best one, so any extra search time is superfluous.

Conversely, for any $P()$ with support along the whole real line, all 2^h terminal nodes must be searched before it can be definitely established which one of them has the greatest score, so search time only becomes superfluous for $\tau \geq 2^h - 1$. We conclude our study of this model with a short theoretical section in which we shall prove a restriction on the optimal policy for $\tau > h$ and outline how it may be tackled as an optimal stopping problem.

Recall our earlier definition of S, as the set of policies which only ever make a move down the game tree if it is part of a sequence of moves to an unsearched node or to the end of the tree. Define $S^* \in S$ to be set of such policies which never move with $\tau > 0$.

Lemma 2.6 *There is an optimal policy in S^*.*

Proof: We show that for every policy $\pi' \in S \backslash S^*$, there is a policy $\pi^* \in S^*$ which obtains the same payoff. Once again we use induction on H, the number of time units required to search all of the search information. The result is trivially true for states in which H=0, so we now consider cases in

which $H > 0$.

Firstly, let π^* mimic π' until the first state in which π' suggests a move action with $\tau > 0$. Since $\pi' \in S$, such an action must be the first of a consecutive sequence which leads either to the end of the game or to an unsearched node. The first of these cases is trivial; π^* can achieve an identical payoff by taking the same sequence of moves once it has exhausted the search time by making random searches until $\tau = 0$.

The remaining case is that in which $\tau > 0$ and π' moves to an unsearched node, $L_{(i,1)}$, achieving the following payoff:

$$\begin{aligned} V_{M_i\pi'}(\mathbf{L}, \tau) &= E[V_{\pi'}(L_{(i,1)}, \tau)] \\ &= E[V_{\pi'}((h_{(i,1)}, v_{(i,1)}), \tau)] \\ &= V_{\pi'}((h_i - 1, E[v_{(i,1)}]), \tau) \\ &= V_{\pi'}((h_i - 1, v_i), \tau) \end{aligned} \tag{4}$$

Since $L_{(i,1)}$ has a lower H value than L_i, the induction hypothesis tells us that the below is true for some $\pi'^* \in S^*$:

$$V_{\pi}((h_i - 1, v_i), \tau) \leq V_{\pi'^*}((h_i - 1, v_i), \tau) \tag{5}$$

Policy π^* omits the move action of π', but ignores all nodes of $\mathbf{L}$ except L_i.

$$V_{\pi}(\mathbf{L}, \tau) = V_{\pi}((h_i, v_i), \tau) \tag{6}$$

Policy π^* then carries out the same sequence of actions from $((h_i, v_i), \tau)$ as policy π'^* does from $((h_i - 1, v_i), \tau)$. If concludes by making a random move

to reach the end of the game, leaving the expected payoffs of the two policies equal.

$$V_{\pi'^*}((h_i - 1, v_i), \tau) = V_{\pi^*}((h_i, v_i), \tau) \tag{7}$$

Policy π^* carries out a sequence of searches, mimics $\pi'^* \in S^*$ and concludes with a move to an unsearched node, and is therefore also in S^*. From (4), (5), (6) and (7):

$$\begin{aligned} V_{M_i\pi'}(\mathbf{L}, \tau) &= V_{\pi'}((h_i - 1, v_i), \tau) \\ &\leq V_{\pi'^*}((h_i - 1, v_i), \tau) \\ &= V_{\pi^*}((h_i - 1, v_i), \tau) \\ &= V_{\pi^*}(\mathbf{L}, \tau) \end{aligned}$$

■

We now suggest the following conjecture as a step towards proof of the optimal policy for $\tau > h$.

Conjecture 2.1 *For $\tau \geq h$, there is an optimal policy in S which is certain to reach a state in which for some node L_i, $v_i = max_j\{v_j\}$ and $\tau = h_i$ without moving to an unsearched node.*

If this is proved true, then the following result is also proved.

Corollary 2.7 *For $\tau \geq h$, there is an optimal policy, $\pi^* \in S^*$, that is, one which never moves to an unsearched node.*

Proof: From Conjecture 2.1, there is an optimal policy in S which is certain to reach a state in which for some node L_i, $v_i = max_j\{v_j\}$ and $\tau = h_i$, with moving to an unsearched node Theorem 2.5 establishes the following upper bound on the optimal payoff, obtainable from such a point:

$$V(\mathbf{L}, \tau) \leq max_j\{v_j\} + \tau\kappa$$

This applies to states in which for all i, $h_i > \tau$. This case is more restricted, but the bound can be achieved by discarding all nodes other than L_i and playing optimally as described in Theorem 2.5. Since $\tau = h_i$, such a policy searches to the very end of the tree. ■

Corollary 2.7 tells us that there is an optimal policy which never moves to an unsearched node for $\tau = h + 1$. This therefore defines the shapes of the search information which it may be optimal to search; since it must include an unbroken path of search from root down to a leaf, one unit of search remains, which — assuming $P()$ is such that it is needed — will form a bifurcation at some point down this strand.

Once this extra unit of time has been spent, the problem is reduced to the '$\tau = h$' case, and optimal policy is established. The problem may therefore be treated as one of optimal stopping — the 'stopping' being interpreted as using up the extra unit of time to create a bifurcation of the search information. The practical upshot of this is that the optimal policy with $h + 1$ units of time, as shown overleaf, is to search down a single variation, always picking the better daughter, until the Y value is sufficiently small. When this occurs, it is optimal to 'backtrack' up the variation one level and search the sister

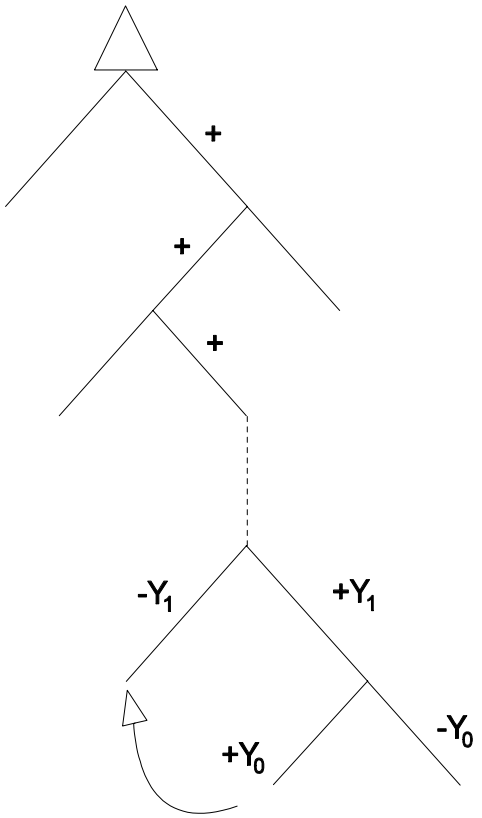


Figure 8: Optimal Policy for $\tau = h + 1$

of the last node just searched. The threshold value of Y below which Y is deemed to be 'sufficiently small' is a function of the last two Y values found. It also decreases as does the height of the tree remaining. If the bottom of the tree is reached without the extra time unit having been used, it is optimal to use it whatever the last two values of Y, since there is no further use for it.

2.3 A Fuel Control Problem

We now consider a fuel control problem inspired by the tree search game of the previous section. As before, the game consists of moving to the bottom of a finite tree, possibly with the aid of one or more searches. Similarly, the reward from playing the game is the value of the leaf arrived at.

In contrast, however, to the original problem, we assume that the set of leaf values is known. Moreover the value of each separate leaf node, i, is $f(i)$, which is known. The uncertainty in this game concerns the *moves*. The player at root has perfect information of the game tree ahead of him, but cannot distinguish which of the two moves available will lead to which half of the game tree. The action 'move' causes the player to choose one random half of the game tree and move there. The 'search' option reveals this information to him. This is equated to the expenditure of fuel (or search time).

Faced with a height h game tree, h time units are sufficient to achieve the optimal result. The following policy is optimal:

1. Select a leaf node, L, which achieves the greatest reward.
2. Carry out a search to discern which move is which, and move to the half of the tree which contains node L. Go to 2.

The optimal policy for less than h time units is more interesting. It is useful to introduce some notation at this point. We shall define two sequences of functions on the game tree, $V_n()$ and $V_n^*()$. Each function accords values

to each node of the tree. We denote by p_{ij} the probability of arriving at j if a move is made at random from i, so $p_{i(i,1)} = p_{i(i,2)} = \frac{1}{2}$.

Define $V_0()$ as follows:

$$V_0(i) = \begin{cases} f(i) & \mid \ i \text{ is a leaf} \\ \sum_j p_{ij} V_0(j) & \mid \ \text{otherwise} \end{cases}$$

That is to say, $V_0()$ is the harmonic extension of the $f()$ values at the leaves.

Define $V_0^*()$ with reference to $V_0()$ as follows:

$$V_0^*(i) = \begin{cases} f(i) & \mid \ i \text{ is a leaf} \\ \max_{j|p_{ij}>0} \{V_0(j)\} & \mid \ \text{otherwise} \end{cases}$$

Since $V_0()$ is the harmonic extension of $f()$, $V_0^*()$ is a majorant of $V_0()$. It is equal to the payoff from a policy of expending one unit of time straight away, and then making random moves thereafter. We now define the following function $V_1()$ with respect to $V_0^*()$:

$$V_1(i) = \begin{cases} f(i) & \mid \ i \text{ is a leaf} \\ \max \left\{ V_0^*(i), \sum_j p_{ij} V_1(j) \right\} & \mid \ \text{otherwise} \end{cases}$$

Theorem 2.8 shows that this function gives the value of the game when one unit of time remains and the optimal policy is followed. An example of these two functions is given overleaf in Figure 9.

The $V_1^*()$ tree is now defined with respect to $V_1()$ in a similar fashion to the way in which $V_0^*()$ was defined in terms of $V_0()$. This process is repeated, allowing computation of $V_\tau()$ for any τ.

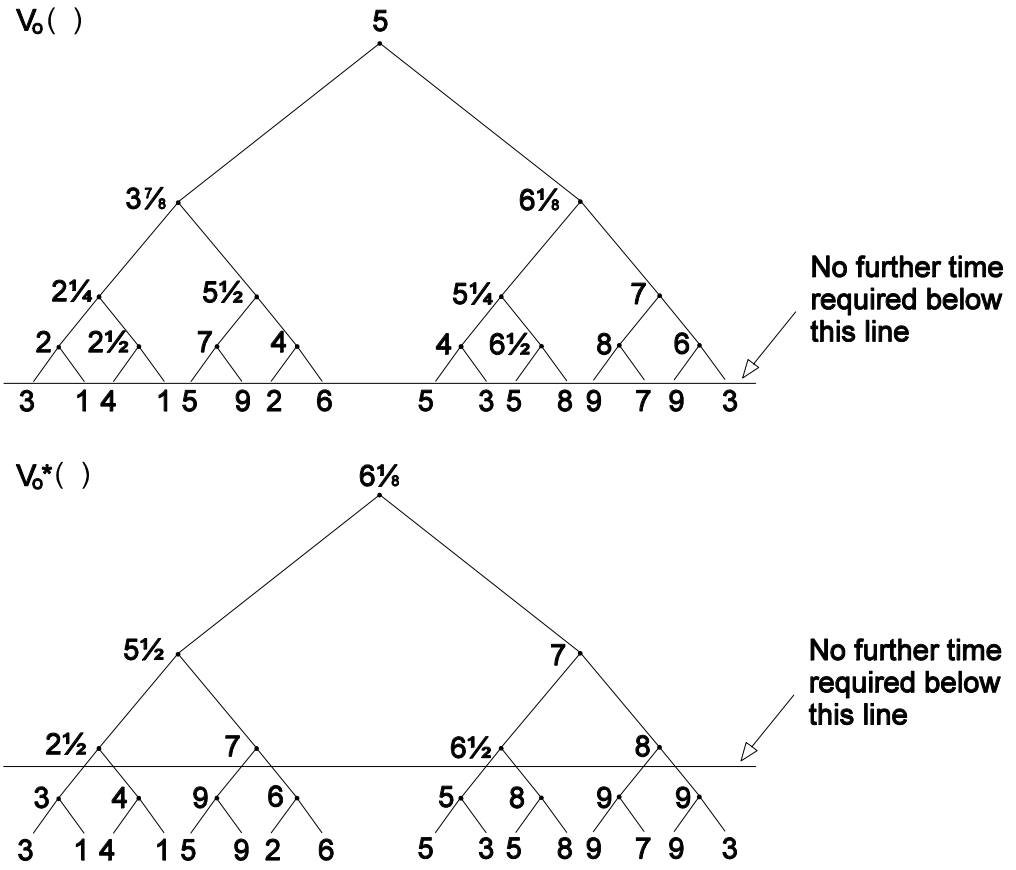


Figure 9: Example $V_0()$ and $V_0^*()$ Values

Theorem 2.8 *The payoff from playing optimally with τ units of time is given by $V_{\tau+1}()$.*

Proof: If $\tau = 0$, the only available policy is to move at random, so the result is true since $V_0()$ is the harmonic extension of $f()$. We now apply induction, assuming that it is true for $\tau = n$. The optimality equation for $V_{n+1}()$ is as

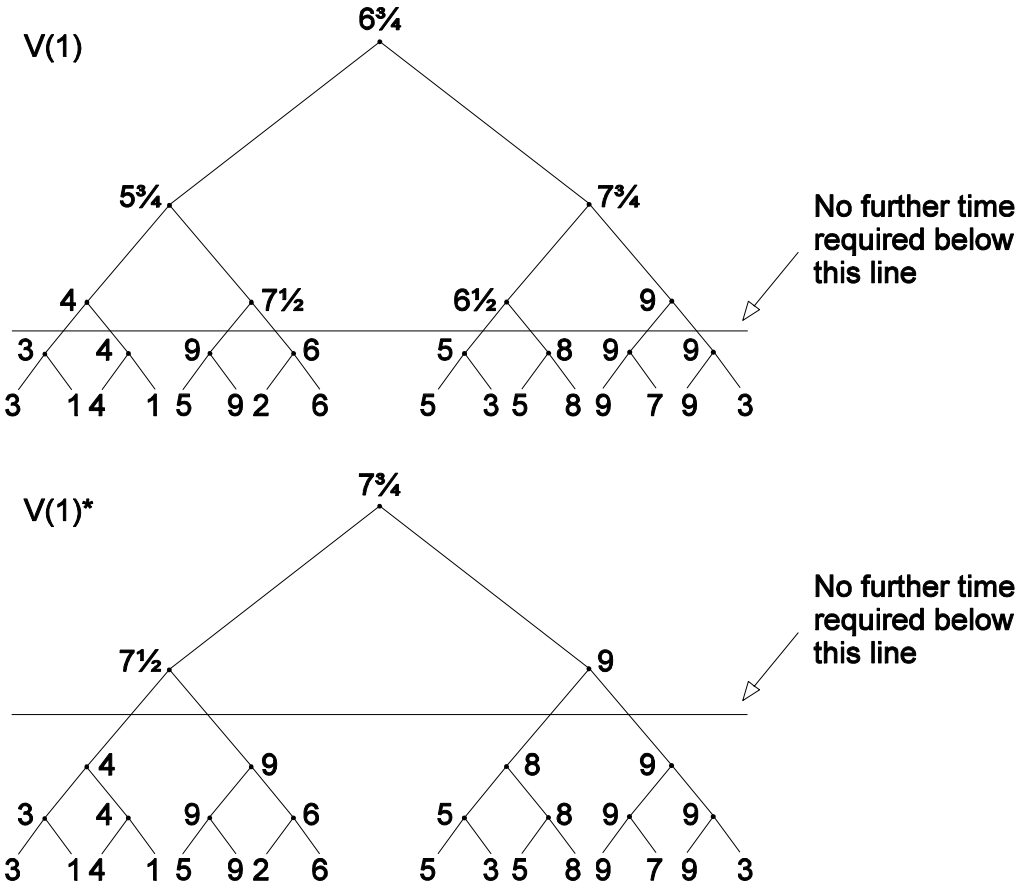


Figure 10: Example $V_1()$ and $V_1^*()$ Values

follows:

$$V_{n+1}(i) = \max\left\{V_n^*(i), \sum_j p_{ij} V_{n+1}(j)\right\}$$

This shows it is the minimal superharmonic majorant of $V_n^*()$. Hence it is the optimality equation for optimally stopping V_n^*. ■

Corollary 2.9 *The function $V_\tau^*()$ gives the optimal payoff if $\tau + 1$ units of fuel are available, and one of them has to be expended immediately.*

Proof: If i is a leaf, then only one payoff is possible, so the corollary is trivially true. Otherwise:

$$V_\tau^*(i) = \max_{j|p_{ij}>0} \{V_\tau(j)\}$$

■

For $n \geq h$, the process is trivial since $V_n() = V_{n+1}()$, reflecting the fact that no more than h search units are required to ensure an optimal choice of moves. The problem is now solved, since the optimal action from a node with τ time units available for search can be deduced from consideration of the $V_\tau()$ values of that node and its children. The remaining $V_n()$ and $V_n^*()$ functions for the example are shown in Figures 10 and 11.

Having understood this simple model, we now consider several generalisations. Firstly, the game tree need not be binary, indeed it need not have a constant branching factor. Furthermore, it need not have a fixed height, since any finite irregular tree of maximum height h, maximum branching factor b can be modelled as a regular tree of height h, branching factor b, simply by the addition of 'dummy' branches where each daughter has the same score as the parent node.

What is more, the structure of the game need not be a tree; a DAG can be treated in an identical fashion. This is perhaps most easily seen by observing that any DAG may be expanded to an equivalent out-tree, by simply making copies of the nodes which have in-degree of greater than one, as shown below. We note that in practice it is not even necessary to perform a transformation.

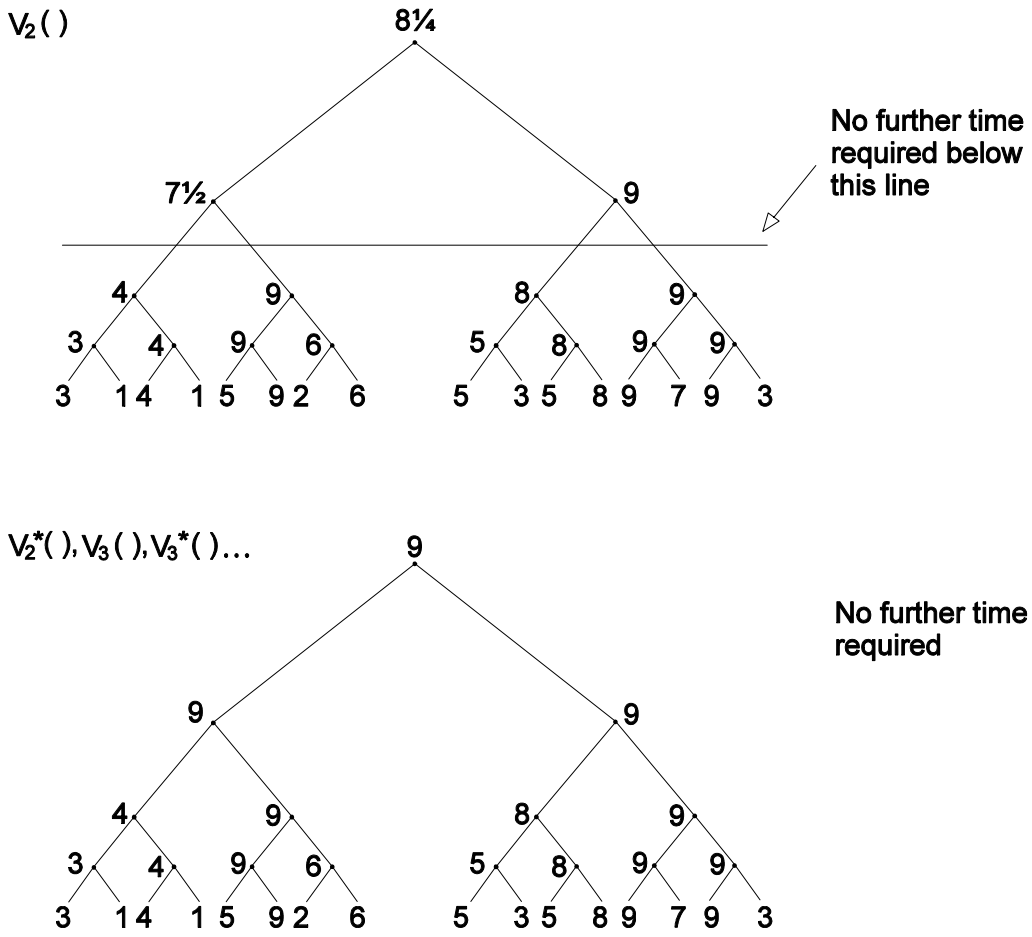


Figure 11: Example $V_2()$ and $V_2^*()$ Values

We have now seen how the method developed for the binary tree case may be applied without difficulty to any finite DAG. We now broaden the scope of the model still further by re-interpreting the nature of the uncertainty involved in 'moving' in a more general fashion. Let us replace the notion of spending a unit of time to 'discover which move is which' by that of 'expending a unit of fuel in order to control our movement'. This allows specification of a general probability distribution over a node's possible daughters. The

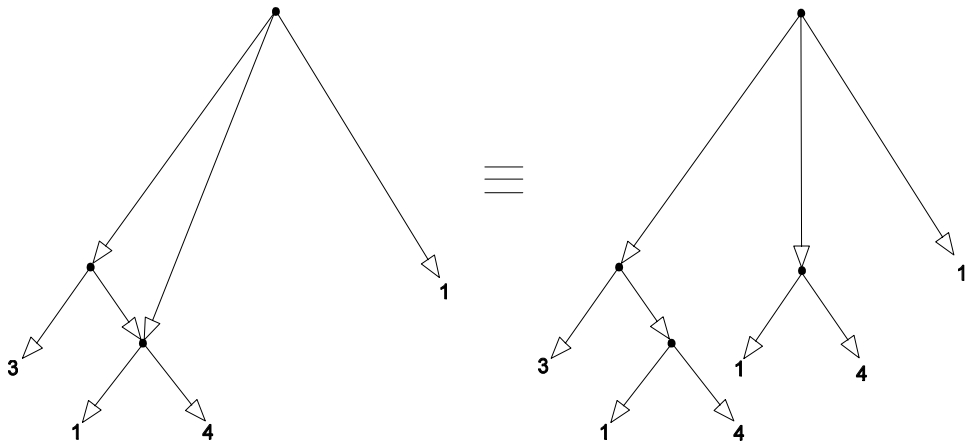


Figure 12: Expansion of a DAG to an Equivalent Out-tree

previous model, which assumed that random movement to each of a node's daughters was equally likely, is thus reduced to a special case.

We are now in a position to understand the problem description as a fuel allocation problem. The problem is defined on a DAG, with one source node, at which some 'particle' begins, and one or more sink nodes, at which it can end the game. Each sink node, i, is associated with a finite payoff, $f(i)$, the value of the game if the particle reaches that node. Every arc is associated with a positive weight. We naturally require that the sum of weights associated with the outarcs of a node be 1, unless the node is a sink, in which case it has no outarcs, by definition.

If no fuel is expended, the particle makes a random transition, following an outarc of the node where it currently resides with a probability equal to the associated weight. If the player chooses to expend a unit of fuel he may select any of the outarcs from the particle's current position and move the particle along it.

If enough time units are available, it is possible to control every transition made by the particle, directing it to reach a sink with the greatest reward, which is of course optimal. This will be reflected in the calculation of the $V_\tau()$ value of root.

We now consider one final generalisation — that of allowing cyclical graphs. This causes no problem for the theoretical definition of the $V()$ and $V^*()$ values. In practice, they become harder to calculate, since the straightforward 'bottom up' recursive calculation method is no longer possible if the graph contains possible cyclical particle trajectories, owing to the mutual dependence of the $V()$ values of the nodes in the cycle.

In this case we resort to application of the policy improvement algorithm. We calculate $V_0()$, the harmonic extension of $f()$ as before. To calculate $V_0^*()$, the policy improvement algorithm starts with a policy which everywhere expends a unit of fuel and then proceeds at random. It then proceeds by iteratively decreasing E, the set of points at which it is optimal to expend fuel. The policy of expending the fuel unit at a node in set E has a payoff $V_0^E()$, defined as follows:

$$V_0^E(i) = \begin{cases} f(i) & \mid \; i \text{ is a sink node} \\ \max\limits_{j \mid p_{ij} > 0} \{V_0(j)\} & \mid \; i \in E \\ \sum\limits_j p_{ij} V_0^E(j) & \mid \; \text{otherwise} \end{cases}$$

The steps of the policy improvement algorithm are:

1. Let E contain all the non-sink nodes of the graph.
2. Delete from E any nodes i such that:

$$\sum_j p_{ij} V_0^E(j) > \max_{j|p_{ij}>0} \{V_0(j)\}$$

   (If there are no such nodes, go to step 4.)
3. Recalculate the $V_0^E()$ values. The fact that the new $V_0^E()$ function is a strict majorant of the old one is a consequence of the definition of $V_0^E()$ and the choice of points to omit from E. Go to step 2.
4. Since this choice of E is optimal, $V_1() \equiv V_0^E()$. Deduce the set A_1 of nodes with $V_1(i) = max_j\{f(j)\}$. This set contains points from which the optimal payoff is attainable without further time units.

A similar method can be used to deduce $V_{N+1}()$ from $V_N()$, for any N. Examination of step 2 shows that termination occurs after a number of iterations not exceeding the number of nodes. Note the criterion for removing a point from E:

$$\sum_j p_{ij} V_0^E(j) > \max_{j|p_{ij}>0} \{V_0(j)\}$$

Since the $V_N^E()$ values are never adjusted downwards, it is never optimal to add a node back into E once it has been removed. This, together with

the criterion for reaching step 4, prove the assertion that the choice of E is optimal at this point. Finally, we note that once a node is added to A_i it need not be included in subsequent iterations. Iterations are therefore likely to involve less and less calculation as time proceeds, since more and more nodes have been added to A_i.

We conclude our discussion of this problem with a special case — grids which have a translationally invariant transition structure. Define the *interior* of a translationally independent lattice to be the set of points which are not sink nodes, and which have a transition structure identical to each of their neighbours. i.e. those more than one step away from any sink nodes, edges or other irregularities in the lattice.

Theorem 2.10 *It is optimal never to expend the last remaining unit of fuel from a state in the interior of a translationally independent lattice.*

Proof:

Let node x be a node on the interior of a translationally invariant lattice, with neighbours denoted $x+1, \ldots x+n$. Suppose that if no fuel is expended, cell $x+i$ is reached with probability p_i. Note that because x is in the interior of the lattice, $(x+i)+j = (x+j)+i$, since making a move in direction i followed by one direction j is equivalent to making these moves in the reverse order.

$$V_{N+1}(x) = \max \left\{ \max_i \{V_N(x+i)\}, \sum_{j=1}^{n} p_j V_{N+1}(x+j) \right\}$$

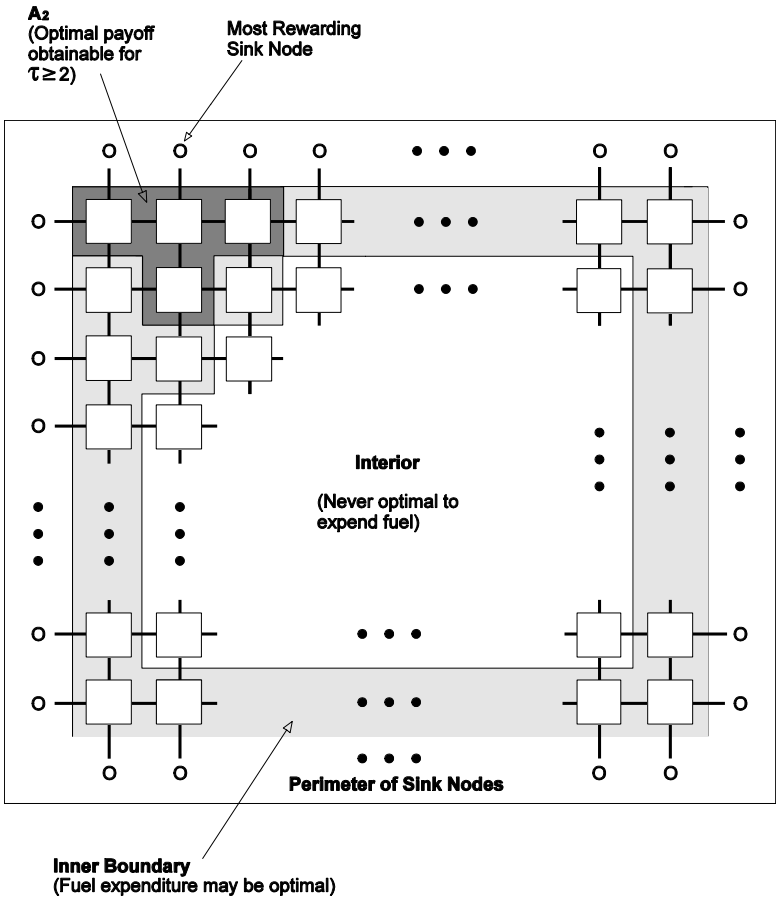


Figure 13: Calculation of $V_3()$ on a Translationally Invariant Grid

Consider the optimality equation above. We show that the second of the terms of the maximisation dominates the first.

$$\begin{aligned}
\sum_{j=1}^{n} p_j V_{N+1}(x+j) &\geq \sum_{j=1}^{n} p_j V_N((x+j)+i) && \forall i \\
&\geq \sum_{j=1}^{n} p_j V_N((x+i)+j) && \forall i \\
&\geq \max_i \left\{ \sum_{j=1}^{n} p_j V_N((x+i)+j) \right\} \\
&\geq \max_i \{V_N(x+i)\}
\end{aligned}$$

Specifically, if $N = 0$, we observe:

$$\sum_{j=1}^{n} p_j V_1(x+j) \geq \max_i \{V_0(x+i)\}$$

■

This result speeds up application of the policy improvement algorithm for cases in which some subset of the graph is a translationally invariant lattice, as is illustrated in above. It can be understood in familiar terms. Not expending fuel equates to making a random move, while expending fuel entails making a decision about how to do so. For positions in which the nearest sink or edge is more than one transition away, a random move causes a reversible change to the node's position, since if desired, a time unit may be expended to move back. Therefore, for every policy which first expends fuel and then moves randomly there is an equivalent policy which interchanges these actions. Not only does this entail no loss, it is better in general, since more information is available (the direction of one more random transition) upon which to base the decision about how to move.

2.4 Summary

In the general case of the game-playing problem as we have discussed it, strictly 'optimal' play is, even in theory, an unachievable concept, due to the complications in trying to anticipate the opponent's policy. We must therefore study less complicated models if a strictly optimal policy is sought. One such is the one-player game of Section 2.2. The optimal policy for the case of $\tau \leq h$ is fairly trivial, but considerable work had to be put into its proof. Similarly, the results for $\tau > h$ represent the fruits of some considerable effort, even though they stop short of actually specifying an optimal policy, and are probably of theoretical rather than practical interest.

The fuel control model of Section 2.3 was deduced as something of a 'spin-off' of investigation into the main one-player tree search model. It has a pleasing generality, in that it goes so far beyond the binary tree case of its parent model. I therefore believe that it may well turn out to be of practical use without further modification, and its serendipitous discovery is further evidence of the value of developing tree search models.

3 OR-Tree Search

This chapter addresses a problem which we shall refer to as 'OR-tree' search[14]. The model that we address can be viewed as an investigation into the truth of a logical expression, L, which is a finite conjunction of logical primitives, l_i. L is a logical expression iff one of the following is true:

1. L is a logical primitive.
2. $L \equiv (X \cup Y)$, where X and Y are both logical expressions.

The symbol '$\cup$' represents the customary binary Boolean 'OR' operator, which is associative, and so the above definition implies the following:

$$L \equiv l_1 \cup l_2 \cup \ldots l_N$$

For proofs, different types of logical primitive will be differentiated by numerical subscripts. However, when giving examples, we shall refer to logical primitives by capital letters, A, B, $C \ldots$ so that a numerical subscript can be added to distinguish between logical primitives of the same type.

Sometimes it will be useful to have a label for logical expressions other than primitives. e. g. $Y \equiv (A \cup B)$. These we shall refer to by using uppercase letters, starting from X. Again, independent copies will be distinguished by use of numerical subscripts.

[14]The reader interested in rule-based methods of theorem proving should note that this chapter — indeed this thesis — does not contain any material of *direct* relevance.

3.1 Deterministic Case

We begin our consideration of this problem by restricting our attention to the case in which every logical primitive is a *box*. By *box* we mean a logical primitive which, when searched, will be shown for certain to be either true or false. The time taken to search a box may be a random variable, but we require that the expected time required to search it, t, is known. The probability, p, that it contains an object is also known.

We take the term 'box' from a classic and well-discussed case — see Dean [22], Joyce [35], Mitten [52] and Sweat [83] amongst others — which involves the search of a set of N boxes. In terms of the original model, p_i is the probability that box i contains an object, which is found if search is carried out on that box. We can view L, therefore, as the statement "one or more of the boxes $l_1 \ldots l_N$ contains an object". We seek to deduce a search policy which can determine, for sure, the truth of a general logical expression, L, in the smallest possible expected time.

This search is termed *satisficing*, since it aims to find a solution which is *good enough* for some specified purpose. It is to be contrasted with an optimising search which aims to find the *best* solution available. In contrast to an optimisation, satisficing search does not necessarily have a solution – in this case search proceeds until all the possible solutions have been shown not to meet the constraints which define what is 'good enough', whereupon the search terminates unsuccessfully. If, however, a solution is found, search may be terminated immediately.

We shall assume for simplicity that a search of a box which contains an object is certain to be successful, that is, an object will be found, although this restriction will be relaxed in Section 3.7.

Search continues until an object is detected, or until all the boxes have been unsuccessfully searched. A policy is termed *optimal* if it searches boxes in such a way that it minimises the payoff, V, equal to the expected time until termination.

In the model described above, each policy can be identified with what Kadane and Simon [37, 77] term a *strategy*, that is, a permutation of the integers from 1 to N. A strategy, A, represents the policy which searches the boxes in the order of the integers given by A, ceasing as soon as an object is detected or if all the boxes have been searched.

Consider an arbitrary strategy, $A = \{a_1, a_2 \dots\ a_n\}$. Using $\mathbf{x}$ to represent the state, we denote by $V_A(\mathbf{x})$ the expected termination time of applying strategy A to search the boxes. We shall use q_i to denote the probability that an object is not found when box i is searched, and $p_i = 1 - q_i$.

Hence:

$$V_A(\mathbf{x}) = \sum_{i=1}^{N} t_{a_i} \prod_{j=1}^{i-1} q_{a_j}$$

We now modify A by interchanging the k^{th} and $k + 1^{th}$ elements to obtain $A' = \{a_1, a_2 \dots a_{k-1}, a_{k+1}, a_k, a_{k+2}, \dots\ a_n\}$, and consider the payoff from applying this modified strategy:

$$V_{A'}(\mathbf{x}) = \sum_{i=1}^{k-1} t_{a_i} \prod_{j=1}^{i-1} q_{a_j} + t_{a_{k+1}} \prod_{j=1}^{k-1} q_{a_j} + q_{a_{k+1}} t_{a_k} \prod_{j=1}^{k-1} q_{a_j} + \sum_{i=k+2}^{N} t_{a_i} \prod_{j=1}^{i-1} q_{a_j}$$

It is instructive to consider the expected difference in the time taken by policies A and A':

$$\begin{aligned} V_A(\mathbf{x}) - V_{A'}(\mathbf{x}) &= \sum_{i=k}^{k+1} t_{a_i} \prod_{j=1}^{i-1} q_{a_j} - t_{a_{k+1}} \prod_{j=1}^{k-1} q_{a_j} - q_{a_{k+1}} t_{a_k} \prod_{j=1}^{k-1} q_{a_j} \\ &= \prod_{j=1}^{k-1} q_{a_j} (t_{a_k} + (1 - p_{a_k}) t_{a_{k+1}} - t_{a_{k+1}} - t_{a_k}(1 - p_{a_{k+1}})) \\ &= \prod_{j=1}^{k-1} q_{a_j} (p_{a_{k+1}} t_{a_k} - p_{a_k} t_{a_{k+1}}) \\ &> 0 \qquad \Leftrightarrow \frac{p_{a_{k+1}}}{t_{a_{k+1}}} > \frac{p_{a_k}}{t_{a_k}} \end{aligned}$$

This simple interchange argument shows that strategy A' is an improvement upon strategy A if the exchanged boxes, a_k and a_{k+1}, were not in decreasing order of $\frac{p}{t}$. If we define the *reward rate*, $\emptyset_i = \frac{p_i}{t_i}$, of a box i, we see that the optimal policy must therefore be to search the boxes in decreasing order of $\emptyset$.

3.1.1 Linear Precedence Constraints

We now suppose that some restrictions exist as to the order in which the boxes can be searched. Specifically, we consider the case of linear precedence constraints, as illustrated below.

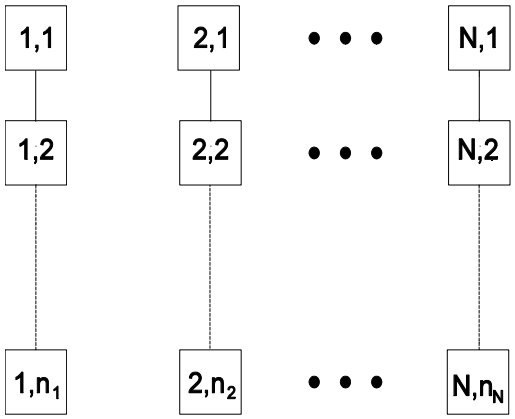


Figure 14: Linear Precedence Constraints

Such a constraint structure arises if we assume that the boxes are arranged in N stacks, so that box (i, j) is the j^{th} box in stack i, and the searcher is limited to searching only the uppermost unsearched box of each stack. The goal of search, to discover whether there is an object, remains unchanged.

There are now two factors that influence the choice of box; not only the immediate possibility of finding an object, but also the future benefit from access to boxes below must be considered. This problem has a straightforward solution, first shown by Mitten [52].

Let us begin by considering briefly the case in which the boxes are arranged so that the $\emptyset_i$ decreases from the top of each stack to the bottom.

In this case, it is possible to select the boxes in decreasing order of $\emptyset_i$, and so the same payoff is possible. This is, therefore, optimal by the trivial observation that restricting the possible actions cannot possibly improve the payoff. We now show that it is possible for any boxes problem with linear precedence constraints to find an equivalent one with this structure.

Suppose $\emptyset_{(i,j)} < \emptyset_{(i,j+1)}$. We show that this implies that boxes (i,j) and $(i,j+1)$ belong to the same *indivisible block*. By this we mean that it cannot be optimal to immediately follow search of (i,j) by search of any box other than $(i,j+1)$. To see this, we compare the payoffs of the following three policies:

π_1: Search some other box(es), k, then box (i,j), then box $(i,j+1)$.

π_2: Search box (i,j), then some other box(es), k, then box $(i,j+1)$.

π_3: Search box (i,j), then box $(i,j+1)$, then some other box(es), k.

(It is understood that the policies only search if necessary — i.e. that they terminate if an object is found). Let us denote by p_k the overall probability that the other box(es) contain an object and by t_k the expected time to search them.

$$\begin{aligned} V_{\pi_1} &= t_k + q_k t_{(i,j)} + q_k q_{(i,j)} t_{(i,j+1)} \\ V_{\pi_2} &= t_{(i,j)} + q_{(i,j)} t_k + q_k q_{(i,j)} t_{(i,j+1)} \\ V_{\pi_3} &= t_{(i,j)} + q_{(i,j)} t_{(i,j+1)} + q_{(i,j)} q_{(i,j+1)} t_k \end{aligned}$$

Hence:

$$\begin{aligned} V_{\pi_2} - V_{\pi_1} &= t_{(i,j)} + q_{(i,j)}t_k - t_k - q_k t_{(i,j)} \\ &= p_k t_{(i,j)} - p_{(i,j)} t_k \\ &= t_{(i,j)} t_k (\emptyset_k - \emptyset_{(i,j)}) \qquad (8) \\ V_{\pi_2} - V_{\pi_3} &= q_{(i,j)}t_k + q_k q_{(i,j)} t_{(i,j+1)} - q_{(i,j)} t_{(i,j+1)} - q_{(i,j)} q_{(i,j+1)} t_k \\ &= q_{(i,j)} (p_{(i,j+1)} t_k - p_k t_{(i,j+1)}) \\ &= q_{(i,j)} t_k t_{(i,j+1)} (\emptyset_{(i,j+1)} - \emptyset_k) \qquad (9) \end{aligned}$$

Equation (8) implies that if $\emptyset_k > \emptyset_{(i,j)}$ then policy π_1 achieves a lower payoff than policy π_2, while equation (9) implies that if $\emptyset_k < \emptyset_{(i,j+1)}$ then policy π_3 achieves a lower payoff than policy π_2. At least one of these conditions applies, since $\emptyset_{(i,j+1)} > \emptyset_{(i,j)}$, and so policy π_2 is therefore not optimal.

The solution then proceeds by processing the boxes, starting from the bottom of each stack, grouping them into indivisible blocks wherever possible. This process of grouping together separate searches into a single unit we shall refer to as *chunking*. Once no more chunking can be carried out, the indivisible blocks are said to be *maximal*. In this case, it is a consequence of the criterion for chunking that the maximal indivisible blocks within a stack must now be in order of decreasing $\emptyset$, and so the optimum policy is just to search them in decreasing order of $\emptyset$.

3.2 Stochastic Case

We now increase the scope of the model to cater for a more general class of logical primitives, to allow dynamic revelation of nodes (previously referred to as boxes) in the course of search. We assume that all nodes belong to one of n different types, the details of which are known.

Satisficing search on an out-tree differs from more familiar tree search models, such as shortest path search, in that the structure of nodes already searched is of no importance in guiding further search. Apart from the set of nodes now available for search, the only relevant result of previous search is whether or not an object has been found. The state of an ongoing search problem may therefore be represented by $\mathbf{x}$, a vector of length n that contains the number of nodes of each type that are currently visible.

Subsection 3.3.2 compares the values of different problems and so there the state of a search will be represented $(\mathbf{x}, \mathbf{d})$ where $\mathbf{d}$ is a vector of length n which summarises the details of each type of node.

$$\mathbf{d} = (d_1, d_2 \ \ldots \ d_n) \text{ where } d_i = (p_i, t_i, f_i)$$

The definitions of p_i and t_i are unchanged, whilst we use $f_i(s)$ to denote the *offspring distribution* of node i. This is defined as the n-dimensional probability distribution of extra nodes revealed when a type i node is searched.

We shall use familiar notation, $V_\pi(\mathbf{x}, \mathbf{d})$, to refer to the expected time taken to terminate starting from state $\mathbf{x}$ with nodes of type $\mathbf{d}$, when policy π is applied. In cases where $\mathbf{d}$ is fixed, we shall abbreviate this as $V_\pi(\mathbf{x})$. The

optimal value, $V(\mathbf{x})$ is given by:

$$V(\mathbf{x}) \;=\; min_{\pi}\{V_{\pi}(\mathbf{x})\}$$

A (non-randomised) policy π is a (deterministic) function of a state $\mathbf{x_i}^{\pi}$, with past history, $\mathbf{H_i} = (\mathbf{x_0}^{\pi}, a_1, \mathbf{x_1}^{\pi}, a_2 \;\ldots \mathbf{x_i}^{\pi})$, where $a_i \in \{1 \;\ldots\; n\}$. Denote the $\tau + 1^{\text{th}}$ action taken by π, as $\pi_{\tau}(\mathbf{H}_{\tau})$. A Markov policy is a policy which does not take the past history of states or actions into account, so can be expressed $\pi(\mathbf{x_i}^{\pi}) = a_i$. Since this is a one player game, there is an optimal policy amongst the class of non-randomised Markov policies. We shall therefore restrict our attention for the rest of this paper to these policies, so any policy mentioned may be assumed to be both non-randomised and Markov.

We shall expand the state space by adding the special state, $\langle \mathrm{T} \rangle$, which corresponds to having found an object. This must be a trapping state. The expected time to search for an object from this state is always 0. From any other state, action a may only be taken if there is a node of that type available for search. In such a case, the expected time required to take action a is a constant, t_a, so the cost, $c(\mathbf{x}, a)$, of taking action a from state $\mathbf{x}$ satisfies:

$$E[c(\mathbf{x}, a)] = t_a I_{\mathbf{x} \neq \langle \mathrm{T} \rangle}$$

A node type is termed *most rewarding* if it achieves the maximum reward rate. A node is termed *most rewarding* if it is of a most rewarding type. In the exposition that follows, we use I^* to denote the set of most rewarding node types.

We define an *exhaustive search* of type i nodes as a sequence of searches of type i nodes, which terminates either upon finding an object or when there are no more nodes of type i available for search, whichever happens first.

Now define a *k-exhaustive search* of nodes of type i as a sequence of searches of type i nodes which terminates either as soon as it finds an object or when there are no more nodes of type i are available, or when it has carried out k searches, whichever happens first. The above defined exhaustive search is therefore equivalent to an ∞-exhaustive search by this definition.

A *simple-minded* policy is defined as a policy which carries out an exhaustive search of the most rewarding node type(s) whenever possible.

We now explain the equivalence with tree search. Each logical primitive corresponds to a node in the search tree. If search reveals the node to contain an object, this is interpreted as discovering that the corresponding logical primitive is true. We shall disregard degenerate search 'opportunities' — nodes which have no probability of containing an object — and so the only way to conclude for certain that L is false is to show that each of the L_i is false. This corresponds to the game ending when all the nodes are exhausted.

3.3 Nature of the Optimal Policy

Our analysis proceeds in three stages. In Subsection 3.3.1, we prove that the optimal policy must be simple-minded. This establishes the optimal action from states in which there are any most rewarding nodes available for search. We then show how it is possible to treat an exhaustive search in a similar

fashion to the search of a single node. The next section then uses this to prove Theorem 3.2, which then establishes a connection between a problem with n node types and a problem with $n-1$ transformed node types.

Finally, Subsection 3.3.3 establishes that dynamic programming may be used to solve the problem by recursive application of Theorem 3.2, proving the optimal policy to be a simple priority order rule. The structure of this proof is identical to that of the proof by Tsitsiklis[85] for semi-Markov bandits.

3.3.1 A Restriction on the Optimal Policy

To simplify the following theorem, we slightly modify our conception of the game. Suppose that, rather than stopping once an object has been found, all policies keep searching as long as any boxes are available for search. The equivalence between the games is maintained by assuming that searches carried out once an object has been discovered are made at no cost. This construction is useful since it allows conditioning upon the states encountered to be independent of whether or not an object has been found.

The state we denote as $(W, \mathbf{X})$, where X has its previous meaning, and W keeps track of whether an object has been found, assuming value 0 if an object has been found, and value 1 if not. Hence:

$$E[c(W, \mathbf{X}, a)] = W t_a$$

Theorem 3.1 *A policy which is not simple-minded cannot be optimal.*

Proof: Consider an arbitrary policy, π, which is not simple-minded. It is shown that there exists a policy, π', which achieves a strictly better payoff. Define the stopping time, $N < \infty$, as the time at which policy π first breaks the simple-minded criterion. (That is $\pi_N(\mathbf{x}_N) \notin I^*$, although for some $i^* \in I^*$, $\mathbf{x}^\pi_{N_{i^*}} > 0$ and so i^* would have been a legal action in state $\mathbf{x}^\pi_N$). The payoff achieved by a policy π, $V_\pi[\mathbf{x}]$, we break into three parts. The expected cost of actions taken from states $\mathbf{X}_0$ up to $\mathbf{X}_{N-1}$ is written A. The expected cost of actions taken from the state in which policy π first misses a chance of searching a node of type i^*, $\mathbf{X}_N$, until the state in which it next takes the chance, $\mathbf{X}_M$, is written B. The expected cost of actions after this point is written C. Note that the time at which policy π first takes action i^* again, $M \leq \infty$, is a stopping time. (If policy π never does, $M = \infty$ and $C = 0$).

$$
\begin{aligned}
V_\pi[\mathbf{x}] &= E\left[\sum_{i=0}^{\infty} c(W_i^\pi, \mathbf{X}_i^\pi, \pi_i(\mathbf{X}_i^\pi))\right] = A + B + C \\
\text{where :} & \\
A &= E\left[\sum_{i=0}^{N-1} c(W_i^\pi, \mathbf{X}_i^\pi, \pi_i(\mathbf{X}_i^\pi))\right] \\
B &= E\left[\sum_{i=N}^{M} c(W_i^\pi, \mathbf{X}_i^\pi, \pi_i(\mathbf{X}_i^\pi))\right] \\
&= E\left[\sum_{i=N}^{M-1} c(W_i^\pi, \mathbf{X}_i^\pi, \pi_i(\mathbf{X}_i^\pi)) + c(W_M^\pi, \mathbf{X}_M^\pi, i^*)\right] \\
C &= E\left[\sum_{i=M+1}^{\infty} c(W_i^\pi, \mathbf{X}_i^\pi, \pi_i(\mathbf{X}_i^\pi))\right]
\end{aligned}
$$

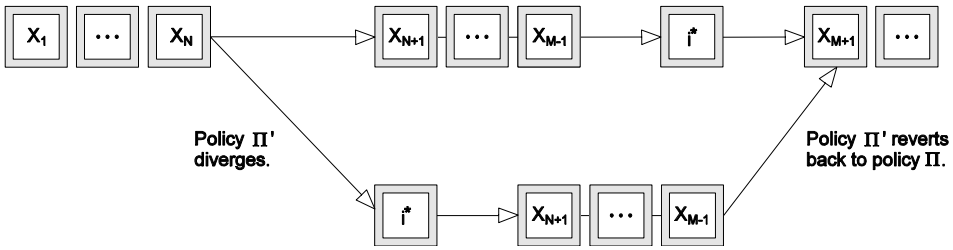


Figure 15: Alternative Policy π'

Consider a policy π' which mimics policy π until time N, when π first misses an opportunity to search a node of some type $i^* \in I^*$. Policy π' now deviates by searching a node of type i^*. This is admissible, since N is a stopping time. The search of a node of type i^* may result in extra nodes of other types being added to the state, but it cannot result in such nodes being removed. It is therefore an admissible policy to mimic the actions taken by policy π once more. Policy π' does this until time M. If $M < \infty$, then at this time policy π searches the node of type i^* which π' searched earlier, and the two policies reconverge and play identically once more.

Denoting by $\mathbf{x} + y$ the state arrived at from $\mathbf{x}$ when a node of type i^* is expanded, policy π' leads to the following series of states:

$$\{\mathbf{X}_0^\pi, \mathbf{X}_1^\pi, \ \dots \ \mathbf{X}_N^\pi, \mathbf{X}_N^\pi + y, \mathbf{X}_{N+1}^\pi + y, \dots \ \mathbf{X}_{M-1}^\pi + y, \mathbf{X}_{M+1}^\pi, \mathbf{X}_{M+2}^\pi, \ \dots \}$$

The payoff achieved by policy π' we express, as before, as the sum of three parts: A' is the expected cost of the first N actions, B' of the next $M - N$, and C' of the remainder:

$$V_{\pi'}[\mathbf{x}] = E\left[\sum_{i=0}^{\infty} c(\mathbf{X}_i^{\pi'}, \pi_i'(\mathbf{X}_i^{\pi'}))\right] = A' + B' + C'$$

Where:

$$\begin{aligned}
A' &= E\left[\sum_{i=0}^{N-1} c(W_i^{\pi'}, \mathbf{X}_i^{\pi'}, \pi_i'(\mathbf{X}_i^{\pi'}))\right] = A \\
B' &= E\left[\sum_{i=N}^{M} c(W_i^{\pi'}, \mathbf{X}_i^{\pi'}, \pi_i'(\mathbf{X}_i^{\pi'}))\right] \\
&= E\left[c(W_i^{\pi'}, \mathbf{X}_N^{\pi'}, i^*) + \sum_{i=N+1}^{M} c(W_i^{\pi'}, \mathbf{X}_i^{\pi'}, \pi_i'(\mathbf{X}_i^{\pi'}))\right] \\
C' &= E\left[\sum_{i=M+1}^{\infty} c(W_i^{\pi'}, \mathbf{X}_i^{\pi'}, \pi_i'(\mathbf{X}_i^{\pi'}))\right] = C
\end{aligned}$$

$$\begin{aligned}
V_\pi[\mathbf{x}] - V_{\pi'}[\mathbf{x}] &= (A + B + C) - (A' + B' + C') = B - B' \\
&= E\left[[E\ D|\ \pi_N(\mathbf{X}_N^\pi) = a_N, \ldots \pi_M(\mathbf{X}_M^\pi) = a_M]\right]
\end{aligned}$$

We consider D, the difference in expected payoff of policies π and π', conditional upon the sequence of actions taken.

$$
\begin{aligned}
D &= \sum_{i=N}^{M-1} c(W_i^\pi, \mathbf{X}_i^\pi, \pi(\mathbf{X}_i^\pi)) + c(W_M^\pi, \mathbf{X}_M^\pi, i^*) \\
&\quad -c(W_N^{\pi'}, \mathbf{X}_N^{\pi'}, i^*) - \sum_{i=N+1}^{M} c(W_i^{\pi'}, \mathbf{X}_i^{\pi'}, \pi_i(\mathbf{X}_i^{\pi'})) \\
&= \sum_{i=N}^{M-1} W_i^\pi t_i + W_M^\pi t_{i^*} - W_N^{\pi'} t_{i^*} - \sum_{i=N+1}^{M} W_{i-1}^{\pi'} q_{i^*} t_{a_{i-1}} \\
&= p_{i^*} \sum_{i=N}^{M-1} t_{a_i} \prod_{j=N}^{i-1} q_{a_j} + t_{i^*} (\prod_{j=N}^{M} q_j - 1) \\
&= t_{i^*} \left(\sum_{i=N}^{M-1} \frac{p_{i^*}}{t_{i^*}} t_{a_i} \prod_{j=N}^{i-1} q_{a_j} + \prod_{j=N}^{M} q_j - 1 \right)
\end{aligned}
$$

Since i^* is a most rewarding node, $\frac{p_{i^*}}{t_{i^*}} > \frac{p_i}{t_i}\ \forall i \notin I^*$.

$$
\begin{aligned}
&= t_{i^*} \left(\sum_{i=N}^{M-1} \frac{p_{a_i}}{t_{a_i}} t_{a_i} \prod_{j=N}^{i-1} q_{a_j} + \prod_{j=N}^{M} q_j - 1 \right) \\
&= t_{i^*} \left(\sum_{i=N}^{M-1} p_{a_i} \prod_{j=N}^{i-1} q_{a_j} + \prod_{j=N}^{M} q_j - 1 \right) \\
&> 0
\end{aligned}
$$

■

We have now established that the optimal policy is simple-minded. This is equivalent to stating that it carries out an exhaustive search for most rewarding nodes at every opportunity. Rather than considering a single search we now consider search of a type i node followed by an exhaustive search for nodes of type i^*. We denote as $\hat{p}_i(i^*)$ the probability of finding an object with such a chuck of search. The expected time taken to search such a chunk

we denote $\hat{t}_i(i^*)$, and let us use $\hat{f}_i(i^*)$ to represent the distribution of nodes revealed. We can now use these values to enable the mathematical treatment of this chunk of search as if it were the expansion of a single node.

3.3.2 An Equivalence Between Search Problems

This section leads to a theorem which establishes an identity between $V(\mathbf{x}, \mathbf{d})$ and the value of a modified problem $V(\mathbf{x}', \mathbf{d}')$ which has one less node type.

Theorem 3.2 *If i^* is a most rewarding node type then*

$$V((x_1 \dots x_{i^*-1}, 0, x_{i^*+1} \dots x_N), (d_1 \dots d_N))$$
$$= V((x_1 \dots x_{i^*-1}, x_{i^*+1} \dots x_N), (d'_1 \dots d'_{i^*-1}, d'_{i^*+1} \dots d'_N))$$

where the transformed node types satisfy

$$d'_i = (\hat{p}_i(i^*), \hat{t}_i(i^*), \hat{f}_i(i^*))$$

Proof: Denote by U' the space of strategies applicable to the game $(\mathbf{x}', \mathbf{d}')$, and by U the space of simple-minded strategies applicable to the game $(\mathbf{x}, \mathbf{d})$. There is a bijection between the two, since any $u' \in U'$ may be expressed as a sequence of substrategies, each of which corresponds to searching a single node, i.e. $u' = (u'_1, u'_2 ...)$, whilst any $u \in U$ may be represented as a conjunction of sub-strategies e_k and u_k, where the $\{e_.\}$ are substrategies that carry out a (possibly zero length) exhaustive search of nodes of type i^*, and the $\{u_.\}$ are searches of a single node of any type other

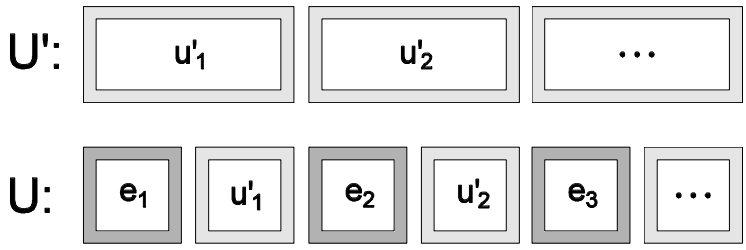


Figure 16: Policy Spaces U and U'

than k. The strategy u may be represented as $u = (e_1, u_1, e_2, u_2 \ldots)$ because it is simple-minded.

The value obtained by applying strategy $u = (e_1, u_1, e_2, u_2 \ldots)$ to game $(\mathbf{x}, \mathbf{d})$ is exactly that obtained by applying strategy $s(u) = (u'_1, u'_2 ...)$ to game $(\mathbf{x}', \mathbf{d}')$, since the effects of carrying out an exhaustive search of nodes of type i^* are accounted for by the transformations $d'_i = (\hat{p}_i(i^*), \hat{t}_i(i^*), \hat{f}_i(i^*))$.

The above bijection argument is easily extensible to policies; if policy π applies strategy u to game $(\mathbf{x}, \mathbf{d})$, define policy $s(\pi)$ to be the policy which applies strategy $s(u)$ to game $(\mathbf{x}', \mathbf{d}')$. This includes the optimum policy, π^*, of game $(\mathbf{x}', \mathbf{d}')$ which Theorem 3.1 tells us is simple-minded:

$$V(\mathbf{x}', \mathbf{d}') = V_{s(\pi^*)}(\mathbf{x}', \mathbf{d}') = V_{\pi^*}(\mathbf{x}, \mathbf{d}) = V(\mathbf{x}, \mathbf{d})$$

■

3.3.3 Proof of Optimal Policy

Theorem 3.3 *For a game with n types of node there exists a permutation,* $(y_1, \ldots y_n)$ *of the integers 1 to n, termed the* optimal ordering*, such that policy* π *is optimal if it searches a node of type* y_k *iff* $k = min\{i : y_i > 0\}$.

Proof: The theorem is trivially true for n=1. For $n > 1$, an induction argument applies. Theorem 3.1 proves that any optimal policy, π, for the game $(\mathbf{x}, \mathbf{d})$ is simple-minded. Since π searches nodes of type i^* iff $x_{i^*} > 0$, let $y_1 = i^*$. The optimal policy has only to be determined when $x_{i^*} = 0$. In this case, Theorem 3.2 proves that for

$$\begin{aligned} \mathbf{x}' &= (x_i \ldots\ x_{i^*-1}, x_{i^*+1}, \ldots\ x_n) \\ \mathbf{d}' &= (d'_i \ldots\ d'_{i^*-1}, d'_{i^*+1}, \ldots\ d'_n) \\ d'_i &= (\hat{p}_i(i^*), \hat{t}_i(i^*), \hat{f}_i(i^*)) \end{aligned}$$

$$V(\mathbf{x}, \mathbf{d}) = V(\mathbf{x}', \mathbf{d}')$$

Now let $(y_2,\ \ldots\ y_n)$ be the optimal ordering of the game $(\mathbf{x}', \mathbf{d}')$, which involves only $n-1$ different types of node, so optimal play is a consequence of the induction hypothesis. ■

The proof considers n different games, since the equivalence proved by Theorem 3.2 is applied $n-1$ times. Each node type is optimal to search in at least one of these games, so consider a node type in a game for which it is optimal. The transformed node details of such a node type account for the chunking of any more rewarding offspring. We therefore define p_i^*, the

corrected probability that a node of type i contains an object, as the p_i value of this game. We define t_i^*, the *corrected* expected time taken to search a node of type i, in a similar way. The ratio of these two we term the *corrected reward rate*, denoted $\emptyset_i^*$. This is a true reflection of the box type's reward rate in the sense that $1/\emptyset_i^*$ is the expected time to find an object when the only available boxes are an unlimited supply of boxes of type i.

3.4 The OR-Tree Model in Practice

The model may be used for situations with a problem which requires individual searches to be carried out in an effort to find a 'solution' (an object). The domain of applicability is narrowed by the satisficing nature of the model. This imposes the following requirements:

1. A problem may have one, none or many solutions.
2. All the solutions are of equal value.

The model also requires that the performance indicator is the *expected use* of some resource (typically time or money) while the problem is being investigated. For the model to be of use, the search must also be of a kind which can be broken down into independent units, of which there are finitely many categories, each of which has its own expected resource requirement, its own known chance of yielding a solution and of creating other search units. The original model also allows search to terminate only when a solution is

found, or when no further units remain to be searched, although we shall relax this requirement in Section 3.6.

Note that 'finding a solution' does not necessarily imply a favourable outcome. In the drug-testing example that follows, if a solution is found then the drug being tested has failed the required tests. Another such example is the task of scheduling a difficult multi-stage manufacturing process. Each stage of the manufacturing process is a node type. Its resource cost is straightforward, while the chance of its finding a solution corresponds to the chance it goes wrong and permanently destroys the product.

A useful consequence of the generality of the model is that it can be applied without modification to cases in which some of the tree is known in advance. This may be achieved by the addition of extra node types. In the general case there is a one-to-one correspondence between the extra node types and the nodes in the out-tree which are known. The search time, t_i and probability of containing an object, p_i, of a node type are set so as to match these values of the node in the known out-tree. The internal nodes in the known tree are represented by node types which have offspring distributions which reflect the fact that it is known for certain exactly which nodes will be made available. In the case where the previously known out-tree contains subtrees which are identical, the number of node types added to the model may be reduced by assigning the matching nodes in the tree to the same node type. The problem of searching a set of boxes or a tree which is wholly pre-determined [23, 52] is thus a special case of this model.

3.4.1 Complexity and Ill-conditioning

Selection of the most rewarding node requires $O(n)$ computations and must be done n times, so this part of the algorithm has complexity $O(n^2)$. The time required to transform the node type details by the equations of Theorem 3.2 depends upon the properties of the descendant distributions, f_i. Repeated slight inaccuracies during calculation of the values of $\hat{p}()$, $\hat{t}()$ and $\hat{f}()$ may be compounded and hence lead to greater inaccuracies later on in calculations, because of the recursive nature of the algorithm. Complexity problems are therefore likely to arise in determining the optimal policy in cases with large numbers of node types and complicated descendent distributions. Such ill-conditioning arises particularly when two or more node types have very similar reward rates, and so the reward rates of the nodes concerned must be calculated to a great accuracy to determine which is optimal. However the practical consequence of such ill-conditioning is that even a sub-optimal policy can be expected to achieve a payoff which is very close to optimal.

If the model is applied to problems which are to be studied repeatedly in real time then the fact that it is computationally expensive to *deduce* the optimal policy may be of little relevance. The model has the great advantage that the optimal policy, once deduced, is very easy to store and apply; it need only be deduced once, and can then be applied swiftly in all states that arise. This feature makes the model suitable for applications such as computer game playing in which speed of exercising sequences of optimal control decisions is at a premium.

In practical applications with a large number of node types which exhibit ill-conditioning, some degree of approximation may well be required. The tension between the wish to deduce a strictly optimal policy and the need to limit the complexity of the calculations required is a matter to be determined by the relative costs involved.

3.4.2 Optimal Search for Conspiracies of Size 1

Although conspiracy numbers may be applied to the usual multi-valued trees, we now consider their application to trees in which all the nodes are scored with one of two values. With a certain loss of information, any evaluation of any game tree may be treated in this way.

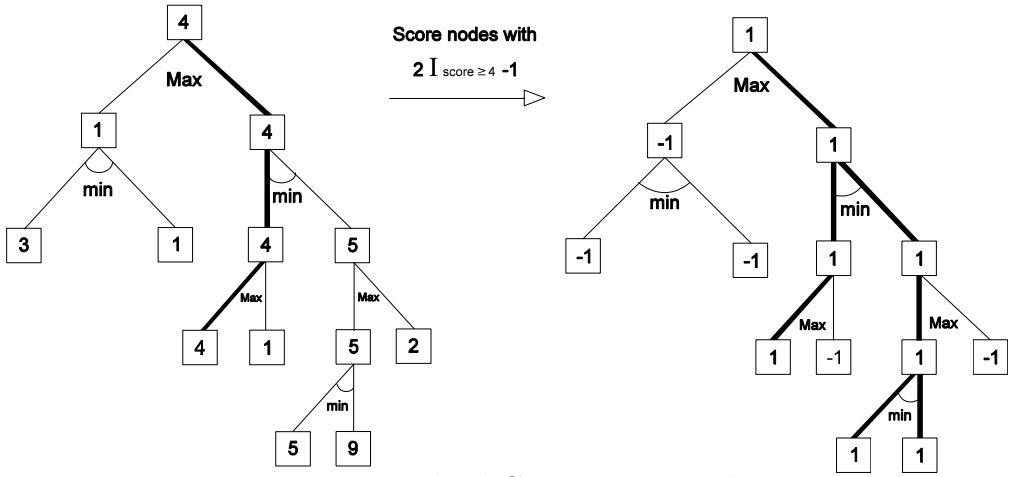


Figure 17: 2-Valued Conspiracy Numbers

Two-valued evaluation functions simplify the calculations about and expositions of conspiracy-based techniques, since the only nodes which can

conspire are nodes on the principal variation (marked in bold in Figure 17). Consider an evaluation function which assigns nodes one of two values, $H(x) \mapsto \{-1, 1\}$. These may be thought of as 'probable win' and 'probable loss' nodes. We follow Knuth and Moore's negamax notation [44]. Hence, assume that there are two sorts of node, -1 and 1. For the sake of simplicity of exposition, let us assume that the game tree and evaluation function are such that:

$$\begin{array}{lcll} d \text{ is a daughter of } p & \Rightarrow & P[H(d) = -H(p)] & = & 1 - \delta \\ & & P[H(d) = H(p)] & = & \delta \end{array}$$

We now show how the tree search model we have developed can be used to analyse the process of searching for conspiracies of size one, and to deduce the optimal search order. The first step of the original conspiracy number search, upon being given a game tree, is to search it for conspiracies of size one. That is, to search it either until expansion of a single leaf has the effect of changing the minimax backed up score at the root, or until we can conclude that there are no more *single* leaf nodes which can do this.

This problem can be seen to fit into the satisficing out-tree search model framework without further adjustment. The state of the investigation may be represented by L, a list of the critical leaves. The search terminates if investigation of a critical leaf finds a conspiracy, or if the list of critical leaves becomes empty. The leaf investigated is removed from L, and any descendants which are critical are placed on L. We assume, for simplicity, that both nodes take one time unit to expand. Optimal policy is to search the

nodes in the order $\{1, -1\}$, as deduced from solving the problem involving the following two node types:

$$\begin{aligned} d_1 &= (1-(1-\delta)^2, 1, {s_{-1}}^2) \\ d_{-1} &= (\delta^2, 1, \tfrac{2\delta(1-\delta)}{1-\delta^2} s_1 + \tfrac{(1-\delta)^2}{1-\delta^2}) \end{aligned}$$

We have seen in Section 1.2 that several authors have derived algorithms which evaluate game positions not only with a scalar score but also with a measure of uncertainty, and that these can yield significant performance gains. In this, simplified example, we suppose, that as well as assigning them a score $\in \{-1, 1\}$ the evaluation function separates its estimates into two classes, types a and b, with different amounts of reliability. There are therefore four types of nodes: $\{+1(a)$, $+1(b)$, -1(a), -1(b)$\}$. A realistic estimate of δ is required for each node type, and could be deduced empirically. If we assume that a node's daughters have a probability, $\dot{p}$, of having the same type as their parent, then as shown overleaf in Figure 18 this model requires the following 4 node types:

$$\begin{aligned} d_{1a} &= (1-(1-a)^2, 1, (\dot{p}s_{-1a} + \dot{q}s_{-1b})^2) \\ d_{1b} &= (1-(1-b)^2, 1, (\dot{p}s_{-1b} + \dot{q}s_{-1a})^2) \\ d_{-1a} &= (a^2, 1, \tfrac{2a(1-a)}{1-a^2}(\dot{p}s_{1a} + \dot{q}s_{1b}) + \tfrac{(1-a)^2}{1-a^2}) \\ d_{-1b} &= (b^2, 1, \tfrac{2b(1-b)}{1-b^2}(\dot{p}s_{1b} + \dot{q}s_{1a}) + \tfrac{(1-b)^2}{1-b^2}) \end{aligned}$$

This problem is in fact the one which inspired the investigation of the tree search model which forms the main subject of Chapter 3. The tree search model, as we have seen, is capable of solving models of considerably

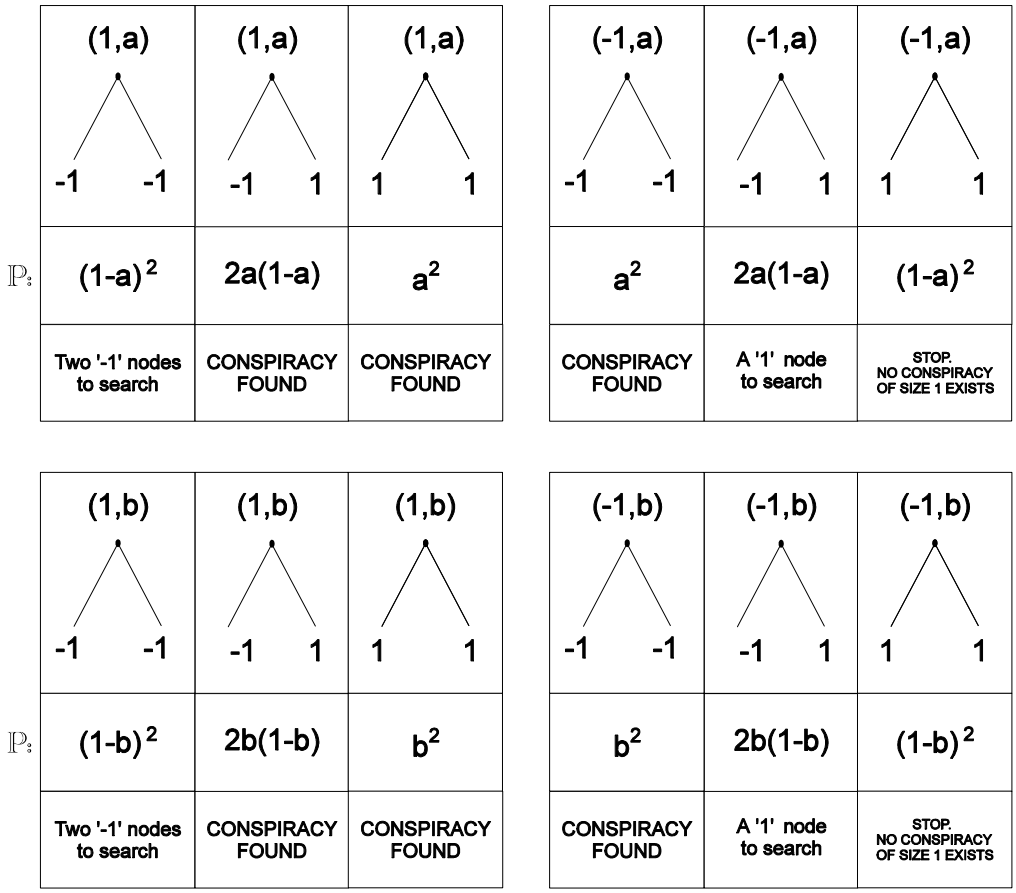


Figure 18: Possible Node Expansion Results

greater generality; the branching factor may be varied as required, as may the number of classes of uncertainty, or the rules about how nodes of one type give rise to nodes of another. Any of these things may be correlated, so as to reflect observations made about a specific game. It is also possible to model the partial expansion of nodes – by which we refer to the creation of less than a full set of node's daughters.

3.4.3 Mathematical Example

For the sake of convenience in the following two examples we shall replace the offspring distribution, f_i, by a multivariate generating function, $G_i(s)$. Consider the search problem involving the three node types below:

$$\begin{aligned} d_1 &= \left(\frac{1}{4}, 6, \frac{1}{3} + \frac{1}{3}s_1s_2 + \frac{1}{3}s_3\right) \\ d_2 &= \left(\frac{1}{2}, 9, \frac{1}{4} + \frac{1}{4}s_1 + \frac{1}{2}s_2\right) \\ d_3 &= \left(0, 1, \frac{2}{3}s_1s_2 + \frac{1}{3}s_1s_2^2s_3\right) \end{aligned}$$

The first step is calculate the reward rates:

$$\begin{aligned} \text{Ø}_1 &= \tfrac{1}{4}/6 = \tfrac{1}{24} \\ \text{Ø}_2 &= \tfrac{1}{2}/9 = \tfrac{1}{18} \\ \text{Ø}_3 &= 0/1 = 0 \end{aligned}$$

Hence, node type 2 is the most rewarding one. In order to deduce the second element of the optimal ordering we require $(\hat{p}_2(1), \hat{t}_2(1), \hat{G}_2(s)(1))$ and $(\hat{p}_2(3), \hat{t}_2(3), \hat{G}_2(s)(3))$, the modified details of the remaining node types. To do this we first chunk together a single search of the most rewarding node type with an exhaustive search of its descendants, to deduce $\hat{p}_2(2)$, $\hat{t}_2(2)$ and $\hat{G}_2(s)(2)$, the characteristics that define an exhaustive search of type 2 nodes carried out upon a single type 2 node.

$$\hat{p}_2(2) = \frac{1}{2} + \frac{1}{4}\hat{p}_2(2)$$

$$
\begin{aligned}
&= \frac{2}{3} \\
\hat{t}_2(2) &= 9 + \frac{1}{4}\hat{t}_2(2) \\
&= 12 \\
\hat{G}_2(s)(2) &= \frac{1}{4} + \frac{1}{4}s_1 + \frac{1}{2}\hat{G}_2(s)(2) \\
&= \frac{1}{2} + \frac{1}{2}s_1
\end{aligned}
$$

Solution of the above equation for $\hat{G}_2(s)$ is a straightforward matter in this case, since a search of the most rewarding node never reveals more than one more node of this same type. This is the case for several classes of problem, including the multi-armed bandit model described in Section 3.5. Numerical methods are required in other cases.

$$d_2 = \left(\frac{2}{3}, 12, \frac{1}{2} + \frac{1}{2}s_1\right)$$

The next step is to amend d_1 and d_3 to reflect what we already know about the optimal ordering, i.e. that if a node of type 2 is revealed then it

will be optimal to search it.

$$\begin{aligned}
\hat{p}_2(1) &= \tfrac{1}{4} + \tfrac{3}{4}\tfrac{1}{3}\hat{p}_2(2) &&= \tfrac{5}{12} \\
\hat{t}_2(1) &= 6 + \tfrac{3}{4}\tfrac{1}{3}\hat{t}_2(2) &&= 9 \\
\hat{G}_2(s)(1) &= \tfrac{1}{3} + \tfrac{1}{3}s_1\hat{G}_2(s) + \tfrac{1}{3}s_3 &&= \tfrac{1}{3} + \tfrac{1}{6}s_1 + \tfrac{1}{6}s_1^2 + \tfrac{1}{3}s_3
\end{aligned}$$

$$\begin{aligned}
\hat{p}_2(3) &= 0 + 1\tfrac{2}{3}\hat{p}_2(2) + 1\tfrac{1}{3}(1 - (1 - \hat{p}_2)^2)) &&= \tfrac{20}{27} \\
\hat{t}_2(3) &= 1 + 1\tfrac{1}{3}\hat{t}_2(2) + 1\tfrac{2}{3}(\hat{t}_2(2) + (1 - \hat{p}_2(2))\hat{t}_2(2)) &&= \tfrac{43}{3} \\
\hat{G}_2(s)(3) &= \tfrac{2}{3}s_1\hat{G}_2(s)(2) + \tfrac{1}{3}s_1(\hat{G}_2(s)(2))^2 s_3 \\
&= \tfrac{1}{3}s_1 + \tfrac{1}{3}s_1^2 + \tfrac{1}{12}s_1 s_3 + \tfrac{1}{6}s_1^2 s_3 + \tfrac{1}{12}s_1^3 s_3
\end{aligned}$$

The modified node details are therefore

$$\begin{aligned}
d_1' &= \left(\frac{5}{12}, 9, \frac{1}{3} + \frac{1}{6}s_1 + \frac{1}{6}s_1^2 + \frac{1}{3}s_3\right) \\
d_3' &= \left(\frac{20}{27}, \frac{43}{3}, \frac{1}{3}s_1 + \frac{1}{3}s_1^2 + \frac{1}{12}s_1 s_3 + \frac{1}{6}s_1^2 s_3 + \frac{1}{12}s_1^3 s_3\right)
\end{aligned}$$

We now compute the reward rates as shown below and conclude that type 3 is the new most rewarding node type since $\emptyset_3 > \emptyset_1$, so the optimal ordering is $\{2,\ 3,\ 1\}$.

$$\begin{aligned}
\emptyset_1 &= \tfrac{5}{12}/9 &&= \tfrac{5}{108} \\
\emptyset_3 &= \tfrac{20}{27}/\tfrac{43}{3} &&= \tfrac{20}{387}
\end{aligned}$$

3.4.4 Drug Testing Example

As a simplified example of a realistic application, suppose permission is being sought to market a newly developed drug. Before a license can be given there

are various statutory tests which must be carried out. Assume that legislation requires that a drug be tested for its allergic potential, a, interaction with other drugs, i, and for effectiveness, e.

The anti-allergenic trial, A, requires that the drug not trigger an allergic reaction in any of four test subjects particularly susceptible to allergic reactions. The drug interaction trial, I, consists of a test upon three test subjects, and the drug is deemed to pass if at most one patient shows evidence of a negative interaction. In order to be marketed the drug also must show evidence of a sufficiently high rate of effectiveness. The drug is deemed to pass the effectiveness trial, E, if it has a therapeutic effect on at least two out of four sufferers.

Suppose that the relative costs of testing a single subject in trials A, I and E are 1:5:4, while previous work developing the drug is such that the prior belief about the parameters is as follows:

$$a_p \sim Beta(1, 22) \qquad e_p \sim Beta(44, 6) \qquad i_p \sim Beta(5, 25)$$

The problem now is to determine *the order in which the trials should be conducted so as to minimise the expected cost until completion.* Define the following nodes. For the anti-allergenic trial:

$$\begin{aligned} d_{a_0} &= (\tfrac{1}{23}, 1, s_{a_1}) \qquad d_{a_1} = (\tfrac{1}{24}, 1, s_{a_2}) \\ d_{a_2} &= (\tfrac{1}{25}, 1, s_{a_3}) \qquad d_{a_3} = (\tfrac{1}{26}, 1, 1) \end{aligned}$$

For the effectiveness trial:

$$
\begin{array}{lcllcl}
d_{e_{00}} & = & (0, 4, \frac{6}{50}s_{e_{10}} + \frac{44}{50}s_{e_{11}}) & d_{e_{11}} & = & (0, 4, \frac{6}{51}s_{e_{21}} + \frac{45}{51}) \\
d_{e_{10}} & = & (0, 4, \frac{7}{51}s_{e_{20}} + \frac{44}{51}s_{e_{21}}) & d_{e_{21}} & = & (0, 4, \frac{7}{52}s_{e_{31}} + \frac{45}{52}) \\
d_{e_{20}} & = & (\frac{8}{52}, 4, s_{e_{31}}) & d_{e_{31}} & = & (\frac{8}{53}, 4, 1)
\end{array}
$$

For the interaction trial:

$$
\begin{array}{lcllcl}
d_{i_{00}} & = & (0, 5, \frac{5}{30}s_{i_{10}} + \frac{25}{30}s_{i_{11}}) & d_{i_{10}} & = & (\frac{6}{31}, 5, s_{i_{21}}) \\
d_{i_{11}} & = & (0, 5, \frac{5}{31}s_{i_{21}} + \frac{26}{31}) & d_{i_{21}} & = & (\frac{6}{32}, 5, 1)
\end{array}
$$

The process of deducing the most rewarding node type and updating the details of other nodes types accordingly is repeated just as before, to deduce the optimal ordering: $\{A_0,\ A_1,\ A_2,\ I_{10},\ A_3,\ E_{20},\ E_{31},\ I_{21},\ I_{00},\ E_{10},\ I_{11},\ E_{00},\ E_{11}\}$. The optimal ordering is sufficient to calculate optimal policy. The problem starts with three nodes available for search, one each of types A_0, E_{00} and I_{00}, of which A_0 has the highest priority, so should be searched first. Then, assuming no allergic reaction is observed, there will be a node of type A_1, and so that will be have the highest priority of the available nodes, and so on. The optimal policy is therefore to carry out the anti-allergenic trial first. Similarly, since all of the $I_{\cdot\cdot}$ node types have a higher priority than the node type E_{00}, it will optimal next to carry out the interaction trial to its conclusion, and then, if necessary, the effectiveness trial.

Further detail can be added as required. If for example, trial I has a set up cost of α which must be paid before any experimentation can take place it suffices to add one more node type:

$$d_i = (0, \alpha, s_{i_{00}})$$

Now suppose, in addition, that the number of separate drug interaction trials required is itself variable and can only be determined by a preliminary investigation at a cost of β. The number of interaction trials required may be modelled by a general discrete distribution, but for the purposes of illustration, let us suppose that it was a $geometric(k)$ distribution; the extra node type added would be:

$$d_{pi} = \left(0, \beta, (1-k)\sum_{n=0}^{\infty}(ks_{i_{00}})^n\right)$$

3.4.5 Computer Software Example

We now present an example of an application that motivates the model extension described in Section 3.6. Suppose that a software developer contracts another firm to check the reliability of a piece of a new product before it is released. For this process, the software involved is broken down into a set of individual modules of code, each of which is supplied with its own specification that describes the intended functionality.

The consultancy firm is paid a certain amount for each module of code investigated. Modules in which a deviation from the accompanying specification is detected are returned to the original company together with details of the bug. Modules in which no fault is detected are guaranteed 'OK' by the consultancy. The nature of the agreement is such that the consultancy is obliged to pay a certain levy for every module which is guaranteed 'OK' but later discovered to contain a bug.

Modern methods of software design mean that each module of code may

be well modelled as a separate problem. The 'object' that is searched for is a fault in the code. The node types are submodules of code. Upon investigation, each submodule of code may be shown to be functioning incorrectly (object detected), to be functioning correctly (no object detected, no offspring generated), or to have a functionality that depends upon the conjunction of the functionality of one or more other submodules (no object detected, offspring generated).

Early retirement is an essential feature for application of the model to this problem, since in the marketplace it is simply too time consuming to test a program to the point where one can have 100% confidence that no bugs exist. With the extra assumption that once the software is released onto the marketplace, any module containing a bug will eventually be detected as such, this means that the expected cost of ceasing investigations into a particular module and declaring it 'OK' is proportional to the probability that it contains a bug, which is the form of the early retirement function specified in Subsection 3.6.1.

3.5 Bandits

We now review a class of standard models referred to as *bandit problems*. A one armed-bandit is in one of a finite number of states. When activated it returns a random payoff the expectation of which is dependent upon that state. Each activation also entails a random transition to another state, according to known probability distribution. A multi-armed bandit is a problem com-

posed of a set of one-armed bandits, in which the operator of the bandits can choose which project to activate, and is aiming to do so in a manner that maximises his expected reward.

If there is no discounting, the multi-armed bandit problem is a discrete Markov problem. With discounting, the length of time taken for each activation becomes important. If this is allowed to vary randomly, the problem becomes semi-Markov – as well as the states of the projects, the time at which the last transition occurred is also relevant. In this model future rewards are continuously discounted by α.

3.5.1 Gittins Indices

In 1979, Gittins[25] proved that it is optimal to allocate each 'arm' of the bandit a separate index [15] depending only upon the state of that project, and then the projects with the greatest available indices. Gittins' proof proceeds by comparing projects with a 'standard project' which yields a constant stream of rewards.

Whittle[89] produced a more natural proof of the optimality of Gittins indices, and provided an interpretation of them by introducing the idea of a 'retire' option. *Retirement* from a bandit process results in receipt of a single amount referred to as a *terminal reward*.

[15] Gittins termed them *dynamic allocation indices*, but at the suggestion of Whittle[89], *Gittins index* has become standard

3.5.2 Branching

Whittle[89] developed his proof of the optimality of Gittins indices to address a problem considered by Nash[53], in which the number of projects is not constant. He assumed, for convenience, that projects fall into one of finitely many classes, each of which has finitely many states. He proved the optimality of the Gittins index policy for the case in which new projects arrive at a known random rate. He referred to this model as the 'arm-acquiring bandit'.

The 'branching bandits' model of Weiss[88] is a very powerful generalisation of the semi-Markov multi-armed bandit model. It allows the number of new projects that arrive, termed 'descendants' by Weiss, to depend in a general fashion upon the project activated, and so the arm-acquiring model, in which it is not, becomes a simple special case. Weiss proves the optimality of a Gittins index policy.

A welcome development in the theory of semi-Markov multi-armed bandits was a very simple proof of the optimality of index policies by Tsitsiklis[85]. This owes quite a lot to Weiss' paper and is generalisable to the branching bandit case. Its key feature, however, an induction on the cardinality of the bandit's statespace, makes it considerably simpler. Its structure is identical to the independently discovered proof of the optimal policy for the OR-tree model given in Subsection 3.3.3.

The important paper of Bertsimas and Niño-Mora[17] establishes a framework with which to analyse a wide range of stochastic and dynamic schedul-

ing problems in a radically different fashion. Their approach is not based around dynamic programming, but they characterise the optimal policy by using a linear program. This approach yields closed formulae for the maximum reward of a multi-armed bandit. Glazebrook and Garbe[28] use this to develop simpler dynamic programming proofs of the optimality of Gittins index policies for finite state branching bandits, as well as suboptimality bounds.

3.5.3 Search Problem Applications

In his comment on Gittins[25], Kelly[42] points out how multi-armed bandits can be used to solve the 'boxes' search problem, as described in Section 3.1, with overlook probabilities. He does this by considering a family of alternative bandit processes, with no transition costs, that give a reward of α^t it the object is found at time t. The issue of stopping is conveniently dealt with by assuming that the searcher does not know whether the object has been found. Kelly also explains how another classic problem, the 'gold-mining', or 'bombing', problem first formulated by Bellman[11] can be solved by applying the multi-armed bandit framework.

As the solution to both of these problems was already known, he remarks, the real advantage of formulating them as bandit problems is that this illustrates how slightly more general problems may also be fitted into the framework. As an example, he supposes the problem in which the probabilities of some of the boxes were not known exactly; the learning that results

from unsuccessful search can easily be fitted into the bandit framework.

The paper by Kadane and Simon[37] of two years earlier established the optimal policy for the boxes and slices case, and contains a flawed proof of the case in which search of boxes is constrained by a general partial ordering. This case, both with and without discounting, was proved independently by Gittins and Glazebrook[29].

Glazebrook[27] had earlier proved the case in which the precedence constraints formed an out-tree. As pointed out by Gittins[26], it is a simple matter to solve this problem by constructing a branching bandit process.

3.5.4 Link with OR-Tree Model

The 'boxes' search model of Section 3.1 specifies parameters p_i as the probability that a box contains an object, and t_i as the time taken to search it. We now show how it can be understood by using the framework for semi-Markov bandits. As suggested by Kelly in his comment on Gittins[25], each box has an equivalent bandit.

The cost structure, however, is different. The bandits have two states - 'searched' and 'unsearched'. The retirement penalty and costs of searching an already searched bandit should be sufficiently large that it is optimal to search all the unsearched bandits, in some order, and then retire at once.

The expected cost of searching a box, t_i, is minus the expected running reward, R_i, of the activating the corresponding bandit in the 'unsearched' state, suitably adjusted to account for its being paid at time T_i, assuming

continuous discounting at rate $\alpha \in (0, 1]$:

$$R_i = -e^{\alpha T_i} t_i$$

The discounting of the bandits is associated with the possibility of finding the object in the boxes search. To this end, the bandits have the following activation times:

$$T_i = -\frac{ln(1 - p_i)}{\alpha}$$

The boxes problem minimises $\sum t_i$, whilst the associated semi-Markov bandit problem maximises $\sum R_i$.

Construction of an equivalent semi-Markov bandit problem for the linear precedence constraints model of Section 3.1 is straightforward, once the reward rate for semi-Markov bandits has been calculated. From the above formulae:

$$\emptyset_i = \frac{p_i}{t_i} = \frac{1 - e^{-\alpha T_i}}{-R_i e^{-\alpha T_i}} = \frac{1 - e^{\alpha T_i}}{R_i}$$

We observe that this is the reciprocal of the conventional form of the Gittins index.

The stochastic search case is equivalent to the semi-Markov branching bandit model, as described by Weiss[88]. The distribution of descendants, $g_i(s, z_1 \ldots z_N)$ is the the offspring distribution f_i described in Section 3.2.

Tsitsiklis[85] writes in his proof of the Gittins index theorem for semi-Markov bandits:

> The proof given here is very simple and it is quite surprising that it was not known earlier. Perhaps a reason is that for the proof

to go through, we have to consider semi-Markov bandits rather than the usual discrete-time Markov bandits.

It is correct that the method of proof does not work on the standard discrete-time Markov bandit model. However, as made clear by the above there is an equivalence between the semi-Markov bandit model and the Markov tree search model. This clarifies the position of the independently discovered proof given in Subsection 3.3.3. The tree search model has the 'probability of discovering an object' which is equivalent to a nodetype-specific discount factor. This enables the crucial induction step of the proof by providing a means to carry out the chunking of node types.

3.6 Retiring Early

Retirement was introduced to the theory of multi-armed bandits by Whittle[89]. This is the option of permanently rejecting all the bandits, and earning instead a single payoff, termed a *retirement reward*, $M \in \mathbf{R}$. To see how this can be conveniently brought within the existing multi-armed bandit framework, consider a bandit with a single state which yields reward $M/(1-\alpha)$ when activated. Activiating this bandit does not change the state. Therefore once it becomes optimal to do this it will remain so, resulting in a stream of payoffs $M/(1-\alpha)$, $\alpha M/(1-\alpha)\ldots$, equivalent to a one-off payment of M.

Seen in this way, the addition of a retirement option looks more like an interesting feature of the multi-armed bandit framework than a real extension

to it. Indeed, the fact that it was introduced by Whittle principally to facilitate a shorter proof of Gittins Index Theorem would certainly support this view, and may explain why 'retirement' has not been developed much further. The inadequacy of this model of retirement is exemplified by the software development scenario presented in Subsection 3.4.5 above, in which the expected payoff from retiring should be allowed to depend upon the states of the projects. We now address this problem.

Let us add an action, *retirement*, which may be taken at any stage, with the effect of immediately terminating the search, incurring a penalty cost $M(P)$, a function of the probability that there is at least one object somewhere. Whittle's retirement option is equivalent to using a retirement function $M(P) = MI_{P\in(0,1)}$.

The previous model allowed termination only upon discovery of an object or when all search opportunities had been exhausted, and so suggests use of the following penalty function:

$$M(P) = \begin{cases} \infty & : \quad P \in (0,1) \\ 0 & : \quad P \in \{0,1\} \end{cases}$$

This new model is identical to the original one, because of the regularity conditions imposed concerning the probability, p_i'', that there is an object amongst the offspring of a type i node. The two models are not equivalent if there is an i for which $p_i'' = 1$, since an occurrence of such a node would allow an early retirement in the second model which would be prohibited in

the original model (because discovery of such a node shows that an object exists somewhere without actually locating it). Similarly, if there is a node type with $q_i q_i'' = 1$, then the two models again diverge.

We now consider some important retirement functions, in rough order of tractability. Only for the first of these is the optimal policy proved in the general case.

3.6.1 Retire and Say "No"

Theorem 3.4 *The optimal policy for penalty functions of the form* $M(P) = I_{P<1}(a + bP)$, *with* $a \in [0, \infty]$, $b \in (-a, \infty)$ *is to search nodes in decreasing order of* $\emptyset^*$, *until either an object has been found or all the remaining node types have* $\emptyset^* < (a + b)^{-1}$, *at which point it is optimal to retire.*

Proof: We first show that it cannot be optimal to retire if there is a node type i available which has $\emptyset_i^* > (a + b)^{-1}$. Then we show by an interchange argument that, in this situation, it is optimal to search these nodes in order of $\emptyset^*$. Finally, we use a one step lookahead argument to show that if there is no node type i available with $\emptyset_i^* > (a + b)^{-1}$ then it is optimal to retire. The case of $a = \infty$ has already been proved, so we assume $a < \infty$.

$M(P)$ is bounded below by 0. Since this bound is achieved for $P = 1$, it must always be optimal to retire if an object has been found.

When a node of type i is expanded, suppose that with probability q_i no object is found, and that the probability that no object exists amongst nodes

revealed by the search is q_i''. We shall let q_i' be the probability that there is no object amongst the other nodes or their descendants. Hence:

$$P = 1 - q_i q_i' E[q_i'']$$

Denote by V_{π_0} the expected value of immediate retirement, and by $V_{\pi_{i0}}$ the expected value of searching a node of type i and then retiring. Let us now consider the relative merits of these two courses of action.

$$\begin{aligned}
V_{\pi_0} &= a + bP \\
&= a + b(1 - q_i q_i' E[q_i'']) \\
V_{\pi_{i0}} &= t_i + q_i E[M(1 - q_i' q_i'')] \\
&= t_i + q_i M(E[1 - q_i' q_i'']) \quad \text{as } \mathrm{q_i' q_i''} > 0 \text{ and M() is linear in } [0, 1) \\
&= t_i + q_i(a + b(1 - q_i' E[q_i''])) \\
V_{\pi_{i0}} - V_{\pi_0} &= t_i + q_i(a + b(1 - q_i' E[q_i''])) - (a + b(1 - q_i q_i' E[q_i''])) \\
&= t_i + (a + b)(q_i - 1) \\
&= t_i - (a + b)p_i \qquad (10)
\end{aligned}$$

Retirement is therefore not optimal if there are any nodes of type i available, where $(a + b)p_i > t_i$, that is, if $\emptyset_i > (a + b)^{-1}$.

We now extend the scope of the above line of reasoning to deal not only with boxes, but also with chunks of search. A consequence of their definition is that the reward rate, $\emptyset$, of a partially searched chunk is strictly less than that of the remaining unsearched part. This implies that, as we argued in Subsection 3.1.1 that it is not optimal to intercalate any other search in the

middle of searching a chunk. It is not optimal to retire in the midst of a chunk, since at least one of the following is always true:

1. The searched portion had $\emptyset > (a+b)^{-1}$.

2. The unsearched portion has $\emptyset > (a+b)^{-1}$.

In the former case, equation (10) implies that it would have been better to retire before starting search of the chunk, whilst in the latter, it implies that it would be better to complete search of the chunk before retiring.

Up to now, we have seen that search should proceed in chunks, and should not stop as long as there are no nodes available with $\emptyset^* \geq (a+b)^{-1}$. Equation (10) implies that, once this point has been reached, immediate retirement is a better policy than carrying out one (or by induction, many) further searches and then retiring. This establishes the set of nodes which it is worth searching before retiring as those with $\emptyset^* > (a+b)^{-1}$. The optimal policy must search those in the order which minimises the expected time taken to discover an object. This is exactly the problem of Section 3.3, which was shown in Theorem 3.3 to be solved by searching the chunks in decreasing order of $\emptyset^*$.

■

3.6.2 Retire and Guess

Now suppose that upon retirement a guess is taken as to whether or not an object exists, and a constant cost of K is incurred for an incorrect guess.

This corresponds to a retirement function of $M(P) = K(\frac{1}{2} - |P - \frac{1}{2}|)$, where P is the overall probability that an object exists.

Lemma 3.5 *If $P \geq \frac{1}{2}$, it is not optimal to search a box or sequence of boxes and then retire unless at this point $P < \frac{1}{2}$, or an object has been found.*

Proof: We show that the lemma holds for a single box, from which the result for a sequence follows immediately by induction. Let policy π_i be the policy of searching box i and then retiring.

$$\begin{aligned} V_{\pi_{i0}} &= t_i + q_i M(1 - q_i') \\ V_{\pi_0} &= M(1 - q_i q_i') = K q_i q_i' \\ V_{\pi_{i0}} - V_{\pi_0} &= t_i + q_i M(1 - q_i') - K q_i q_i' \end{aligned}$$

If the policy retires such that $1 - q_i' \geq \frac{1}{2}$:

$$V_{\pi_{i0}} - V_{\pi_0} = t_i + q_i K q_i' - q_i K q_i' = t_i > 0$$

■

Theorem 3.6 *If all the nodes available for search are boxes, the optimal policy with retirement penalty function $M(P) = K(\frac{1}{2} - |P - \frac{1}{2}|)$ is either to retire immediately or to search the boxes in decreasing order of Ø until an object is found or until no boxes remain with $Ø > K^{-1}$.*

Proof:

$P \in [0, \frac{1}{2}] \cup \{1\}$:

We observe that no sequence of searches can cause P to assume a value in the interval $(\frac{1}{2}, 1)$, successful search fixes P as 1 and unsuccessful search

decreases it. Over this domain $M(P) = K(\frac{1}{2} - |P - \frac{1}{2}|)$ assumes identical values to the function $M(P) = I_{P<1}KP$, so Theorem 3.4 proves the result.

$P \in (\frac{1}{2}, 1)$:

Lemma 3.5 establishes that an optimal policy which searches at all must continue to do so until $P \in [0, \frac{1}{2}] \cup \{1\}$. Such an optimal policy, therefore, cannot leave unsearched any boxes with $\emptyset > K^{-1}$, from application of Theorem 3.4, always assuming no object is found. This establishes that an optimal policy must be prepared to search all the boxes with $\emptyset > K^{-1}$. The standard interchange argument proves that unless it does so in decreasing order of $\emptyset$ it can be improved upon by a policy which plays a suitably permutated strategy.

To see that it is optimal not to include in the search any boxes with $\emptyset \leq K^{-1}$, we consider the payoff of such a policy, π_A.

$$\begin{aligned} V_{\pi_A} &= \sum_{i=1}^{m} t_i \prod_{j=1}^{i-1} q_j + \prod_{i=1}^{m} q_i M \left(1 - \prod_{j=m+1}^{n} q_j\right) \\ &= \sum_{i=1}^{m} t_i \prod_{j=1}^{i-1} q_j + \prod_{i=1}^{m} q_i K \left(1 - \prod_{j=m+1}^{n} q_j\right) \\ &= \sum_{i=1}^{m} t_i \prod_{j=1}^{i-1} q_j + K \prod_{i=1}^{m} q_i - K \prod_{i=1}^{n} q_i \end{aligned}$$

Now consider the payoff of a policy, $\pi_{A'}$, which omits search of box m:

$$\begin{aligned} V_{\pi_{A'}} &= \sum_{i=1}^{m-1} t_i \prod_{j=1}^{i-1} q_j + \prod_{i=1}^{m-1} q_i M \left(1 - \prod_{j=m}^{n} q_j\right) \\ &\leq \sum_{i=1}^{m-1} t_i \prod_{j=1}^{i-1} q_j + \prod_{i=1}^{m-1} q_i K \left(1 - \prod_{j=m}^{n} q_j\right) \\ &\leq \sum_{i=1}^{m} t_i \prod_{j=1}^{i-1} q_j + K \prod_{i=1}^{m-1} q_i - K \prod_{i=1}^{n} q_i \end{aligned}$$

Hence:

$$\begin{aligned} V_{\pi_A} - V_{\pi_{A'}} &= t_m \prod_{j=1}^{m-1} q_j - p_m K \prod_{i=1}^{m-1} q_i \\ &= \prod_{j=1}^{m-1} q_j (t_m - p_m K) \\ &\geq 0 \qquad \text{if } \emptyset_m \leq K^{-1} \end{aligned}$$

Since policy π_A considers m boxes in decreasing order of $\emptyset_i$, this establishes that it is optimal not to consider any which have $\emptyset_i \leq K^{-1}$. ■

We assume for convenience that the boxes are indexed in decreasing order of $\emptyset$, so $i < j$ implies that $\emptyset_i \geq \emptyset_j$. Theorem 3.6 implies that the following policy is optimal either for $j = 0$ or for the largest j such that $\emptyset_j \geq K^{-1}$.

Policy π_j searches boxes $1 \ldots j \leq n$ until it finds an object. If no object is found, it then retires.

Corollary 3.7 *If $P > \frac{1}{2}$, policy π_j may be optimal iff $\prod_{i=1}^{n} q_i > 1 - \prod_{i=j+1}^{n} q_i$. In this case, the critical value of K, for which immediate retirement gives the*

same payoff as a policy π_j above, is given by the below:

$$K^{-1} = \frac{\prod_{i=j+1}^{n} q_i + \prod_{i=1}^{n} q_i - 1}{\sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k}$$

Proof:

$$\begin{aligned}
V_{\pi_j} - V_{\pi_0} &= \sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k + M\left(1 - \prod_{i=j+1}^{n} q_i\right) - M\left(1 - \prod_{i=1}^{n} q_i\right) \\
&= \sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k + K\left(1 - \prod_{i=j+1}^{n} q_i\right) - K\left(\prod_{i=1}^{n} q_i\right) \\
&= \sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k + K\left(1 - \prod_{i=j+1}^{n} q_i - \prod_{i=1}^{n} q_i\right) \\
&> 0 \qquad\qquad \text{if } \left(1 - \prod_{i=1}^{n} q_i \geq \prod_{i=j+1}^{n} q_i\right)
\end{aligned}$$

We have shown that if $\left(1 - \prod_{i=1}^{n} q_i \geq \prod_{i=j+1}^{n} q_i\right)$ then it is always optimal to retire. In the remaining cases, the critical value of K is calculated as follows:

$$\begin{aligned}
\sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k + K\left(1 - \prod_{i=j+1}^{n} q_i - \prod_{i=1}^{n} q_i\right) &= 0 \\
\sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k &= K\left(\prod_{i=j+1}^{n} q_i + \prod_{i=1}^{n} q_i - 1\right) \\
K^{-1} &= \frac{\prod_{i=j+1}^{n} q_i + \prod_{i=1}^{n} q_i - 1}{\sum_{i=1}^{j} t_i \prod_{k=1}^{i-1} q_k}
\end{aligned}$$

■

Linear Precedence Constraints. We now add to the model constraints about the order in which the boxes may be searched, as illustrated overleaf. We suppose the boxes are indexed so that for $j > 0$, box $(i, j+1)$ cannot be searched until box (i, j) has been searched.

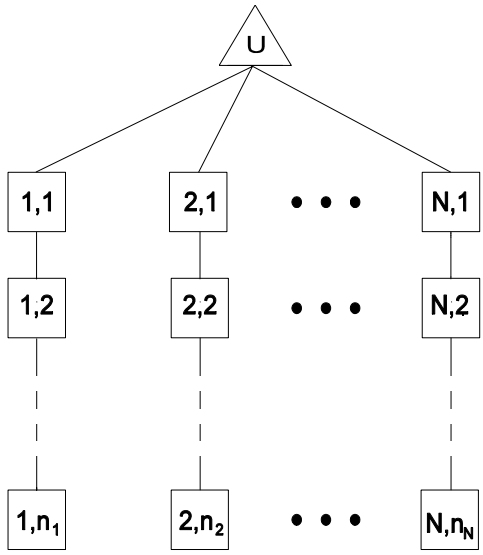


Figure 19: Linear Precedence Constraints

This structure is that of the model of Subsection 3.1.1, and the problem can be analysed in the same fashion. That is, we work backwards from the end of each stack, parsing the tree into sections that must be searched together.

Consider the last two boxes of the ith stack. If $\emptyset_{(i,n_i-1)} < \emptyset_{(i,n_i)}$ then these two boxes are part of a single chunk of search. The chunking procedure used is identical to that in Subsection 3.1.1. The validity of the chunking process is established by the result below, which will be applied recursively

to each stack:

Theorem 3.8 *If $\emptyset_{(i,n_i-1)} < \emptyset_{(i,n_i)}$ then any optimal policy which searches box $(i, n_i - 1)$ must search box (i, n_i) next, unless an object is found.*

Proof:

If $\emptyset_{(i,n_i)} < K^{-1}$:

Theorem 3.6 implies that no optimal policy would search either box (i, n_i) or box $(i, n_i - 1)$ even if there were no constraints. Adding constraints can never increase the payoff from searches, and so the result is proved because no optimal policies search $(i, n_i - 1)$.

If $\emptyset_{(i,n_i)} \geq K^{-1}$:

Let $\pi_{(i,n_i-1)}$ be a policy which searches box $(i, n_i - 1)$, then some (possibly empty) set of boxes, j, then retires, leaving box (i, n_i) unsearched. Denoting by q'_{ij} the probability of there being no object amongst the boxes other than those in stack i or set j, the payoff of policy $\pi_{(i,n_i-1)}$ can be expressed as follows:

$$V_{\pi_{(i,n_i-1)}} = t_{(i,n_i-1)} + q_{(i,n_i-1)}t_j + q_{(i,n_i-1)}q_j M(1 - q_{(i,n_i)}q'_{ij})$$

We shall compare the payoff of this this with that of two others, policy π_1, which differs from policy $\pi_{(i,n_i-1)}$ in that it searches box (i, n_i) before retiring and with policy π_0, that of immediate retirement. These policies have the following payoffs:

$$\begin{aligned} V_{\pi_0} &= M(1 - q_{(i,n_i-1)}q_{(i,n_i)}q_j q'_{ij}) \\ V_{\pi_1} &= t_{(i,n_i-1)} + q_{(i,n_i-1)}t_j + q_{(i,n_i-1)}q_j t_{(i,n_i)} + q_{(i,n_i-1)}q_j q_{(i,n_i)} M(1 - q'_{ij}) \end{aligned}$$

If $q_{(i,n_i)}q'_{ij} \geq \frac{1}{2}$, policy π_1 is an improvement:

$$\begin{aligned} V_{\pi_{(i,n_i-1)}} - V_{\pi_1} &= q_{(i,n_i-1)}q_j M(1 - q_{(i,n_i)}q'_{ij}) \\ &\quad -q_{(i,n_i-1)}q_j t_{(i,n_i)} - q_{(i,n_i-1)}q_j q_{(i,n_i)} M(1 - q'_{ij}) \\ &= q_{(i,n_i-1)}q_j(K(1 - q_{(i,n_i)}q'_{ij}) - t_{(i,n_i)} - q_{(i,n_i)}K(1 - q'_{ij})) \\ &= q_{(i,n_i-1)}q_j(Kp_{(i,n_i)} - t_{(i,n_i)}) \\ &= q_{(i,n_i-1)}q_j K t_{(i,n_i)}(\emptyset_{(i,n_i)} - K^{-1}) \ > \ 0 \end{aligned}$$

If $q_{(i,n_i)}q'_{ij} < \frac{1}{2}$, policy π_0 is an improvement:

$$\begin{aligned} V_{\pi_{(i,n_i-1)}} - V_{\pi_0} &= t_{(i,n_i-1)} + q_{(i,n_i-1)}t_j + q_{(i,n_i-1)}q_j M(1 - q_{(i,n_i)}q'_{ij}) \\ &\quad -M(1 - q_{(i,n_i-1)}q_{(i,n_i)}q_j q'_{ij}) \\ &= t_{(i,n_i-1)} + q_{(i,n_i-1)}t_j + q_{(i,n_i-1)}q_j K(q_{(i,n_i)}q'_{ij}) - K(q_{(i,n_i-1)}q_{(i,n_i)}q_j q'_{ij}) \\ &= t_{(i,n_i-1)} + q_{(i,n_i-1)}t_j \ > \ 0 \end{aligned}$$

We now consider the remaining case, that in which a policy π searches box (i, n_{i-1}) and then intercalates search of a non-empty set of boxes, j, before searching (i, n_i). Since either $\emptyset_{(i,n_i)} > \emptyset_j$ or $\emptyset_j > \emptyset_{(i,n_{i-1})}$, policy π can be improved by a simple interchange argument, as the constraint structure does not debar swapping j with either (i, n_{i-1}) or, once (i, n_{i-1}) has been searched, with box (i, n_i). ■

The case of linear precedence constraints can therefore be solved by applying the same chunking process that was used on the model without retirement. This reduces it, as before, to the unconstrained boxes case treated above.

We now consider what can be deduced about the general OR-tree model with early retirement function $M(P) = K(\frac{1}{2} - |P - \frac{1}{2}|)$. From the form of

this function, we see that as $K \to 0$, the cost of early retirement becomes vanishingly small, and so, for small enough K, the optimal policy is to retire immediately. As $K \to \infty$, early retirement becomes increasingly expensive, so the optimal policy tends to that of the model without the retirement option. Note, however, that the models are only asymptotically equivalent, since for any fixed K, $M(P) \to 0$ as extra nodes are added, and so retirement is still optimal from states with small enough P.

By comparing V_{π_0} with $V_{\pi_{i0}}$ we can prove the following necessary (though not sufficient) condition for states in which it is optimal to retire. We denote by q_i' the probability that no object exists outside box i or its offspring.

Theorem 3.9 *For it to be optimal to retire there must not be a node of type i available with $q_i' \geq \frac{1}{2}, \emptyset_i > K^{-1} + I_{q_i' q_i \leq \frac{1}{2}} \frac{1-2q_i' q_i}{t_i}$.*

Proof: If node i is a box:

$$\begin{aligned}
V_{\pi_0} &= M(1-q_i'q_i) \\
V_{\pi_{i0}} &= t_i + q_i M(1-q_i') \\
V_{\pi_{i0}} - V_{\pi 0} &= t_i + q_i M(1-q_i') - M(1-q_i'q_i) \\
&= t_i + q_i K(\tfrac{1}{2} - |\tfrac{1}{2} - q_i'|) - K(\tfrac{1}{2} - |\tfrac{1}{2} - q_i'q_i|)
\end{aligned}$$

$$\begin{aligned}
\text{For } q_i'q_i \geq \tfrac{1}{2}: \quad &= t_i + q_i K(1-q_i') - K(1-q_i'q_i) \\
&= t_i + K(q_i - q_i'q_i - 1 + q_i'q_i) \\
&= t_i + K(q_i - 1) \\
&= t_i - Kp_i \\
&< 0 \text{ iff } \emptyset_i > K^{-1}
\end{aligned}$$

$$\begin{aligned}
\text{For } q_i' \geq \tfrac{1}{2} \geq q_i'q_i: \quad &= t_i + q_i K(1-q_i') - K(q_i'q_i) \\
&= t_i + K(q_i - q_i'q_i - q_i'q_i) \\
&= t_i + q_i K(1-2q_i') \\
&< 0 \text{ iff } q_i K(2q_i' - 1) > t_i \\
&\quad \Leftrightarrow \tfrac{2q_i'q_i - q_i}{t_i} > K^{-1} \\
&\quad \Leftrightarrow \tfrac{2q_i'q_i t_i - 1}{t_i} + \tfrac{p_i}{t_i} > K^{-1} \\
&\quad \Leftrightarrow \emptyset_i > K^{-1} + \tfrac{1-2q_i'q_i}{t_i}
\end{aligned}$$

$$\begin{aligned}
\text{For } \tfrac{1}{2} > q_i': \quad &= t_i + q_i K(q_i') - K(q_i'q_i) = t_i \\
&> 0
\end{aligned}$$

Thus:

$$V_{\pi_{i0}} - V_{\pi_0} \quad \begin{cases} < 0 \;\; | \;\; q_i' q_i \geq \frac{1}{2}, & \emptyset_i > K^{-1} \\ < 0 \;\; | \;\; q_i' \geq \frac{1}{2} \geq q_i' q_i, & \emptyset_i > K^{-1} + \frac{1-2q_i' q_i}{t_i} \\ \geq 0 \;\; | \;\; \text{otherwise} & \end{cases} \tag{11}$$

Adding more descendants to a node can only ever increase the desirability of searching it, and so, since we assumed the node type i had no descendants, the result is also valid for any nodes of type i with probability p_i of containing an object. ■

Useless node types. We define a node type as being *useless*, if there is an optimal policy which will never search it, no matter which other boxes are available for search.

Corollary 3.10 *A box of type i with $\emptyset_i \leq K^{-1}$ is* useless.

Proof: Let π_{iA} be an arbitrary policy which starts by searching a box of a type i, with $\emptyset_i \leq K^{-1}$. By A we denote the sequence of searches it carries out before retiring if the initial search is unsuccessful. Inequality (11) of Theorem 3.9 above implies that if A is empty, then policy π_{iA} is no better than a policy of immediate retirement, which we shall denote π_0.

We now address the case of non-empty A. Suppose the expected time to carry out search A is t_A, and that an object is revealed with expected probability p_A. We denote by p_A'' the expected probability that an object

exists amongst the nodes revealed by searching A, and by p'_{Ai} the probability that an object exists amongst the nodes unsearched by policy π_{Ai}.

For $\emptyset_A \geq K^{-1}$:

$$\begin{aligned} V_{\pi_{iA}} &= t_i + q_i(t_A + q_A M(1 - q'_{Ai}q''_A)) \\ V_{\pi_{Ai}} &= t_A + q_A(t_i + q_i M(1 - q'_{Ai}q''_A)) \\ \text{So}: V_{\pi_{iA}} - V_{\pi_{Ai}} &= t_i + q_i(t_A + q_A M(1 - q'_{Ai}q''_A)) \\ &\quad -t_A - q_A(t_i + q_i M(1 - q'_{Ai}q''_A)) \\ &= p_A t_i - p_i t_A \\ &= t_A t_i(\emptyset_A - \emptyset_i) \geq 0 \end{aligned}$$

In this case therefore, policy π_{Ai} is at least as good as policy π_{iA}.

For $\emptyset_A < K^{-1}$:

$$\begin{aligned} V_{\pi_{iA}} &= t_i + q_i(t_A + q_A M(1 - q'_{Ai}q''_A)) \\ V_{\pi_0} &= M(1 - q_i q_A q'_{Ai}q''_A) \\ \text{So}: V_{\pi_{iA}} - V_{\pi_0} &= t_i + q_i(t_A + q_A M(1 - q'_{Ai}q''_A)) - M(1 - q_i q_A q'_{Ai}q''_A) \end{aligned}$$

For $q_i q_A q'_{Ai}q''_A \geq \frac{1}{2}$:

$$\begin{aligned} &= t_i + q_i(t_A + q_A K(1 - q'_{Ai}q''_A)) - K(1 - q_i q_A q'_{Ai}q''_A) \quad (12) \\ &= t_i + q_i t_A + K(q_i q_A - q_i q_A q'_{Ai}q''_A - 1 + q_i q_A q'_{Ai}q''_A) \\ &= t_i + q_i t_A + K(q_i q_A - 1) \end{aligned}$$

Since $\emptyset_A < K^{-1}$ and $\emptyset_i \leq K^{-1}$, $Kp_A < t_A$ and $Kp_i \leq t_i$. Thus:

$$\begin{aligned} &> Kp_i + q_i K p_A + K(q_i q_A - 1) \\ &> K(p_i + q_i p_A + q_i q_A - 1) \end{aligned}$$

$$> \quad 0$$

For $q''_A q'_{Ai} \geq \frac{1}{2} \geq q_i q_A q''_A q'_{Ai}$:

$$= \quad t_i + q_i(t_A + q_A K(1 - q'_{Ai} q''_A)) - K(q_i q_A q'_{Ai} q''_A)$$

$$\geq \quad t_i + q_i(t_A + q_A K(1 - q'_{Ai} q''_A)) - K(1 - q_i q_A q'_{Ai} q''_A)$$

$$> \quad 0 \qquad \text{from (12)}$$

For $\frac{1}{2} \geq q''_A q'_{Ai}$:

$$= \quad t_i + q_i(t_A + q_A K(q'_{Ai} q''_A)) - K(q_i q_A q'_{Ai} q''_A)$$

$$= \quad t_i + q_i t_A$$

$$> \quad 0 \qquad (13)$$

In this case therefore, π_0 is better than π_{iA}.

We have now proved that for any policy π_{iA}, which starts by searching a box of type i, with $\emptyset_i \leq K^{-1}$, there exists an alternative policy which does not, and which achieves a payoff at least as good. The theorem therefore follows by recursive application of this result. ■

Identification of useless node types allows a compression of the description of the state, since the number of such nodes available is relevant only in so far as it influences P, the probability that an object exists somewhere. Thus, if node types $X_{j+1} \ldots X_n$ are useless, they can be 'hidden' in our description of the state, and the vector $(X_1, \ldots X_n)$ replaced by the vector $(X_1, \ldots X_j, h)$ where h represents the 'hidden probability', that is, the probability that an object exists somewhere amongst the nodes of type $j + 1 \ldots n$ or their

descendants. The h value has the effect of changing the effective cost of retirement:

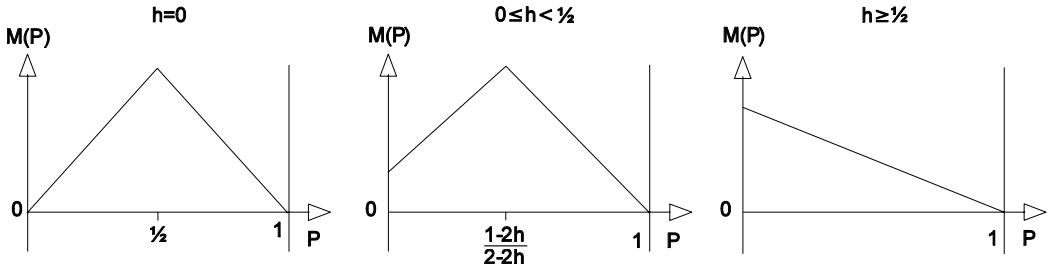


Figure 20: Effective M() for Different Hidden Probabilities

Since the effective $M(P)$ is linear for $h \geq \frac{1}{2}$, the optimal policy from such a state is to retire immediately.

One-step Lookahead Policies. We now consider the class of one-step lookahead policies for this retirement function. A one-step lookahead policy is optimal for the boxes case without retirement, as shown at the start of this chapter. This may be understood to be a consequence of the simplicity of the motives for searching a box. In the case of linear precedence constraints, this simplicity is upset. However, we have seen that if the nodes are grouped into maximal indivisible blocks, and search of a maximal indivisible block is taken to be a single step then the optimal policy is a one-step lookahead policy. The OR-tree search problem may be treated in a similar fashion, with stochastic chunks of search taking the place of the deterministic maximal indivisible blocks. One is therefore prompted to wonder whether the OR-tree search problem with retirement function $M(P) - K(\frac{1}{2} - |\frac{1}{2} - P|)$ can also be solved

by a similar process of chunking together a node with some subset of its descendants.

Suppose that there are three node types, as defined below, with ϵ small but positive.

$$\begin{aligned} A &= (0, \epsilon, \tfrac{10}{11}s_B + \tfrac{1}{11}s_C) \\ B &= (\tfrac{1}{100}, K, 1) \\ C &= (\tfrac{1}{10}, K, 1) \end{aligned}$$

Consider a state in which there are seven nodes of type A available. Observe that $p''_A = \frac{54}{55}$, so the overall probability that an object exists is $P(\mathbf{x}) = 1 - (\frac{54}{55})^7 < \frac{1}{2}$. We note that if each of the type A nodes is searched and gives rise to a type C node, then the probability that an object exists will rise to $1-(\frac{9}{10})^7 > \frac{1}{2}$. Hence, $M()$ is concave but not linear over the range of possible values taken by P if all the type A nodes are searched. Thus, the expected payoff from retirement decreases if these searches are carried out, and it will be optimal to do some searching if ϵ is small enough.

We observe that expansion of a single type A node can only cause P to adopt a value of either $1 - \frac{9}{10}(\frac{54}{55})^6$ or $1 - \frac{99}{100}(\frac{54}{55})^6$. Since $M(P)$ is linear over this range, a one-step lookahead policy would retire rather than carry out any search, and so fail to play optimally.

Hence, *if the optimal policy is to be a one-step lookahead policy, the step size must be such as to consider expansion of all the type A nodes in one go.* This would require a 'horizontal' equivalent of the chunking concept, that is, one which treats sets of siblings as a single chunk[16]. The interaction between these, new, chunks, and the 'vertical' chunking process which defines maximal indivisible blocks does not seem to be tractable. A fuller understanding of the problems with one-step lookahead policies may lead to the development of a more theoretical approach to the choice of step size than that presented in Section 6.3.

3.6.3 Retire and Say "Yes"

Now suppose that upon retirement, a cost of K is incurred if no object exists. This corresponds to a retirement function of $M(P) = I_{P>0}K(1-P)$.

Theorem 3.11 *In the boxes case, if* $K \leq \sum_{i=1}^{n} \prod_{j=1}^{i-1} q_j t_i \,/(1-\prod_{i=1}^{n} q_i)$ *then it is optimal to retire immediately. If* $K \geq \sum_{i=1}^{n} \prod_{j=1}^{i-1} q_j t_i \,/(1-\prod_{i=1}^{n} q_i)$ *then it is optimal to search all the boxes for an object in decreasing order of* $\emptyset_i$.

Proof: Suppose that a non-empty sequence of searches, A, is carried out, which has probability p_A of revealing an object, and takes expected time t_A. Denoting by q'_A the probability that an object exists in the remaining boxes, we compare the payoff of a policy, π_0, of immediate retirement, with policy

[16]There is an analogy here with the 'conspiracy' concept mentioned in Subsection 1.2.2.

π_{A0}, which carries out search of A, and then retires:

$$\begin{aligned} V_{\pi_{A0}} &= t_A + q_A M(q'_A) \\ &= \begin{cases} t_A + q_A K q'_A & \mid \quad q'_A < 1 \\ t_A & \mid \quad q'_A = 1 \end{cases} \\ &> K q_A q'_A = V_{\pi_0} \qquad \mid q'_A < 1 \end{aligned}$$

If $q'_A < 1$ it is therefore better to retire immediately than to carry out such a sequence of searches, A, and then retire. Since we exclude boxes with $p_i = 0$, there is only sequence of searches with $q'_A = 1$. This is made up of all the boxes. The two possible optimal policies are therefore searching all available boxes or retiring immediately. The interchange argument can be used as usual to show that the optimal order in which to carry out search is in order of decreasing $\emptyset$. The payoff from searching all the boxes in this order is $\sum_{i=1}^{n} \prod_{j-0}^{i-1} q_j t_i$, while the cost of immediate retirement is $K\left(1 - \prod_{i-1}^{n} q_i\right)$. ■

The following conjecture, if proved, would give some insight onto the shape of the stopping region for the boxes case.

Conjecture 3.1 *If it is optimal in the boxes case to retire from state* $\mathbf{x}^A$ *and from state* $\mathbf{x}^B$*, then will also be optimal to retire from the state below:*
$\left(\left\lceil \frac{x_1{}^A + x_1{}^B}{2} \right\rceil, \left(\left\lceil \frac{x_2{}^A + x_2{}^B}{2} \right\rceil \ldots \left\lceil \frac{x_n{}^A + x_n{}^B}{2} \right\rceil\right)\right.$.

We present a simple example to show that Conjecture 3.1 is not true in the general case. Consider the following four node types:-

$$\begin{aligned} A &= \left(0, 10, \tfrac{1}{2}s_C + \tfrac{1}{2}s_D\right) & \qquad B &= (\epsilon, 46, 0) \\ C &= (\epsilon, 16, 0) & \qquad D &= \left(\tfrac{1}{2}, K', 0\right) \end{aligned}$$

Let ϵ be positive but negligibly small. We now calculate the optimal course of action from states $(A_1 \cup A_2)$, $(B_1 \cup B_2)$ and $(A \cup B)$, assuming that $K' > K$, so that, if a node of type D is found, the optimal action will be to retire. For some value of K, we will be indifferent between immediate retirement from $(A_1 \cup A_2)$ and searching in an effort to show that no object exists.

$$V(A_1 \cup A_2) = 10 + \frac{1}{2}K + \frac{1}{2}\left(10 + \frac{1}{2}K + \frac{1}{2}(16+16)\right) = 23 + \frac{3}{4}K$$

So, the indifference value of K from state $(A_1 \cup A_2)$ is 92 since this is the solution to $23 + \frac{3}{4}K = K$. From $(B_1 \cup B_2)$, the cost to search and show that there is no object is 92, so this is also the indifference value of K from that state.

$$V(A \cup B) = 10 + \frac{1}{2}K + \frac{1}{2}(46+16) = 41 + \frac{1}{2}K$$

Solving this equation to get the indifference value of K for $(A \cup B)$, we get 82, a *lower* value than that for either $(A_1 \cup A_2)$ or $(B_1 \cup B_2)$, and so conclude that the retirement region in this case is not convex. The lower indifference value can be understood to stem from an interaction of the A and B nodes; for $K < 92$, the expression $(A_1 \cup A_2)$ is too unlikely to be worth searching, while the expression $(B_1 \cup B_2)$ cannot be searched in a way that yields any information.

3.6.4 Other Retirement Functions

We have considered above three of the most natural choices for the retirement function. The 'Retire and Guess' function of Subsection 3.6.2 could be modified without major difficulty to allow for different penalties of type I and type II errors. For many practical applications, $M()$ might not have one of the forms described above. If the node types did not have any simplifying properties, this would probably require some approximation or solution via dynamic programming since complete mathematical treatment of more complicated retirement functions seems likely to be a difficult exercise.

To underline the complexity of the optimal policy for other retirement functions, we present an example with two node types:

$$d_A = \left(\frac{1}{10}, 1, 1\right) \qquad\qquad d_B = \left(\frac{1}{10}, 1, \frac{1}{9} + \frac{8}{9}s_A\right)$$

Table 2 overleaf shows the optimal policy from various states for the retirement function $M(P) = 90I_{P>0.06}$.

```
            1111111111222222222233333333333444444
   01234567890123456789012345678901234567890123456
 0: ........................aaaaaaaaaaaaaaaaaaaa
 1: ......................=b==================
 2: .....................=bbb=================
 3: ....................aaaaaaaaaaaaaaaaaaaa
 4: ..................=b====================
 5: ................=bbb====================
 6: ..............=bbbbb====================
 7: ............=bbbbbbb====================
 8: ..........=bbbbbbbbb====================
 9: ........=bbbbbbbbbbb====================
10: ......=bbbbbbbbbbbbb====================
11: ....=bbbbbbbbbbbbbbb====================
12: ...aaaaaaaabbbbbbbb====================
13: ..=b========bbbbbbbbb====================
14: .=bbb=======bbbbbbbbbb====================
15: bbbbb======bbbbbbbbbb====================
16: bbbbb=====bbbbbbbbbb====================
17: bbbbb====bbbbbbbbbbb====================
18: bbbbb===bbbbbbbbbbb====================
19: bbbbb==bbbbbbbbbbbb====================
20: bbbbb=bbbbbbbbbbbb====================
21: bbbbbbbbbbbbbbbbb====================
22: bbbbbbbbbbbbbbbbb=======
```

`.` : Optimal to retire.
`a` : Optimal to search a type A node.
`b` : Optimal to search a type B node.
`=` : Optimal to search either node.

Table 2: Optimal Policy for an Alternative Retirement Function

The number of A nodes is along the x axis, the number of B nodes is along the y axis. The optimal policy was calculated by dynamic programming. The program is included in Appendix C.

3.7 Overlook Probabilities

We now modify the model to allow the inclusion of overlook probabilities, as the original satisficing search model in the boxes case of Section 3.1 has been modified by Hall [31], Stone [82] and Wegener [87]. We now assume that when a node of type i contains an object and is searched there is a chance that the object will not be detected. This causes a subtle yet important change to the problem; if a node type with a non-zero overlook probability is available, then however many searches are carried out, it is impossible to conclude for certain that no object exists, because of the possibility of repeatedly having overlooked an object.

Let us revise our initial optimality criterion, 'expected time to termination'. We first observe that if no object exists, then it does not matter in which order the boxes are searched. Hence:

$$\begin{aligned} V_\pi &= E[V_\pi I_{\text{Object exists}} + V_\pi I_{\text{No object exists}}] \\ &= E[V_\pi I_{\text{Object exists}}] + v_{\text{All nodes}} \\ &= E[V_\pi|\text{Object exists}]P(\text{Object exists}) + v_{\text{All nodes}} \end{aligned} \tag{14}$$

Since $v_{\text{All nodes}}$ is a constant, equation (14) shows that a policy which is optimal in the original sense of minimising 'expected time to termination' also minimises the 'expected time to termination given that an object exists'.

There is a positive probability that no object exists, which, with overlook probabilities, causes search to continue indefinitely. We are therefore required to modify the initial definition of optimality if we wish to use it to

discriminate between policies, and so we restrict our attention to cases in which the choice of policy makes a difference – i.e. those cases in which an object exists. Instead of minimising expected time until termination we shall be minimising expected termination time given that an object exists. This modified definition of optimality does not conflict in any way with the one used up to this point, and indeed supersedes the previous definition which was introduced on grounds of simplicity.

Extend node type $d_i = (p_i, t_i)$ by adding $o_i \in [0, 1)$ as an extra parameter, which we shall refer to as the overlook probability. Since there is now no limit to the number of times it may be worthwhile to search a node of type i, we add another subscript to signify the number of times each node has been searched. Define node type (i, j) to be a node of type i which has been searched j times without success. Thus, to add an overlook probability of o_i to node type i, consider each type i node to be of type $(i, 0)$, and replace the single node type i by a family of node types i, j defined as follows:

$$t_{i,j} = t_i \qquad\qquad p_{i,j} = \frac{o_i^j(1 - o_i)p_i}{q_i + p_i o_i^j}$$

We modify the offspring distribution to ensure that the first time a node of type i is searched, a node of type $(i, 1)$ is generated in addition to any offspring revealed, whilst unsuccessful search of an (i, j) node reveals exactly one node, of type $(i, j + 1)$. The only theoretical complications of extending the model in this way arise from the step of the proof in which i^* is set equal

to a node type which maximises $\emptyset_i$. Since there are now an infinite number of node types, we must show that there is a node type which achieves this maximum. We shall do this by showing that $\emptyset_{i,j} \to 0$ as $j \to \infty$. This proves that the maximum is achieved, since it implies that for any $\epsilon > 0$, there are only finitely many $\emptyset_{i,j} \geq \epsilon$.

Before proving this, we shall increase the generality slightly by allowing for non-constant overlook probabilities. Let $o_{i,j}$ be the overlook probability for the j^{th} time a node of type i is searched.

Lemma 3.12 *For any node type i, $\emptyset_{i,j} \to 0$ as $j \to \infty$.*

Proof: The probability that an object is discovered on the j^{th} search of a node of type i is $p_i(1 - o_{i,j}) \prod_{k=0}^{j-1} o_{i,k}$. Hence:

$$\sum_{j=1}^{\infty} p_i(1 - o_{i,j}) \prod_{k=0}^{j-1} o_{i,k} \leq 1 \quad \Rightarrow \quad p_i(1 - o_{i,j}) \prod_{k=0}^{j-1} o_{i,k} \to 0 \ \text{ as j} \to \infty.$$

$$\emptyset_{i,j} \propto p_{i,j} = p_i(1 - o_{i,j}) \prod_{k=0}^{j-1} o_{i,k} \to 0 \ \text{ as j} \to \infty.$$

■

3.8 Continuous Extension of the OR-Tree Model

We now consider an extension to the boxes case of the OR-tree search model of Section 3.1, based upon the understanding of Ø as a reward *rate*. Graphing time spent searching on the X-axis, and the probability that an object is found on the Y-axis, the original model is represented below. Also shown is a continuous representation. In this model, a box with constant reward rate Ø may be searched for time v to reveal an object with probability Øv.

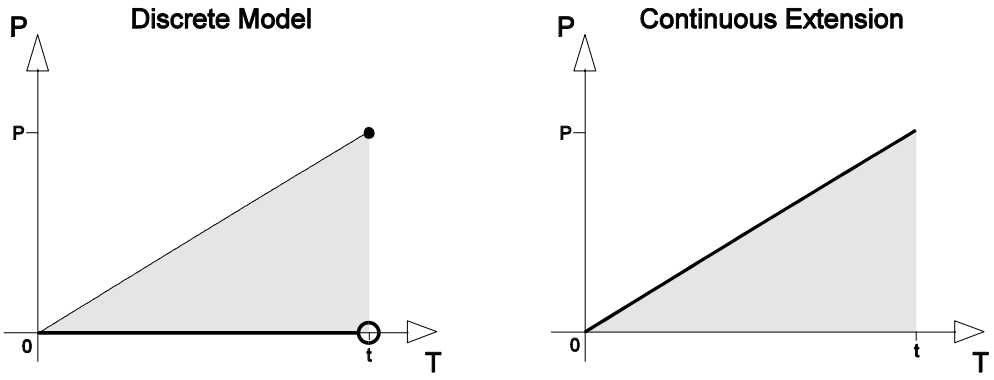


Figure 21: Continuous Extension of the Discrete Model

The boxes model of Section 3.1 has much in common with the continuous model in which box type i has a constant reward rate, Ø$_i$, and may be searched for a maximum time t_i. The optimum policies for the two models are identical, although the payoff is less for the continuous extension, because of the possibility of finding an object before a whole box is searched.

3.8.1 Linear Precedence Constraints

The case of linear precedence constraints dealt with in Subsection 3.1.1 has an immediate graphical interpretation. Consider the two boxes shown below. The gradients of the lines de and ef are $\emptyset_{(i,1)}$ and $\emptyset_{(i,2)}$ respectively, so iff $\emptyset_{(i,2)} > \emptyset_{(i,1)}$, then point e lies strictly inside the convex hull of d, e and f, in which case nodes $(i, 1)$ and $(i, 2)$ form a single indivisible block.

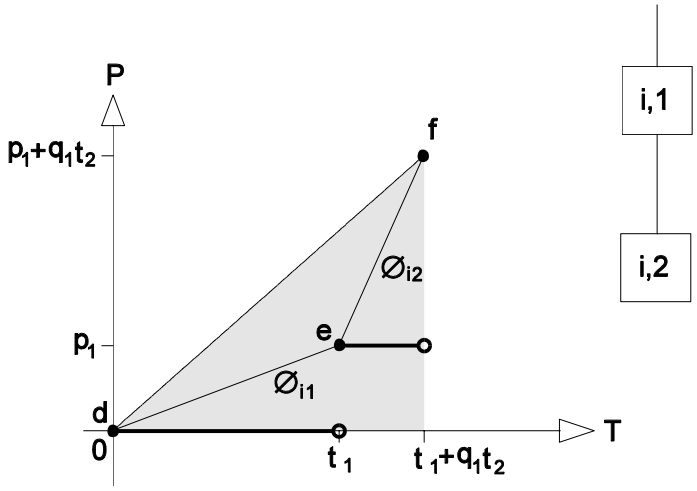


Figure 22: A Maximal Indivisible Block in the Continuous Case

By the same token, suppose further that stack i is charted in a similar fashion. If the points d and f lie on the convex hull of all the points in the stack, and this has different left- and right-gradients at these points, then the indivisible block def is maximal.

3.8.2 Concurrent Searching

We now consider how the continuous case might usefully be widened to deal appropriately with non-constant reward functions. Firstly, we note that if a box has a reward rate which is some non-decreasing function, $\Psi(t)$, of the time spent searching that box, we can consider instead $\Phi(t)$, the convex hull of the function $\Psi(t)$, for the same reason that boxes may be chunked into maximal indivisible blocks in the discrete case with linear precedence constraints. If the new rate of a node is a strictly decreasing function of t, we may wish to search that box for a vanishingly small period of time before changing to another box, which is a theoretical annoyance. Accordingly, we allow concurrent searching of boxes. Any number of boxes may be searched, with varying intensities, $i_1, \ldots i_n$ subject to the restriction that $\sum_j i_j = 1$.

As an example of how this simplifies the description of the optimal policy, suppose that two boxes are available for search, with decreasing reward rate functions $\Phi(t)$ and $\Phi(Kt)$ respectively. The optimal policy in this case is to search the boxes with respective intensities $i_1 = K/(K+1)$ and $i_2 = 1/(K+1)$, since this ensures that the reward rates of the two node types remain equal as search progresses.

3.9 Shape of $V()$

We now consider how $V(\mathbf{x})$ varies in x_i, by examining $\Delta V(\mathbf{x})_i$, defined as the increase in the payoff from adding a node of type i:

$$\Delta V(\mathbf{x})_i = V(x_1, \ldots x_{i-1}, x_i + 1, x_{i+1} \ldots x_n) - V(\mathbf{x})$$

The addition of an extra node of type i has two counteracting influences on $V()$:

1. *Decreasing* V: The extra node or its descendants may contain an object which can be relatively quickly found.
2. *Increasing* V: The extra node and its descendants must be searched before it is possible to terminate and conclude no object is present.

We now look at the net effect of these two influences. Observe that as $x_i \to \infty$, the probability that the existing supply of type i nodes would be exhausted tends to zero, and so both effects tend to zero. Hence $\Delta V(\mathbf{x})_i \to 0$ as $x_i \to \infty$.

Let us assume for notational convenience that the node types are indexed in order of decreasing $\emptyset^*$, so node type n minimises $\emptyset^*$. It is therefore possible to deduce the following value for $\Delta V(\mathbf{x})_n$ since we know that it is optimal to search the extra node last of all.

$$\begin{aligned} V(x_1, \ldots x_{n-1}, x_n + 1) &= V(\mathbf{x}) + q(\mathbf{x}) V(0, 0 \ldots 0, 1) \\ &= V(\mathbf{x}) + q(\mathbf{x}) t_n^* \end{aligned}$$

Hence : $\Delta V(\mathbf{x})_i = q(\mathbf{x})t_n^* > 0$

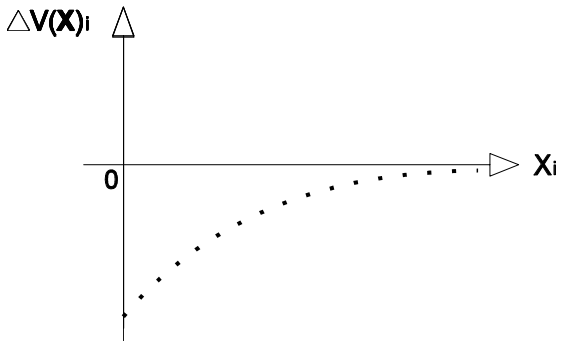


Figure 23: Shape of $\Delta V(\mathbf{x})_i$ for a Least Rewarding Node Type

As shown above, a least rewarding node type is *always* a liability. This is not true for any node types with an adjusted reward rate strictly greater than that of $\emptyset_n^*$. To see this, consider $\mathbf{x} = (0, 0 \ldots 0, N)$, for large N. As $N \to \infty$, the probability of ever terminating without finding an object becomes vanishingly small, and so does the second effect of adding a node of type i. The first effect, however, does not, and is non-zero since we required type i to have a reward rate strictly greater than $\emptyset_n$.

To see that $\Delta V(\mathbf{x})_i$ may have the shape as shown in Figure 24 overleaf, consider adding boxes of type 1 to a state $\mathbf{x}$, with $V(\mathbf{x}) > (\emptyset_{\mathbf{1}}^*)^{-\mathbf{1}}$. The optimal policy is to search the newly revealed boxes first, since they are of type 1. Thus:

$$V(x_1 + 1, x_2 \ldots x_n) = t_1^* + q_1^* V(\mathbf{x})$$

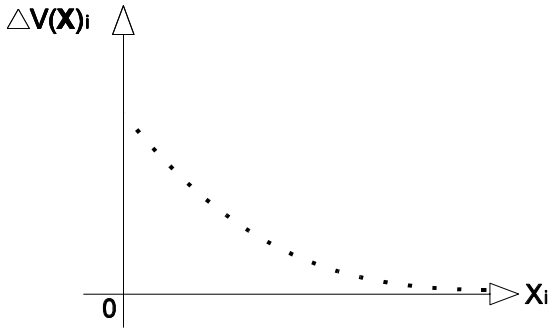


Figure 24: One Possible Shape of $\Delta V(\mathbf{x})_i$

$$\Delta V(\mathbf{x})_1 = V_1^* - p_1^* V(\mathbf{x}) = t_1^*(1 - \emptyset_1^* V(\mathbf{x}))$$

The shape of $\Delta V(\mathbf{x})_i$ may also have a turning point, as shown in Figure 25 below. Consider for example $\mathbf{x} = \mathbf{0}$. For any node type i, $\Delta V(\mathbf{0})_i < 0$, since $V()$ is minimised at $\mathbf{0}$.

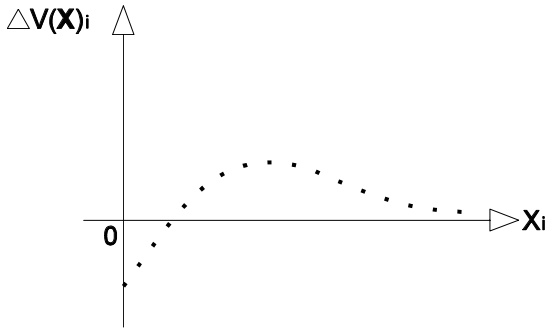


Figure 25: Second Possible Shape of $\Delta V(\mathbf{x})_i$

3.10 Summary

We have extended the branching bandits model of Weiss[88] in a number of ways. The equivalence between the semi-Markov continuous discounting case and the discrete case with transition dependent discounting highlighted in Subsection 3.5.4 could be used to derive a simple proof of the Gittins index theorem for discrete-time multi-armed bandits. The resulting proof would have much in common with Tsitsiklis' [85] proof for semi-Markov bandits.

The formulation of Ø as a reward rate and the discussion of overlook probabilities makes clear that in certain circumstances, the case of a (countably) infinite number of node types is tractable.

The non-constant 'retirement function' introduced in Section 3.6 is a powerful innovation, since it allows for its application to practical situations in which it is desirable to terminate search and make a decision based on its findings before an exact result is known. Specification of a retirement function allows for more powerful control of the search. As an illustration of its potential in the context of computer search, we note that it allows for a single search to be efficiently conducted in parallel, by allowing dynamic allocation and re-allocation of subsearches in accordance with findings.

Heuristic evaluations of nodes are commonly used only to determine which is the best of a given set of positions. This frequent under-utilisation of the information calculated means that the adoption of such an apparently simple evaluation function as that presented in Subsection 3.4.2 need not be as restrictive as at first appears. A standard evaluation function $H : x \mapsto R$

might, for the purposes of search control be replaced by $I_{H(x)>H(root)}$. This would separate out the positions into two classes, those which were an improvement on the current position and those which were the same or worse, which would suffice for some purposes.

An ability to make rational decisions about whether to terminate the search early seems likely to broaden the applicability of the model very considerably, since there are many situations where the goal of search is not to find an object but merely to discern as quickly as possible whether an object exists. Suppose, for example that a batch of goods has been manufactured. It is clearly of great value to have a procedure to automatically calculate whether the probability of eventual success is sufficient to warrant continuation of a series of quality control tests.

As currently described, the model is limited to graphs with an out-tree (or out-forest) structure. An obvious question is whether search of more general DAG's can be modelled in a similar fashion. The main problem with such an extension seems to be that the model is based upon a state which is a vector of scalars of fixed length. A more general DAG structure causes difficulties with this representation since it becomes necessary to keep track of previously searched branches.

4 AND-OR Tree Search

To grasp the notion of an 'AND-OR tree' we recall how the 'OR-tree' search problem of the previous chapter can be envisaged as an investigation into the truth of a logical expression. In this chapter we address a more general problem than that of the previous one, by redefining the class of logical expressions to be those which include L iff it satisfies one of the following:

1. L is a logical primitive.
2. $L \equiv (X \cup Y)$, where X and Y are both logical expressions.
3. $L \equiv (X \cap Y)$, where X and Y are both logical expressions.
4. $L \equiv X^{\mathrm{c}}$, where X is a logical expression.

The symbol '$\cap$' represents the customary binary Boolean 'AND' operator, and '$^{\mathrm{c}}$' the unary Boolean 'NOT' operator, both defined as usual. Any logical expression L can be written as a tree, with the internal nodes as '$^{\mathrm{c}}$', '$\cap$' and '$\cup$' operators, and the leaves as logical primitives. There are a variety of normal forms for representing logical expressions as defined above, of which I have found one particularly clear, because it emphasises the similarity with the OR-Tree model. It uses the following equivalence:

$$X \ \cap \ Y \equiv (X^{\mathrm{c}} \ \cup \ Y^{\mathrm{c}})^{\mathrm{c}}$$

We shall also exploit the associative property of the '$\cup$' operator, and so all the logical expressions in this chapter will appear in a form without

'$\cap$' operators, and with alternate levels of '$\cup$' and '$^{\mathrm{c}}$' operators. For example, the logical expression $(A \cap B^{\mathrm{c}}) \cap (C \cup D)$ would be transformed into $(A^{\mathrm{c}} \cup B \cup (C \cup D)^{\mathrm{c}})^{\mathrm{c}}$, represented in the below figure.

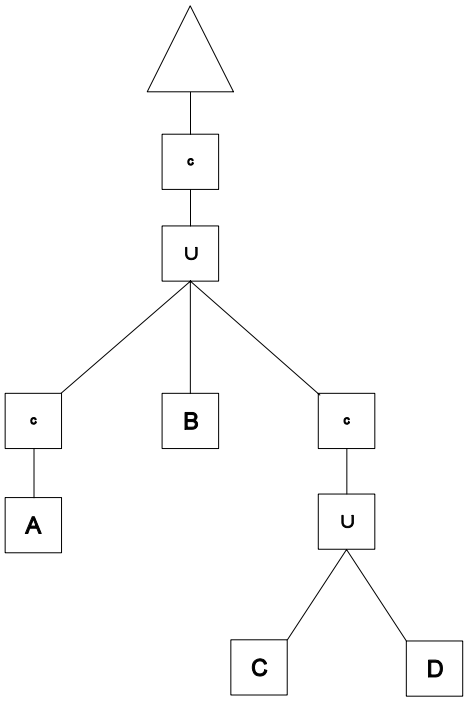


Figure 26: Sample Representation of $(A^{\mathrm{c}} \cup B \cup (C \cup D)^{\mathrm{c}})^{\mathrm{c}}$

Whilst it is well known that any such logical expressions can be transformed to one which has only the single operator 'NAND', this representation is not useful for our purposes. This is because the economy of operators is achieved by duplicating the logical primitives (e. g. $A^{\mathrm{c}} \equiv (A \text{ NAND } A)$, which brings more serious problems of its own[17].

[17]Since the two A's in the above example correspond to a single node, they share a single expansion. This representation would therefore enforce a dependence structure

4.1 Deterministic Case

By way of introduction we first consider the case in which the logical primitives are boxes. This problem was first addressed by Joyce [35]. The optimal policy may be efficiently deduced in time proportional to the overall length of the logical expression by working backwards from the tree which represents it. The leaves are simple logical primitives. Iterative application of the two steps below suffices to transform an arbitrary logical expression into an OR-tree of depth one which can then be solved in the fashion described in the Section 3.1:

1. *Sibling chunking:* This applies to logical expressions of the form $L = l_1 \cup l_2 \cup \ldots \cup l_N$, where each l_i is a simple logical primitive. This is an example of the boxes case of Section 3.1, so the optimal policy is to search the l_i in decreasing order of $\emptyset_i$. Assuming the node types are ordered by $\emptyset$, so that $\emptyset_i \geq \emptyset_j$ for $i < j$, we have the following formulae for t_L, the expected time taken to determine the truth of L, and for p_L, the probability that L is true:

$$t_L = \sum_{i=1}^{N} t_i \prod_{j=1}^{i-1} q_j \qquad\qquad p_L = 1 - \prod_{i=1}^{N} q_i$$

between the logical primitives which violates the fundamental assumption, that each logical primitive is a single search opportunity which may be explored independently of all the others.

Thus, the compound expression, $l_1 \cup l_2 \cup \ldots \cup l_N$ can be represented by a new simple logical primitive, L, with details $d_L = (t_L, p_L)$.

2. *'Complement' Removal:* This applies to logical expressions of the form L^{c}, where L is a simple logical primitive. We replace L^{c} by a new simple logical primitive, $\bar{L}$, with $t_{\bar{\mathrm{L}}} = t_L$, $p_{\bar{\mathrm{L}}} = 1 - p_L$.

4.2 Stochastic Case

We now increase the scope of the model to cater for a more general class of logical primitives. We assume that, upon search, each logical primitive may expand to any logical expression, according to a known distribution. This requires an extension of the notation used for the OR-tree model, since the space of logical expressions which include '$\cup$' as well as '$^{\mathrm{c}}$' cannot be summarised in the same fashion by a finite length vector. As an introduction to this extra notation, consider the node of type A, defined below:

$$d_A \;=\; (\frac{1}{3}, 9, \frac{1}{2} + \frac{1}{4}s_A + \frac{1}{8}s_B^2 + \frac{1}{8}s_A s_B^2)$$

We represent this as follows:

$$A[9] = \begin{pmatrix} \langle \mathrm{T} \rangle & \langle \mathrm{F} \rangle & A & B_1 \cup B_2 & A \cup B_1 \cup B_2 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & \frac{1}{12} & \frac{1}{12} \end{pmatrix}$$

The special elements, $\langle \mathrm{T} \rangle$ and $\langle \mathrm{F} \rangle$ correspond to 'true' (an object is found), and 'false' (no object is found, and there are no descendants). We

shall use this form of notation wherever necessary — that is, wherever a search may give rise to an expression with a '^{c}' or '$\cap$' constructor.

As defined above, the model encompasses some problems for which the notion of 'optimal policy' is ill-defined. Consider, for example, the node type below:

$$A[1] = \begin{pmatrix} \langle \mathrm{T} \rangle & \langle \mathrm{F} \rangle & (A_1 \cap A_2) \cup (A_3 \cap A_4) \\ \epsilon & \epsilon & 1 - 2\epsilon \end{pmatrix}$$

For $\epsilon = 0$, the truth of A is obviously undecidable, since it will never expand to $\langle \mathrm{T} \rangle$ of $\langle \mathrm{F} \rangle$. For small enough positive ϵ, there is non-zero probability of the expression being undecidable, as can be understood from thinking of the search of A as a branching process. Such expressions, where the expected number of searches of the optimal policy is infinite, we term *intractable*, and do not address further.

Another class of expressions *require* only a finite number of searches, but may nevertheless be searched *ad infinitum*. As an illustration, consider the degenerate node type X, defined below:

$$X = (A \cup B) \qquad A[1] = \begin{pmatrix} A \\ 1 \end{pmatrix} \qquad B[1] = \begin{pmatrix} \langle \mathrm{T} \rangle \\ 1 \end{pmatrix}$$

4.3 Nature of the Optimal Policy

Fundamental to the simplicity of the stochastic OR-tree search model is the fact that results from one part of the search tree have no influence upon how it is optimal to explore other parts of the tree. This might seem to be a simple consequence of the independence of each individual offspring distribution, but this is a necessary, not a sufficient condition for this property. In addition to this there must also be a simplicity of structure of the search model. This is satisfied in the stochastic OR-tree search model, since it is possible to deduce whether or not the goal has been met by looking at each leaf in isolation. (If an object has been discovered at any of them, it has!)

This simplicity of structure means that each part of the tree can be considered in isolation, and implies that there is an optimal policy within the class of pre-empt/resume policies. This class contains policies that dynamically choose *which* parts of the tree to search, but not *how* to search them. An obvious first question when addressing the general stochastic AND-OR tree search problem is whether there is a pre-empt/resume policy which is optimal.

Figure 27 overleaf shows two different search problems which include the sub-search of the expression Y, illustrating why a pre-empt/resume policy is not necessarily optimal. Suppose that there are two alternative policies for searching Y. Policy π_1 allows for a quick and relatively accurate judgement to be made about whether or not Y is true. Policy π_2 by contrast does not, but determines the truth in less time, on average.

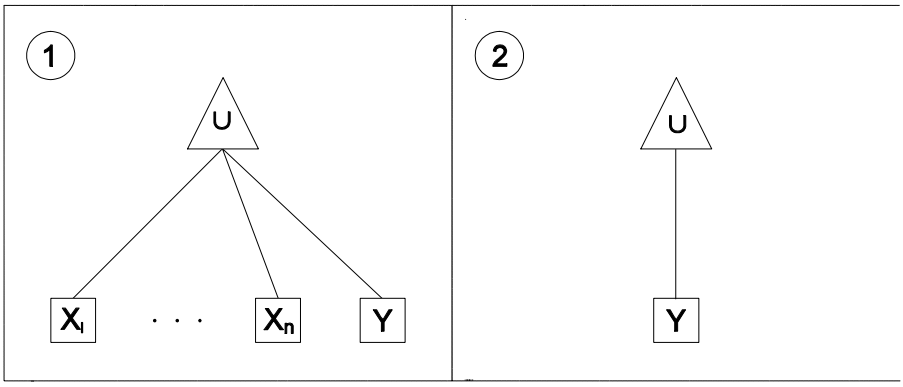


Figure 27: Different Requirements for Searching Y

Policy π_2 is to be preferred in case (2), but not necessarily in case (1). The reason for this is that the 'intermediate information' associated with policy π_1 is of use in case (1), since it may show that switching to search one of the X_i expressions is preferable to further search of Y. In this way, the expected time to search the overall expression can be reduced.

An example of an expression Y for which two such search policies exist is as follows:

$$Y = (A \cup B) \quad X = (Y_1^{\mathrm{c}} \cup Y_2^{\mathrm{c}} \cup \ldots Y_N^{\mathrm{c}})$$

The following node types are defined:

$$d_A = \left(0, 1, \frac{1}{2}s_B + \frac{1}{2}s_C\right) \quad d_B = \left(\frac{99}{100}, 10, 1\right) \quad d_C = \left(\frac{1}{2}, 10, 1\right)$$

We must search one of the identical Y_i, so for definiteness assume we search Y_1. The decision to be made is therefore whether we search A_1 or B_1. A little reflection on the problem will suffice to show that the optimal choice

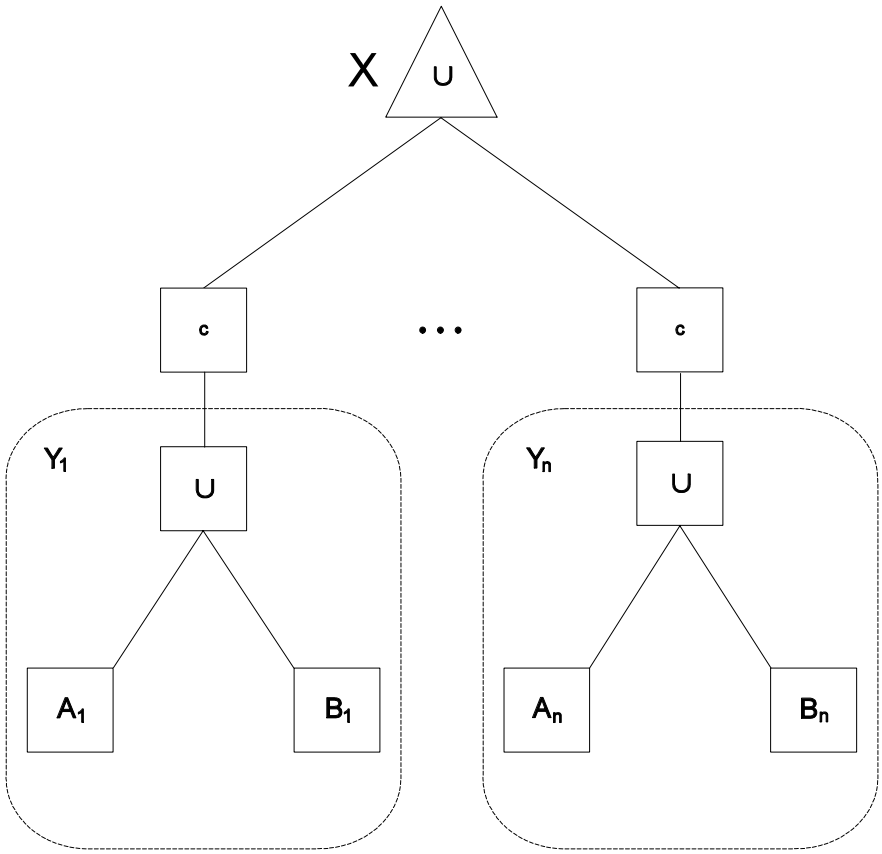


Figure 28: Pre-empt/Resume Counterexample

depends upon the value of N, thus violating non-locality and implying that optimal play cannot in general be a pre-empt/resume policy.

As $N \to \infty$, the probability that X is true tends to 1, and the value of refuting a single Y_i^{c} tends to zero. The aim of search must therefore be to prove a Y_i^{c} to be true, thus proving X to be true and allowing termination of the search. This is equivalent to refuting a Y_i. Bearing this in mind, it is better to start search of a Y_i by an exploratory search of A_i. If this shows the Y_i to be equivalent to $B_{i_1} \cup B_{i_2}$, it is optimal to 'retire' from this Y_i and choose another, until a Y_i is discovered that is equivalent to $B_i \cup C_i$, an

expression which is 50 times more likely to be false, and so more worthy of investigation than $B_{i_1} \cup B_{i_2}$.

Now consider the opposite extreme, $N = 1$. In this case, a refutation of Y_1^{c} will terminate the search, showing X to be false. This can be achieved with probability $\frac{99}{100}$ by search of B_1. Search of A_1, by contrast, offers no immediate chance of terminating the search, and, whatever node it reveals, the best node to search next will be B_1.

To recap, we find that there is no single answer to the question "How is it optimal to search the sub-expression Y_1?"; if it has no siblings, it is optimal to begin by searching B_1, while if it has many siblings, it is optimal to begin by searching A_1.

4.4 Non-optimality of Index Policies

The counterexample described in the previous section demonstrates that the AND-OR tree search problem cannot be solved by an index in the same way that the OR-tree search problem can be solved by calculating $\emptyset()$. We now examine the possibility of treating an AND-OR tree as an OR-tree of depth one with an infinite number of node types.

We see that an AND-OR tree can be written $(Y_1 \cup Y_2 \ldots Y_n)^{\mathrm{c}}$, as shown in Figure 29. The presence or absence of a '$^{\mathrm{c}}$' makes no difference to the fundamental conception, which is to extend the result of the Chapter 3 by dealing with each of the Y_i's as if it were a simple logical primitive as opposed to a compound logical expression. In general, this requires that there be a

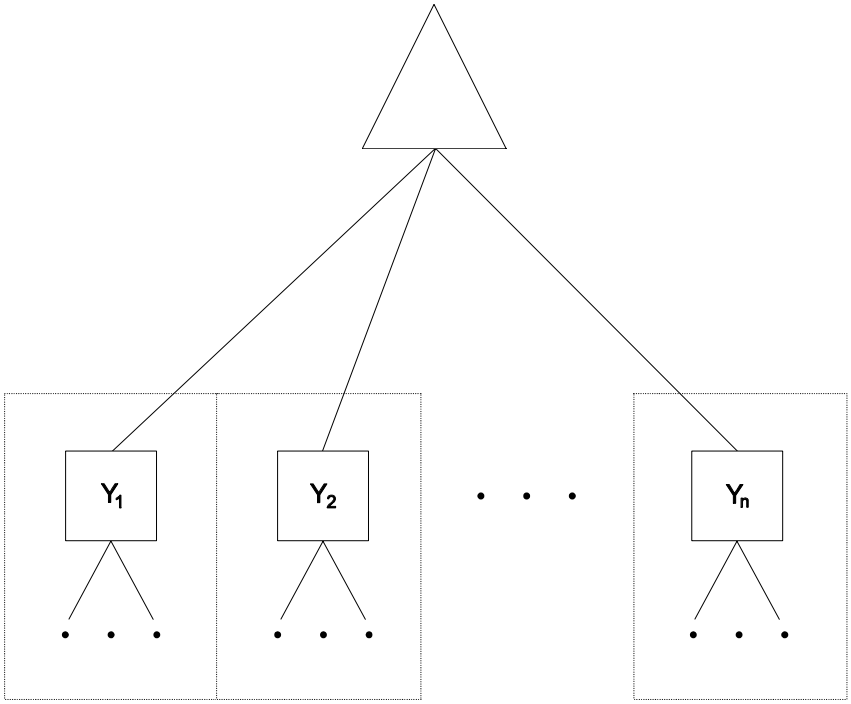


Figure 29: Viewing an AND-OR Tree as an OR Tree

countably infinite number of node types, since each Y_i may be an arbitrarily complex AND-OR expression. Even in the special case in which the Y_i cannot become arbitrarily complicated, but can be transformed by search into one of only a finite number of terms, we show by considering the preempt/resume counterexample of the previous section that this concept still presents problems.

To represent X as an OR expression, we need a state for each logical expression to which search of the Y_i can lead. For clarity, we index these

states numerically:

$$\begin{aligned} s_1 &= C^{\mathrm{c}} & \qquad s_4 &= (B \cup C)^{\mathrm{c}} \\ s_2 &= B^{\mathrm{c}} & \qquad s_5 &= (B_1 \cup B_2)^{\mathrm{c}} \\ s_3 &= A^{\mathrm{c}} & \qquad s_6 &= (A \cup B)^{\mathrm{c}} \end{aligned}$$

The expression Y therefore corresponds to a node of type 6, so state $(0, 0, 0, 0, 0, 1)$, whilst X corresponds to state $(0, 0, 0, 0, 0, n)$. The node types have the following characteristics:

$$\begin{aligned} d_1 &= \left(\tfrac{1}{2}, 10, 1\right) \\ d_2 &= \left(\tfrac{1}{100}, 10, 1\right) \\ d_3 &= \left(0, 1, \tfrac{1}{2}s_1 + \tfrac{1}{2}s_2\right) \\ d_4 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_1\right) \text{ or } \left(0, 1, \tfrac{1}{2}s_5 + \tfrac{1}{2}s_6\right) \\ d_5 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_2\right) \\ d_6 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_3\right) \text{ or } \left(0, 10, \tfrac{1}{2} + \tfrac{1}{2}s_2\right) \end{aligned}$$

This problem is not yet in a form soluble by the OR-tree search model because of node types 4 and 6, which may be expanded in more than one way. In practice, since this example is small, dynamic programming can be used to deduce the optimal policy. Whilst a general solution seems an unlikely prospect, use of points made in Section 3.9 may prove sufficient to allow an ad hoc solution by dynamic programming to some specific classes of problems.

4.5 Summary

The AND-OR model described above is a powerful one, capable of application without modification to the task of (bi-valued) game tree searching. Unfortunately, it does not seem to be an easy task to deduce an optimal policy.

An important observation about the AND-OR tree search case is that the optimum policy is not a pre-empt/resume policy. The non-locality of the optimum policy, together with the observation that no one has yet come close to solving this problem strongly suggest to me that aiming for a method of deducing a strictly optimal policy may, in the general case, be an unrealistic goal. A more fruitful approach may therefore be to pursue methods of deducing a policy which is both easily computable and 'nearly optimal', in some sense.

A final, important, point which deserves to be made is a consequence of the fact that knowledge of whether or not an object exists affects the optimal policy. *Given that an object exists*, a simple Ø-based policy can be used, not to direct all the search, but simply to choose which top level branch it is optimal to search. An admittedly speculative example of how this might be used is the assumption that might be made by a game-playing program that somewhere in the game tree it is searching, a winning variation does indeed exist.

5 Time Control

The difference between a tree search procedure and a game-playing algorithm is an important and underestimated one. The latter will include a tree search procedure, but something more is required — it must also have a method for terminating searches at some point. Basically, a tree search procedure looks upon a single move as the whole problem, whilst a game-playing algorithm looks upon the problem as that of managing a series of moves, with concomitant searches, and makes rational decisions about how to share any transferable resources (i.e. time[18]) between them. The role of time control may therefore be summarised below:

$$\boxed{\text{Game Playing} = \text{Time Control} + \text{Tree Search}}$$

While tree search algorithms have been the subject of much research, very little has been published in the artificial intelligence literature on time control. This paucity of literature demands an explanation, particularly when one considers how poorly the standard 'flat rate' approach models that of human experts. The method of controlling time which still provides the basis for many otherwise refined game-playing programs is to apportion the search time between the moves on a simple pro-rata basis, so that the same amount of search is spent on every move. This is a very crude method of resource allocation, most obviously in cases where the player has only one legal move

[18] Also *information*, but analysis of this issue is beyond the scope of this text.

at his disposal[19]. Any human gameplayer will confirm that some moves are more important than others, in the sense that a player with a fixed time limit for the whole game will naturally wish to consider for longer before making them.

It is almost tautological to state that a player spends the longest on the positions he finds the most difficult, which are the ones in which it is hardest for him to determine which move it is best for him to play. Nevertheless, this statement contains the key to the issue of time control, and provides at least a partial explanation as to why so little work has been done on formal methods of time control:- *to be able to apportion time in an efficient fashion, an 'understanding' of the consequences is essential.*

By their very nature, brute force methods — which were until very recently ubiquitous amongst the top game-playing programs — do not have any such understanding. Subsection 1.1.4 introduced the method of singularity extensions as a refinement to alpha-beta search. It is also possible to view it as a time control algorithm, since it increases the amount of time spent searching variations which involve singular moves. This hints at another important observation about game-playing algorithms:- *the distinction between a search algorithm and a time control algorithm is to some extent an artificial one.* Any human game player knows this, since no one first decides in advance of thinking about a move how much time to spend doing so. If the

[19]Of course, in practice programs have an *if* statement to detect for this, but the very existence of such a special case illustrates the ad hoc nature of this approach.

human thinking process is to be mimicked, then the time allocation process must be responsive to the findings of the search algorithm, as depicted below.

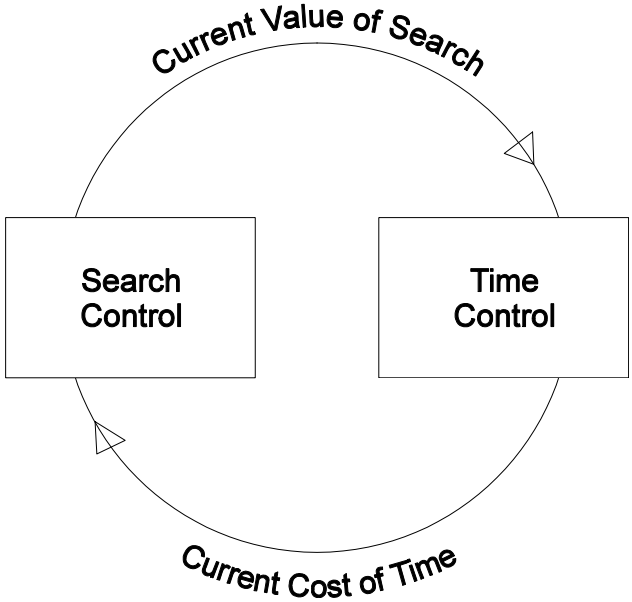


Figure 30: The Relationship between Time Control and Search Control

The reason why most examples in the literature simply have a fixed budget or some other very simple policy is that the 'brute force' nature of their alpha-beta-based search algorithms offers so little understanding of the position. It is worth formalising the justification for this time control policy; uniform time control assumes that the utility to be gained from searching is the same for every move throughout a game. The quality of this approximation depends largely upon the game.

5.1 Previous Methods

To pursue a 'flat rate' time allocation policy is to ignore the information about the tree that is gathered by the searching process. Information of this kind is readily available to a selective search since it is required by the selective nature of the algorithm. This suggests the general form of time control appropriate to selective search. Since the selectivity of the search algorithm depends upon some information being accrued about the tree as search progresses, search can be terminated as soon as the information built up meets some criterion. The nature of this criterion depends of course upon the time allowance, the game being played and the specifics of the selective search algorithm. Flaws in the conception of a search algorithm will lead to corresponding failures of the time control algorithm, since the time control algorithm can only access information gathered by the search algorithm.

For example, consider the paradigm behind the B* family of search algorithms described in Subsections 1.2.3–1.2.5. Originally, B* search was suggested as a method of searching adversary trees which had accurate bounds on their scores. Accordingly, no time control was required; the algorithm terminated as soon as it had found *the answer*. As part of a game-playing algorithm, problems arise with the initially appealing goal of finding the best move. In a real game, some moves are more important than others, a fact which is not recognised by B* or any of its probabilistic variants. Indeed, the notion of a unique 'best' move may be flawed. It is a frequent occurrence in the game of Go, for example, that a pair of moves have *exactly* the same

game theoretic value. The general problem with use of B* in a game-playing problem is illustrated overleaf in Figure 31, which shows four types of uncertainty about the best available move, of which the top left is the most worthy of further search. Since the B* paradigm only looks at the probability that one move dominates the others, and ignores the amount, it does not take into account the variance of the move estimates. Since it therefore fails to recognise that the left hand cases are better candidates for further search, it will not be a good basis for a time allocation policy.

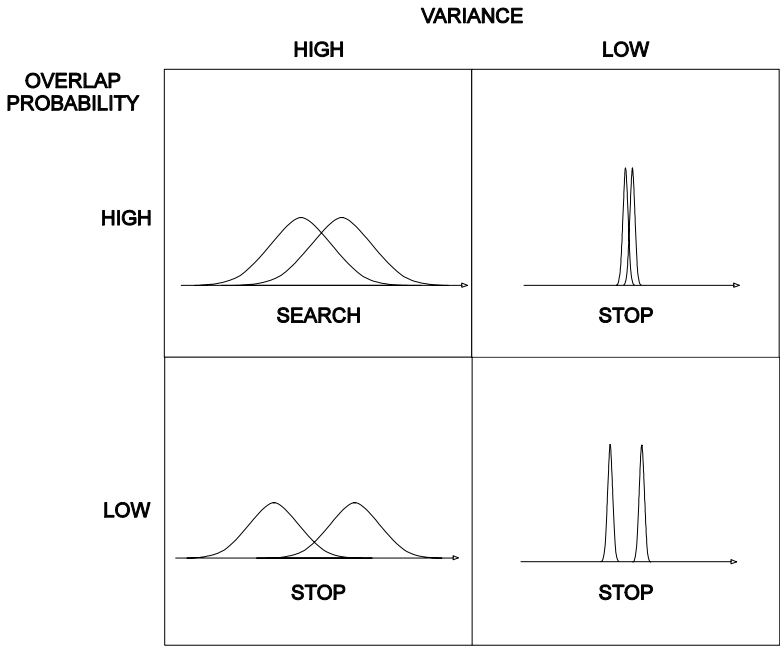


Figure 31: The Problem with the B* Paradigm

We now examine the range of approaches to time control applied to the more important selective search algorithms, in reverse order of sophistication.

5.1.1 Best First Minimax

In the theoretical description of the best first minimax algorithm, Korf and Chickering [47] make the bald statement that

> "While in principle we could make a move at any time, we chose to make a move when the length of the principal variation exceeds a given bound. This ensures that the chosen move has been explored to some depth."

Since this is the sum total of their remarks on time control in their substantial paper, one is forced to assume that they have paid little attention to this topic. One possible explanation for this is to be found in their conclusions, in which they state that "in practice, memory is not a problem because in a two-player game, a move must be made every few minutes". The indication here is that practical rather than theoretical considerations may have determined the nature of the time control policy; as well as being in keeping with the spirit of the best first minimax algorithm — neat and simple — the setting of such a depth threshold is expedient in that it avoids practical problems caused by the space required to store the tree in memory. Where such a compromise is being made to expediency it should be admitted, and the time control policy does not seem to have any theoretical justification.

5.1.2 Conspiracy Numbers

In his original paper on the subject, McAllester [50] describes the method of conspiracy numbers not as a game-playing algorithm, but as a 'procedure ... for growing min-max game trees' and so does not address time control. The results he obtained were achieved by terminating the search after a fixed number of nodes.

Schaeffer [73] is credited with adding the notion of iterative deepening to conspiracy number search. The usage of the term in this context is an analogy with its use in the context of alpha-beta search; instead of increasing the fixed depth of the alpha-beta search by one each time, the size of the conspiracies searched for is increased by one. The most promising results of his paper concern tactical Chess problems of the 'Black to play and win' type. These however are single searches and so do not require any means of time allocation. Schaeffer acknowledges that time control remains a difficult issue.

A natural, if rather straightforward, approach to time control would be to impose a ceiling on the maximum conspiracy size that is searched for. This would have the desired effect of spending less time on positions where the best move was relatively clear and more where it was unclear. This is, however, not merely game specific, but also implementation specific; the 'granularity'[20] of the evaluation function is very important in determining

[20] the precision with which positions are evaluated

the conspiracy thresholds reached by the search algorithm.

The extent of this problem is revealed by the application of Schaeffer's conspiracy numbers algorithm to a set of Chess positions. The granularity of the position evaluation function was $\frac{1}{10}$ of a pawn. For some positions, the algorithm was still hunting for conspiracies of size 2 after an hour of C.P.U. time! Naive use of the conspiracy threshold is therefore not an appropriate method of time control. A 'flat rate' time allocation policy is particularly wasteful for a selective search algorithm, and so the current lack of any feasible alternative must represent a serious drawback of conspiracy number search.

5.1.3 PSVB*

The original paper on the B* search algorithm, like that on conspiracy number search, did not contain any discussion of time control. For his PSVB* search, Palay [62] was therefore forced to deduce an appropriate time control mechanism from first principles. He drew his inspiration for the task from a major strength of B*, namely, the ability to terminate search once it becomes clear that one move is 'better' than the others, even if the score of that move is not certain.

The 'better than' criterion of B* is replaced in PSVB* by a rather dubious[21]

[21]Baum and Smith[6] highlight a weakness of this criterion by giving an example of distributions A, B and C, such that A dominates B,B dominates C,and C dominates A, all with probability $> .5$.

'dominates with probability p' criterion. Another failing is that Palay's method owes too much to the standard 'flat rate' approach to time control, since he assumes that each move is allocated (presumably uniformly) a 'time limit'. His basic philosophy is summarised below:

> "While there is plenty of time remaining, the search should be continued, unless it is certain that one move is better than the remaining moves. As the amount of time used by the search increases the levels of domination used should start to decrease. At first the decrease should be gradual; however, as the amount of time used by the search approaches the amount of time allocated to the search, the levels of dominance should decrease rapidly."

He also recognised that the need for further search is influenced by the relative winning chances of the two players. Again, whilst the idea is not without merit, the choice of implementation is tailored specifically for the game of Chess.

> "The choice of the actual function was somewhat arbitrary. The guiding principle in selecting this function was that as the achieved value of a move increases, the level of domination needed for termination decreases."

Palay's approach to time control was very innovative at the time at which the work was carried out, and he was one of the first authors to pay specific

attention to this topic. However, the fact that he never worked out how to play from a position in which one was almost hopelessly behind merely underlines the piecemeal nature of the approach he took to the issue of time control.

5.1.4 MGSS*

The time control algorithm of MGSS* is based on the following algorithm proposed in 1968 by Good [30]:

1. *If no leaves have positive* $U()$*, stop and make the move which appears best on the basis of the current search tree, otherwise proceed to step* 2.
2. *Search the leaf with the greatest* $U()$*, recalculate the* $U()$ *values as required, and go to step* 1.

The rationale for such a schema is clear, since *if the meta-calculation costs are ignored* and *given a suitable definition of* $U()$, this time control algorithm is indeed optimal. However, as we have seen, neither of these is possible, and so this point is of theoretical rather than practical interest. Indeed, the problem of determining a tractable approximation of $U()$ proved such a difficult one that this method was all but ignored by the mainstream of artificial intelligence research for around twenty years. The efficacy of such a policy hinges upon the utility function, a topic which we shall address in

Section 6.5. The MGSS* algorithm uses $U(S) = E[V(S)] - TC(S)$, where $V(S)$, the value of search, is defined in Subsection 1.2.6.

Russell and Wefald [72] are uncharacteristically vague about the choice of $TC()$ used to implement MGSS*. The simplest choice would be to use a constant function. For games in which different numbers of moves are available from different positions, a more accurate $TC()$ would — since the units of computation are the expansion of a single node — be proportional to the number of children of that node.

5.1.5 BP

The time control policy of the BP algorithm is analogous to that of MGSS*. Since BP expands leaves one gulp at a time rather than one leaf at a time, their expression for the net utility of further search is as follows:

$$\frac{U_{\text{gulp}}}{t_{\text{gulp}}} \; - \; TC(t, m)$$

In this expression, t_{gulp} is the amount of time needed to carry out the next gulp of search, while U_{gulp} is the expected increase in utility from doing so, as explained in Subsection 1.2.7. The chief advance over the time control of MGSS* is the use of a more advanced expression for the cost of time, $TC()$. This takes parameters t, the amount of time left, and m, the number of moves until the next time control. This makes the assumption that the number of moves until the next time control is known, which is not valid if there is an overall limit for the game unless the game is a fixed number of

moves. The following two expressions are suggested for $TC()$:

$$TC() = c_6 \frac{m}{t} U\left(log_B \left[\frac{t}{m}\right]\right) = \begin{cases} c_3 \frac{m/t}{\log_B (t/m)^2} & (Levy) \\ c_4 \left(\frac{m}{t}\right)^1 + c_5 & (Szabo) \end{cases}$$

The B in the Levy formula is the expected branching factor of the search. The two formulae are derived from suggested expressions for the value of time by Levy and by Szabo [84]. Baum and Smith [6] explain their efforts to fit these two formulae — by judicious choice of c. The data from which they derive their observed utility function, $U()$, is a table of Newborn's [58] of how likely a Chess program is to change its opinion of the best move if deeper search is carried out.

5.2 Marginal Value of Information

The appeal to some kind of 'law of diminishing returns' on time spent searching is a very convincing one, following almost as a corollary of the exponential explosion of nodes in the game tree. The exact nature of this principle remains obstinately difficult to codify in general terms.

The 'boxes' problem of Section 3.1 is one in which the principle can be seen to yield the optimal policy in a very straightforward way. Box i has reward rate $\emptyset_i = p_i/t_i$, so the probability of finding an object is $\emptyset_i t_i$, yielding an expected probability of termination proportional to $\emptyset_i$ for every time unit spent searching a box of type i. The optimal policy of searching boxes in

decreasing order of type $\emptyset_i$ equates to a law of diminishing returns over each single box searched.

The addition of linear precedence constraints brings a further level of complication. The $\emptyset_i$ of searches by the optimal policy are no longer monotonically decreasing, since searches are made not solely for their probability of discovering an object, but also to reveal other nodes for search. If we consider the units of the search to be maximal indivisible blocks rather than individual nodes, the law of diminishing returns still applies.

The stochastic OR-tree model of Section 3.2 is similar in principle. The chunking process, however, is affected by the random nature of the model. A law of diminishing expected returns applies, where the unit of search is a single exhaustive search. The *corrected* reward rates, $\emptyset_i^*$, take account of this.

Unfortunately, the AND-OR model that is the subject of Chapter 4 is harder to model in this fashion, due to the way the '$\cap$' branches affect the return from searching. An expression such as '$A \cap B$' presents a problem since the return from searching 'A' depends upon the results of searching 'B' and vice versa. As explained in Section 4.4, considering the composite term '$A \cap B$' as a single unit of search presents problems, not least because search may expand 'A' and 'B' into compound terms.

5.3 Marginal Value of Search

We now revise the two models presented in Chapter 2, so as to give the player a choice as to how much time he uses to search. Instead of having a fixed time limit, an unlimited supply of time units are available to the player, at a cost of $\lambda > 0$ each. This formulation has a natural interpretation in a practical context, and is mathematically easy since the time spent so far is a sunk cost, that is, one which does not affect future play. Accordingly, only very minor adaptations are necessary to the mechanics of the two models. The only extra notational device which we shall use in this section is the symbol $\Lambda(\tau)$, to stand for the marginal value of time, that is, the cost of time at which, if the player must purchase all the time units before the game[22], he is indifferent about purchase of a further unit of time once τ units have already been bought.

In the one-player tree search game of Section 2.2, Lemma 2.4 implies that for $\lambda \geq \kappa$, it is optimal not to buy any time, and for $\lambda = \kappa$, it is optimal to buy up to H time units. For $\lambda \leq \kappa$, it may be optimal to buy more, though this depends upon the values of Y observed in relation to the specific distribution of Y. If $\lambda = \kappa$ the only circumstances in which it is optimal to buy more search is if the last two Y values observed were both equal to 0, in which case it is optimal to 'backtrack' up the searched tree, as illustrated

[22]This restriction is introduced as a slight simplification of the presentation that follows, which is nevertheless sufficient to illustrate the principles involved. Without it, $\Lambda(\tau)$ is a random variable, since it depends upon the findings of earlier search.

on page 51 in Figure 8.

One extreme occurs if Y is distributed as follows:

$$Y \sim \begin{pmatrix} -\kappa & \kappa \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

In this case, only h searches are ever required to find the optimal path down the game tree, so $\Lambda(\tau) = 0$ for $\tau > h$. At the other extreme, if Y has support along the whole real line, $\Lambda(\tau)$ decreases in τ, only reaching 0 for $\tau = 2^h - 1$, at which point the tree is completely searched.

It is understood, although not formally proved, that $\Lambda(\tau)$ is a decreasing function in τ for this game. This seems clear from the way in which search time is exchanged for information. As the time already spent increases, the most profitable search opportunities have already been exhausted, so resulting in some 'duplication of effort'. If the variance of $|Y|$ is low, $M(\tau)$ decreases quickly from an initially high value, since the searches yield a lot of information. By contrast, if the variance of $|Y|$ is high, then $M(\tau)$ decreases more slowly because of the greater chance of revising previous decisions about which move was best from a particular node.

In Section 2.3, we saw how to calculate $V_n()$, the payoff of the fuel control problem with n units of fuel available. We can therefore calculate the marginal value of fuel exactly since $\Lambda_\tau() = V_{\tau+1}() - V_\tau()$. In contrast to the one-player tree search model, $\Lambda_\tau()$ need not be monotonic, a consequence of its more general structure. Consider the construction shown in Figure 32.

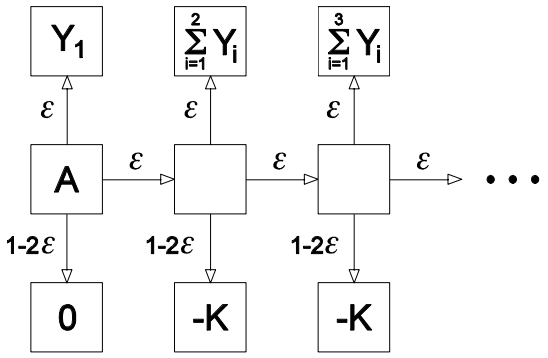


Figure 32: Construction to Show the Generality of $\Lambda_\tau(A)$

By choosing $\epsilon > 0$ to be small, and K to be sufficiently large, this shows that $\Lambda_\tau(A)$ can be any non-negative sequence, $Y_1, Y_2, Y_3 \ldots Y_N$.

5.4 A Two-Player Search Game

We consider the following two-player zero-sum game, similar to the one defined in Section 2.2. The game tree is an infinite binary tree. Players make alternate moves. Each move is associated with score, Y, an independent random variable of known distribution. The player whose move it is has the option of carrying out one or more search actions before moving. Each search makes known to that player the values of the two moves possible from the node searched. The set of nodes searched must at all times form a tree. This is almost the infinite height case of the familiar search structure from Chapter 2. This model differs from the model described in Section 2.2, since the Y values of sisters are independent of each other, rather than being constrained so that $Y_{i1} + Y_{i2} = 0$.

The running reward, R_τ, paid from player 2 to player 1 in time unit τ is

given below:

$$R_\tau = \begin{cases} -\lambda & | \text{ Player 1 carries out a search.} \\ \lambda & | \text{ Player 2 carries out a search.} \\ y & | \text{ Player 1 makes a move with value } y. \\ -y & | \text{ Player 2 makes a move with value } y. \end{cases}$$

It is useful to discount the running rewards by α^N, where N is the number of moves made so far and $\alpha \in (0, 1]$. As in Section 2.2 we specify that $E[Y] = 0$ and $E[|Y|] = \kappa$.

We now consider how to capture the value of search already carried out. If the moves in the search information which leads down to leaf, L, have scores $y_1, y_2 \ldots y_{h_i}$, then the *net value* of leaf L is defined as follows:

$$nv(L) = \sum_{i=1}^{h_i} (-\alpha)^{i-1} y_i$$

This is the net value to player 1 from playing the game until leaf L_i is reached.

We now define the net values of internal nodes of the search information as follows:

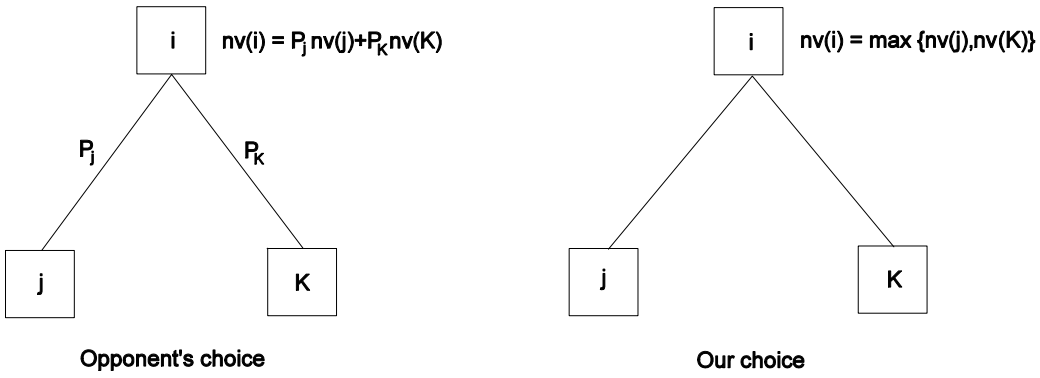


Figure 33: Calculation of Net Value of Internal Nodes

This defines $nv()$ as the superharmonic majorant corresponding to our opponent making random choices while we select the move which leads to the node with the greatest net value. Where L refers to a tree, we shall abbreviate $nv(\text{root}(L))$ as $nv(L)$.

We suppose the left and right subtrees of L to be L_L and L_R, and the left and right moves from root to have scores Y_L and Y_R, as illustrated overleaf in Figure 34.

Theorem 5.1 *For $\lambda \geq \kappa$, the value of the game is 0, and Policy π^*, described below, is the optimal policy for both players.*

Policy π^* moves to the daughter of root with the higher net value, and moves at random if there is no more search information.

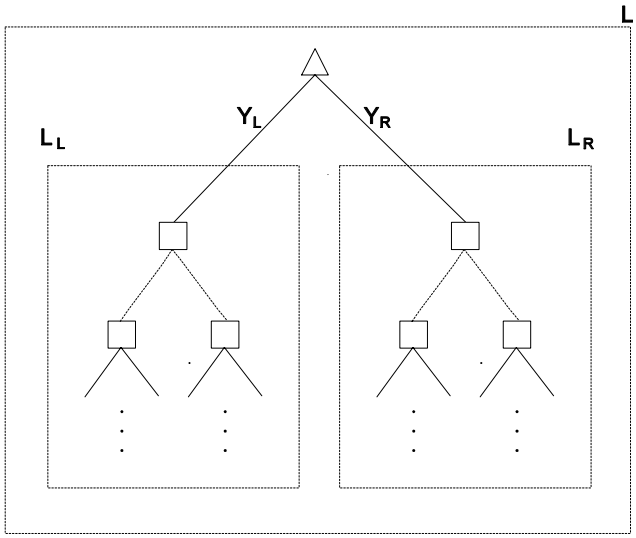


Figure 34: Information Notation

Proof: We denote by $V_{\text{Left }\pi}()$ the payoff from moving left and then following policy π and $V_{\text{Right }\pi}()$ the payoff from moving right and then following policy π. We consider the optimality equation for player 1, assuming his opponent pursues a policy of moving at random:

$$V(L) = \max\left\{E[V_{\text{Left}}(L)], E[V_{\text{Right}}(L)], E\left[\max_i\{V(L+i)\}\right]\right\}$$

Substituting in policy π^*:

$$\begin{aligned} V_{\pi^*}(L) &= \max\left\{E[V_{\text{Left }\pi^*}(L)], E[V_{\text{Right }\pi^*}(L)], E\left[\max_i\{V_{\pi^*}(L+i)\}\right]\right\} \\ &= \max\left\{E[Y_L + \alpha nv(L_L)], E[Y_R + \alpha nv(L_R)], E\left[\max_i\{nv(L+i)\}\right] - \lambda\right\} \\ &= \max\left\{nv(L), E\left[\max_i\{nv(L+i)\}\right] - \lambda\right\} \end{aligned} \tag{15}$$

If there is no search information, Y_L and Y_R are unknown, and $L_L = L_R = \phi$. Hence:

$$nv(L_L) = nv(L_R) = E[Y_L] = E[Y_R] = 0$$

$$E[Y_L + \alpha nv(L_L)] = E[Y_R + \alpha nv(L_R)] = nv(L)$$

Equation (15) therefore follows. If there is information, it follows directly from the definition of net value.

Now, let us consider how the expansion of a leaf i of L affects $nv(L)$. First, we note that if the opponent is to play from i then there is no expected change to $nv(L)$ caused by expanding it. If the searcher can play then expansion of i increases the net value of that leaf by $\alpha^{h_i} E[\max\{Y_{i1}, Y_{i2}\}]$, where h_i is the depth of leaf i.

From the definition of net value, the net value of a the root of a tree cannot change by more than the change to one of its leaves. Hence, for any i:

$$\begin{aligned} nv(L+i) - \lambda &\leq nv(L) + E[\max\{Y_{i1}, Y_{i2}\}] - \lambda \\ &\leq nv(L) \qquad\qquad \text{for } \lambda \geq \kappa \end{aligned}$$

Substitution of this in (15) shows us that $V_{\pi^*}(L) = nv(L)$ satisfies the optimality equation, so π^* is the optimal policy.

We have shown that if player 2 pursues a policy of always playing randomly, then player 1 can do no better than pursue an identical policy. This is a Nash equilibrium, and so both players must therefore be playing optimally, since the game is zero-sum. ■

Corollary 5.2 *For $\lambda = \kappa$, the value of the game is 0. Policy π^{1*}, described below, is optimal, as well as policy π^*.*

Policy π^{1*} moves to the daughter of root with the highest net value, if search information is available. If no search information is available, it expands root.

Proof: Theorem 5.1 establishes the optimality of policy π^* for $\lambda = \kappa$, and so we evaluate the payoff from playing policy π^{1*} against policy π^*. Policy π^* has been proved optimal above. The only case in which policy π^{1*} differs from policy π^* is if there is no search information. In this case, observe from the definition of net value that any action is optimal:

$$\begin{aligned} nv(L + \text{Root}) - \lambda &= nv(L) + E[\max\{Y_L, Y_R\}] - \lambda = nv(L) = 0 \\ &= E[V_{\text{Left } \pi^{1*}}(L)] = E[V_{\text{Right } \pi^{1*}}(L)] \end{aligned}$$

■

5.5 Summary

We have extended the models of Chapter 2 to allow the player to choose how much time to spend. The modifications made were minor, but relatively powerful, since they allow several of these simple games to be grouped together as part of a common problem, with the overall reward of a linear combination of the payoffs of each individual problem.

The result proved about the two-player search game presented in Section 5.4 is perhaps a somewhat unsurprising but nevertheless significant one.

It may be the first time that an optimal policy has been proved for a two-player game with such a realistic search structure, and it demonstrates that dynamic stochastic control can be used for this purpose.

The gap between the theoretical results proved in this chapter and the practical methods of time control described in Section 5.1 is large indeed. Of the time control policies described, those of the MGSS* and BP algorithms are the most theoretically interesting. As one-step lookahead policies, they both make the tacit assumption of decreasing marginal value of search: that if the cost of the next step exceeds the expected utility of the information, the cost of the next two steps will exceed the expected utility of the information they yield. As explained, this seems a fairly reasonable approximation for most games[23].

We have seen how the issue of time control is related to search control, and that for either to be satisfactory, a theoretically sound approximation must be found to the notion of 'utility'. The considerable problems surrounding this concept will be discussed in Section 6.5.

[23]Doubtless, pathological games could be deduced to show the inefficiency of this, just as Nau [55, 54, 56, 57] deduced pathological games to do the same for the minimax backing-up principle.

6 Computer Game-Playing

This chapter considers how the results so far established may be applied to the area of computer game-playing, as well as giving my personal views on the subject. Its tone is intended to be slightly less formal than previous chapters, since the problems tackled are bigger and more open-ended. We put this discussion in context by highlighting the tension between what is 'optimal' — in the dynamic stochastic control sense — and what is desirable in a practical sense. The remaining sections then outline, in varying degrees of detail, some new approaches to game tree searching and game-playing.

6.1 Optimality and Limited Rationality

The recent paper of Baum and Smith [6] summarises as follows the dilemma facing those who struggle for provably optimal policies in the field of computer game-playing:

> "One could . . . in principle, attempt a catalog of all leaf expansion strategies for any given game tree with a fixed number of leaves. If we could do vast computations offline, we could pinpoint one such strategy as optimal. But to be meaningful in practice, any "optimal" strategy must take into account its time cost (if we spend less time deciding which leaves to expand we can expand more leaves), and also the interaction of one leaf expansion with future expansion decisions. We know of no tractable approach to

> computing a provably "optimal" strategy. Since we don't know how to compute efficiently the exact decision theoretic utility of expanding leaves, we search for a useful approximation."

The result of Section 5.4 can be seen as a partial solution to the problems raised above, in that, by application of standard dynamic stochastic control methods, we have calculated the exact decision theoretic utility of expanding a leaf in a simple search game. However, this ignores the problem of limited rationality mentioned above; although the policy we have calculated is optimal — in the standard sense — we have no guarantee that it is useful in a practical sense. Since the dynamic stochastic control model takes no account of its deliberation time, there must always be a question mark over the practical feasibility of solutions adjudged to be 'optimal' on such a basis.

The way in which dynamic stochastic control has been used so far in this work is to deduce optimal policies for a range of simple games. These studies are intended to yield insight into the game-playing problem, and so lead on — directly or indirectly — to more complicated and powerful models.

Another way in which dynamic stochastic control might be applied to game-playing is to look for policies that are optimal within a certain class, such as the work done by Smith [79]. One might wish, for example, to choose a particular subclass of those tree search policies which are effectively implementable — for example, one which is $O(n)$ in the number of leaves searched. In this way one could ensure the feasibility of such optimal policies as were deduced. The success of such an approach would seem to rest entirely

upon an insightful choice of subclass.

Such an approach again stops short of tackling the fundamental problem of limited rationality. More powerful and more challenging still would be a framework which provided the means to show a policy to be optimal *inclusive of its own deliberation costs*, presumably with reference to some standard computer architecture. This would represent a very significant extension of standard dynamic stochastic control theory in the direction of computer science.

6.2 Conspiracy Probabilities

In their recent paper on probabilistic B* search, Berliner and McConnell [16] report the failure of their attempts to combine B* with the conspiracy number search algorithm. They summarise their conclusions as follows (italics in the original):

> "[Berliner] quickly found out ... something very important about why conspiracy number search has not worked. While it is useful to think of conspiracies in the above way, and to plan to attack the conspiracy with the fewest conspirators, in practice this does not work. Conspiracies fall into buckets. There are buckets of conspiracies of magnitude 1, of magnitude 2, magnitude 3, etc. In Chess (and we would assume in almost all domains), there are lots of potential conspiracies, and the number of conspiracies of

> magnitude 2 is usually quite large. Thus, when dealing with conspiracies of magnitude 2, one must examine them in some quasi-random order, and the chances of finding *the* conspiracy that is easiest to break is quite small. It is like a breadth-first search of conspiracies, with no other clue as to what might make a given conspiracy easy to break. *This is the reason for the failure of the conspiracy approach in game-playing.* There is no good method for deciding the weakest conspiracy of a given magnitude."

This is a significant objection to the conspiracy number search as described in Subsection 1.2.2; the implicit assumption in 'conspiracy number' that all nodes are equally likely to conspire is a serious limit to the efficiency of such a search. The successes of such simple methods as singular extensions and the null move are a testimony to how easily the most uncertain positions can be distinguished. It is therefore desirable to use such probabilistic information as is available. Subsection 3.4.2 shows how, in the two-valued case, this can be done optimally for conspiracies of size one by application of the stochastic OR-Tree model of Chapter 3. One step further down this road is to abandon altogether the notion of conspiracy 'number' in favour of the notion of conspiracy 'probability'. We now examine how this might be achieved, and reflect on some of the barriers to be overcome if such a scheme is to be developed into a practical selective search algorithm.

6.2.1 A Redefinition of Conspiracies

The child of root with the highest backed-up score we term the *provisionally best move*, and its score the *provisionally best score*. This is the move we make if no further search is carried out, so defines the utility of moving immediately from a position.

The original definition of a 'conspiracy' is described in Subsection 1.2.2. It refers to a minimal set of leaves which always has a chance of changing the provisionally best score. We note that not all such changes cause a change to the provisionally best move. This is a matter of importance to any algorithm which makes the Meta-greedy and Single-step Assumptions of Subsection 1.2.6, which are important ones on grounds of tractability. The reason is that the conjunction of these assumptions implies that unless a search has a chance of changing the provisionally best move, it has no value, and so the original definition of 'conspiracy' includes some sets of nodes which will never be of interest.

McAllester [50] originally detailed two categories of conspiracy, 'Prove Best' and 'Disprove Rest' strategies, as follows:

1. Sets of nodes that may decrement the score of the provisionally best move, so that it assumes a lower value than that of another move.

2. Sets of nodes that may increment the score of another move, so that it exceeds that of the provisionally best move.

We desire to include the following additional category to include those of the kind shown in the figure below, which were not originally included:

3. Sets of nodes which may decrease the provisionally best move's score and increase that of another move to exceed it.

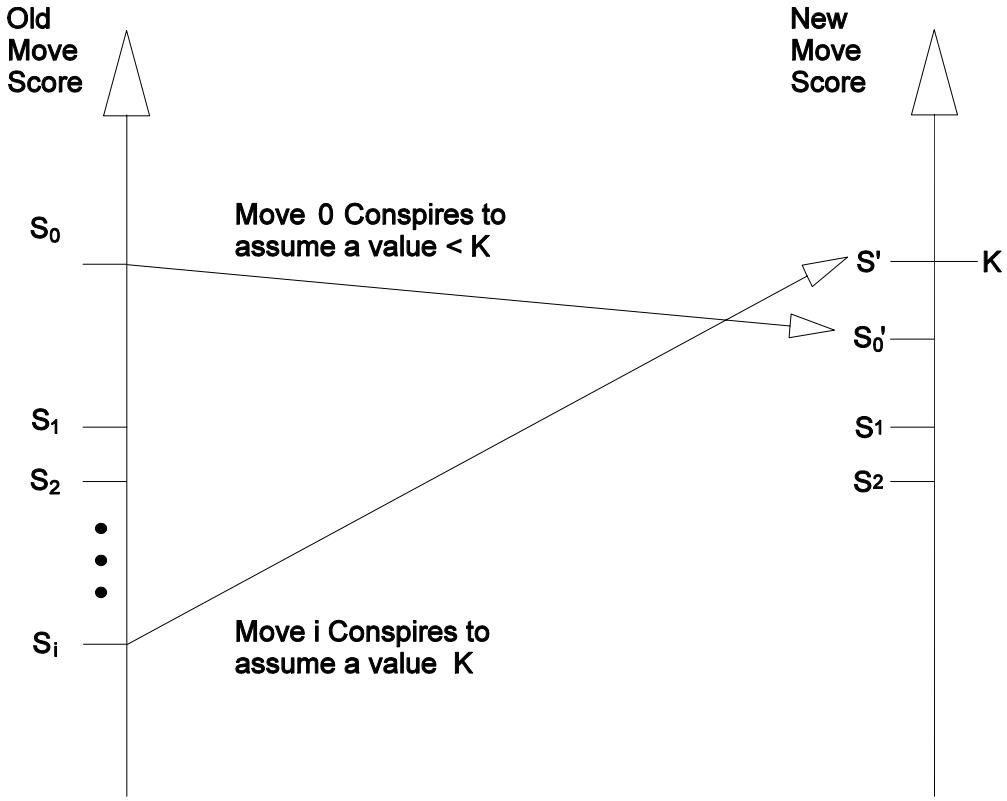


Figure 35: Moves 0 and i Conspire to Change the Provisionally Best Move

We now redefine the term *conspiracy*. We have previously defined a tree's conspiracies to mean a set of nodes which, if they were to assume different

scores (as a result of search) could together effect a change to the backed-up score of that tree's root node. We now introduce the idea of a *conspiracy to change the provisionally best move*. This is similar to the above definition except that, rather than changing root's score, the effect of the nodes' coordinated change is to change the tree's provisionally best move. This redefinition excludes those conspiracies which have no value under the single-step assumption, and includes the third class omitted by McAllester [50].

Assuming the children of root are ordered so that child 0 is the provisionally best move, and that move i has score s_i, a general form for such a conspiracy is (i, K), where i is the number of the move that conspires to assume a value $\geq K$ while move 0 conspires to assume a value of $< K$. Type 1 conspiracies may thus be expressed $(1, s_1)$, and type 2 conspiracies $(i, s_0 + \epsilon)$, where i is the move involved and ϵ the granularity of the evaluation function.

6.2.2 Ø-based Approximation Methods

The reward rate principle may be applied to the task of choosing a conspiracy in the following straightforward way. Each node of the search information is given a p and a t value with meanings analogous to the boxes case. For leaves, t, the expected time to expand each node of the conspiracy, is simply 1, while the probability that the conspiracy will change that node's value may be deduced directly from the prior distribution for expansion of that node.

For internal nodes of the search information with a negamax backed-up

score of 1, all its daughters of value -1 must conspire, so the p and t values are calculated from their daughter nodes as follows:

$$\begin{aligned} p_{\text{root}} &= \prod_{i=1}^{n} p_i \\ t_{\text{root}} &= \sum_{i=1}^{n} t_i \end{aligned}$$

The case in which an internal node is scored with a negamax backed-up score of -1 requires only a single daughter to conspire, and so the efficiency of the search centres around how this choice is made. The naive Ø-based approximation chooses the daughter with the greatest Ø value:

$$\begin{aligned} p_{\text{root}} &= p_{i^*} \\ t_{\text{root}} &= t_{i^*} \end{aligned} \quad \text{where } i^* \in \{1 \ldots n\} \text{ maximises } \frac{p_i}{t_i}$$

We now consider how to percolate these p and t values back up the tree. In order to search efficiently for a conspiracy, we need to seek out those conspiracies with as large a p value and as small a t value as possible. We now extend this principle, in a straightforward fashion, to the task of choosing which daughter to search from a '-1' node. Defining $\text{Ø}_i = p_i/t_i$ as before, suppose that, from a '-1' node, we will select the daughter which maximises Ø. This approach does not lead necessarily to choosing the overall conspiracy which maximises Ø, as we shall see from the following example.

Suppose that we are investigating the tree shown overleaf in Figure 36. The values shown beneath the nodes Q, S, and T, are the p and t values of these nodes. Since $\text{Ø}_{\mathsf{S}} = (.1/1) > (.5/6) = \text{Ø}_{\mathsf{T}}$, S is deemed the most promising descendant of R to search for a conspiracy. Hence, R is scored

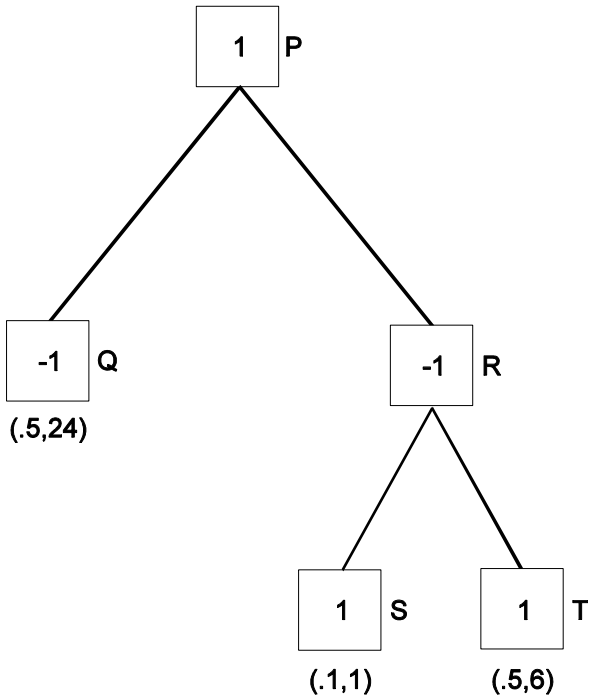


Figure 36: Inefficiency of Ø-based Approximation Method

(.1,1) and so P is scored (.05, 25). This leads to investigation of conspiracy {Q, S} which has a reward rate of 0.002. The optimal minimal conspiracy {Q, T}, has score (.15, 28), and so a reward rate of $0.008\dot{3}$. This example illustrates the problem that the overall reward $(1 - \Pi q)/\Sigma t$ cannot be maximised simply by maximising the individual p/t ratios of the subconspiracies. This problem is most severe for trees in which there is considerable variation between conspiracy t values.

A 'perfect' solution would require a catalogue of all the p and t values of

the conspiracies possible below a node. This would require large overheads, both in terms of meta-calculation and in storage. Since the space required to store the tree would be super-linear in n, the number of nodes in the tree, we discard this approach as infeasible.

A simple compromise which reduces inefficiencies associated with a poor choice of conspiracy whilst increasing resource usage by only a constant factor would be not to store all the possible conspiracy details at a node, but to store up to a certain fixed number.

In fact, since there are different percolation formulae for '1' nodes and '-1' nodes, it may well be efficient to store the details of different numbers of conspiracies for odd and for even depths of the tree. The motivation for storing a set of p and t values is to enable a better choice to be made about which conspiracy we wish to search, and so it is also reasonable to store more conspiracies for nodes higher up the tree (particularly the daughters and granddaughters of root), since these nodes have the largest range of possible conspiracies from which to choose. One way of visualising the approximation which is going on is to consider Figure 37, overleaf, which shows (a continuous approximation of) the profile of available conspiracies available from a node.

A simple example of how this approach would work is to store at each node the p and t values of two conspiracies. A natural approach would be to chose the conspiracy with the largest p value and the one with the smallest t value, (p_1, t_1) and (p_3, t_3) in Figure 37. A choice between these two extremes could therefore be made at the top level '1' node in the tree. If the overall

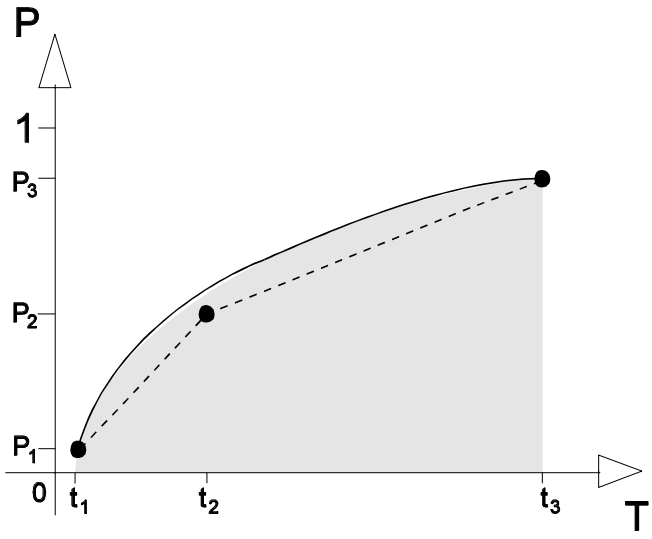


Figure 37: Range of Possible Conspiracies

reward rate of the conspiracy chosen in this manner was still too low, it could be improved by the storage of the details of more conspiracies. As the diagram shows, if details of a third conspiracy were stored at each node (say, that with the greatest Ø value) this would have an intermediate position, although it would not necessarily be the most probable conspiracy of its size.

6.2.3 PCN* Search

We consider a simpler selective tree search technique, PCN*, which is based upon conspiracy number search, but which searches conspiracies in order of probability, p, rather than reward rate, Ø. This is a less ambitious goal than attempting to select conspiracies based upon the p and t values. However, since the most likely conspiracy will tend — all other things being equal —

to be the one with the fewest nodes, and hence the cheapest to evaluate, this approximation may not be as crass as its simplicity would suggest.

We now explain briefly how it works, with the help of an example. The nodes of the tree each have a scalar score, amongst which the usual minimax relationship holds. We assume that the scores take integral values from 1 to m. In addition, every node has a *conspiracy probability vector* (c.p.v.), which details the probability with which this score can be changed by the investigation of a conspiracy amongst its children. Using C to stand for a single conspiracy, and $\mathbf{C}$ for the set of conspiracies, the c.p.v. is defined as follows for a node i with scalar score v_i:

$$\left(\begin{array}{rcl}
p^i_1 & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } = 1]\} \\
p^i_2 & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } \leq 2]\} \\
\vdots & & \\
p^i_{v_i-1} & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } \leq v_i-1]\} \\
p^i_{v_i} & = & 1 \\
p^i_{v_i+1} & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } \geq v_i+1]\} \\
\vdots & & \\
p^i_{m-1} & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } \geq m-1]\} \\
p^i_m & = & \max_{C\in\mathbf{C}}\{P[C \text{ will cause this node to have score } = m]\}
\end{array}\right)$$

The definition of $p^i_{v_i}$ can be motivated by expanding the concept of conspiracy to include the 'null conspiracy', which effects no change to a node's score with probability 1. This has the useful consequence that a node's $p^i_{v_i}$ value can be interpreted in harmony with both $p^i_{v_{i-1}}$ and $p^i_{v_{i+1}}$.

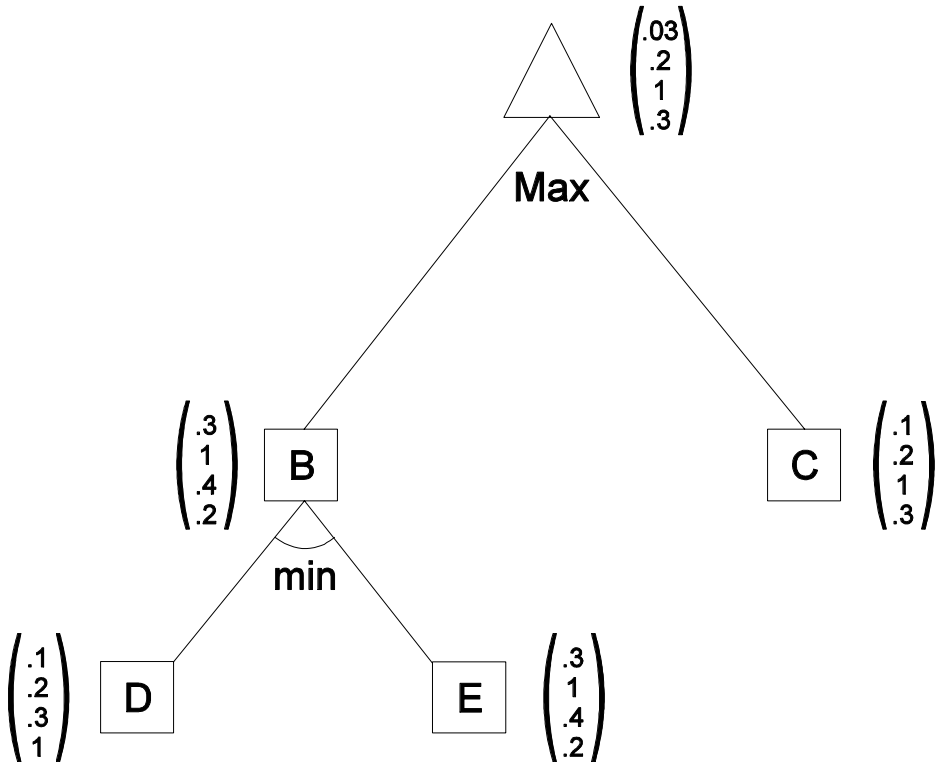


Figure 38: Example Conspiracy Probability Vectors

The above figure shows an example tree with conspiracy probability vectors, where m=4. Nodes C, D and E are leaves of the search information, and so their c.p.v.'s indicate the values that these leaves would take if expanded, which may be determined by the method of empirical evaluation described in Section 6.4. The c.p.v.'s of the internal nodes are percolated up from their daughters according to the formulae overleaf. These assume that a node has n daughters, which are ordered by their scalar score, so that

$v_1 \geq v_2 \geq \ldots \geq v_n$. For succinctness of expression, it will be useful to define two extra values, $v_0 = 1$ and $v_{n+1} = m$.

For *max* nodes : For *min* nodes :

$$p_j^{\text{parent}} = \begin{cases} \prod_{i=1}^{k} p_j^i & \left| v_k \geq j \geq v_{k+1} \right. \\ \max_i \{p_j^i\} & \left| v_0 \geq j \geq v_1 \right. \end{cases} \qquad p_j^{\text{parent}} = \begin{cases} \max_i \{p_j^i\} & \left| v_0 \geq j \geq v_1 \right. \\ \prod_{i=1}^{k} p_j^i & \left| v_k \geq j \geq v_{k+1} \right. \end{cases}$$

To see how this works in practice, suppose node D was expanded, and its c.p.v. was calculated (by percolating up from those of its daughters) to be $(.2, .4, 1, .2)$. Application of the formulae above yields a new c.p.v. for B of $(.3, 1, .4, .04)$. Note that the only element to have changed is p_4^{B}, the probability of a conspiracy causing the value of B to rise to 4. This being so, it is a trivial consequence of the percolation formulae that the only element of the c.p.v. of A which may change is p_4^{A}. In fact, a glance at the formulae show that p_4^{A} remains unchanged, since C was more likely to conspire to achieve a value of 4.

As demonstrated by the example above, the change of a leaf's c.p.v. does not necessarily percolate all the way back to root. For the purposes of complexity analysis, we nevertheless make this pessimistic assumption, and find that the cost of expanding a node is dominated by this percolation, which takes time proportional to the height of the tree. Expansion of a single node therefore has complexity $O(\log n)$, so PCN* search is $O(n \log n)$, comparable to BP and better than the MGSS* algorithm.

The implementation included in Appendix B searches one conspiracy at a time. If a greater degree of selectivity were desired, it could easily be modified to search one node at a time — picking from the most likely conspiracy the node with the greatest probability of conspiring. The price for this greater selectivity would be an increase in meta-calculation costs, although the algorithm would still be $O(n \log n)$. Another refinement over the included implementation would be to modify the criterion for selecting a conspiracy to investigate. Rather than just choosing the conspiracy with the greatest probability of making *a change* in the provisionally best move, account could be taken of the magnitude of the score difference. Conspiracies could be valued as follows:

$$V(C) = P_C(\text{Value of Best Move if } C \text{ works } - \text{ Value of current Best Move})$$

We make the point in Section 6.5 that for such a valuation to be sensible, the 'value' in the above expression should not be the simple scalar score, but should be a replacement that has been calibrated with respect to the final outcome of the game.

6.3 Choice of Search Step

The choice of search step size is inevitably a matter of compromise. On the one hand, the desire for selectivity motivates a small search step, on the other, the desire for minimal meta-calculation costs motivates a large search step. A further complication is the interaction between the choice of

step size and the time-control policy. We saw in Subsection 3.6.2 how the weakness of a one step lookahead policy was exacerbated by choice of too small a step size. The MGSS* algorithm of Subsection 1.2.6 suffers from this problem; the meta-greedy search control policy terminates the search once the search information has a conspiracy number of two or more. This is because the search steps are single leaf expansions, which — under the single-step assumption — are scored individually, so no single step has positive value.

The BP algorithm of Subsection 1.2.7 avoids this problem by not attempting to be so selective. This is a pragmatic response to the problem. The parameter, 'gulp size' controls selectivity, providing a continuum of behaviours from highly selective to non-selective. Experiments by Smith, Baum, Garrett and Tudor [80] with BP in the game of Othello show how the performance of BP depends upon a suitable choice of gulp size. They conclude that the algorithms performs best with a value of around 0.04, so that each search step expands 4% of the leaves of the search information.

It would be interesting to test the suggestion of Section 1.4 that it is desirable for the degree of selectivity to increase during the course of search. This could be done by modifying BP so that the gulp size varied according to the number of nodes in the search information.

Another refinement to the 'gulps' method would be to vary the way in which a leaf is expanded[24]. Leaves of the search information with higher QSS

[24]Russell and Wefald [72] report a significant strengthening of the MGSS* algorithm

could be searched, in a single 'expansion', to greater depths, while those with a lower QSS could be only partially expanded (i.e. some of their children could be generated and appended to the search information). This offers the possibility of a more reasoned way of tackling the problem of variable selectivity than the previous suggestion. At the start of the search, its effect might be expected to be similar, since the low conspiracy number of leaves in a small tree would result in their high QSS.

Such an approach would benefit from the estimation of probability distributions for the expansion of nodes to varying depths, and so might be expected to increase the overhead involved in deciding whether or not to carry out a gulp of search. On the other hand, as well as increasing the selectivity of each gulp, it would increase their size, and so it need not necessarily entail an increase in the total burden of meta-calculations.

6.4 Estimation of Probability Distributions

The PSVB*, MGSS*, BP and PCN* algorithms make use of functions which evaluate a position with a probability distribution. The purpose of these distributions is to show how the evaluation is likely to change if further search is carried out. Whilst the PSVB* algorithm obtains this distribution via a rather obscure method involving the null move search, the more modern algorithms all use the same method, which might be described as empirical evaluation, or 'training'.

when it was modified to partially expand nodes.

Human experts originally suggest a number of (usually quite simple) game-specific evaluation criteria known to be correlated with the winning chances of a position, and a means of combining them to deduce a single scalar score. Training proceeds by applying this evaluation function to a board position. This is searched to a certain depth, and the backed up scalar score of the position after the further search is noted. Once a large number of independently sampled positions have been examined in this manner, it is argued, the evaluation function will have a reliable probability distribution for each combination[25]. Russell and Wefald [70, 71, 72] used depth one searches to deduce their probabilities, while Baum and Smith [6] recommend empirical choice of the training depth.

One potential problem with this approach, in theory at least, is what we call the *representative positions problem*. A set of positions taken from beginners' games will be very different, in general, from a set of positions from expert play, and so the probability distributions may also be expected to differ. Baum and Smith appear to have discovered this problem empirically. Their solution is to use a two stage procedure. Firstly, the probabilistic element of their algorithm is disabled, and games are played using only a static evaluation function. Positions from these games are then used to calculate distributions which are, in turn, used to gather another set of positions. The

[25]The number of training games required may be adjusted by dividing the space of game positions up into a number of bins, according to the number and granularity of these criteria.

final probability distributions are then calculated with reference to the second set of positions.

The fact that their description of this was relegated to a footnote suggests that they did not find this to be a serious problem. They do not mention why they stopped after two steps of this procedure — whether convergence had been observed, or whether they were concerned about the possibility of degeneration. Position evaluation is relatively easy for the games treated by Baum and Smith (Kalah, Warri and Othello). For a game such as Go, for which position evaluation is notoriously difficult, the representative positions problem may prove a serious difficulty. Certainly, the theoretical problem of how to 'bootstrap' a set of positions that are representative of how the algorithm *will* play — when it has used the evaluation function derived from that set — remains unaddressed.

6.5 Utility

The reason why the game tree is searched is that by application of the evaluation function, it is possible to distinguish the good from the bad branches, and so guide both play and search.

For the purposes of guiding search, a static score is too simple a description of what search may reveal; a simple ordering of positions is not sufficient to measure the importance of searching a node because of the complicated effect of interactions with the scores of other nodes. We therefore require a richer representation — i.e. a probability distribution — to represent belief

about the possible consequences of search.

For the purposes of guiding play, a reasoned choice of move requires that an ordering of positions exist. If the positions are scored with probability distributions, the natural choice is its expectation. The only result of ultimate interest is the outcome of the game, and so if the expectation of the probability distribution is to be used in this way, it is essential that the scores be calibrated to the expected reward from playing the game. The MGSS* and BP algorithms carry out this calibration as explained in the previous section. The failure of the PSVB* and Probabilistic B* algorithms to carry out such a procedure is an omission no less serious just because they do not use expectation to choose between moves.

It is not as simple as is generally believed to calibrate position evaluation to something that might appropriately be referred to as 'utility' (i.e. probability of winning[26]), because of the complications raised in Chapter 2. Since the whole notion of attaching a win probability to a particular position is a simplification, problems are bound to occur if an attempt is made to exactly define it in practice.

The notion of 'utility' is central to the MGSS* and BP algorithms. They both make the simplifying assumptions that time cost is separable from the position to which it is applied, and from the opponent against which they play. Using **L** to stand for search information, they assume the following:

[26]The rest of this section makes the simplifying assumption that there are two only outcomes, with rewards 0 and 1. This does not restrict the generality of the discussion.

$$U(\mathbf{L},\ \tau_{\text{Us}},\ \text{Opponent},\ \tau_{\text{Opponent}},\ \mathbf{L}_{\text{Opponent}}) = V(\mathbf{L}) + V(\tau_{\text{Us}})$$

The fact that the opponent's search information, $\mathbf{L}_{\text{Opponent}}$, is unknown is clearly a strong practical argument for ignoring it in the calibration. Similarly, the case for ignoring the opponent in the calibration process is supported by the impracticality of having a separate training set for each opponent, or the difficulty and degree of approximation required to determine statistics able to summarise the range of possible opponents. Eventually, programs will benefit from calibration according to both the opponent and inference about the opponent's search information. In the meantime, the practical consequence from not taking into account the opponent's strength when calibrating the utility function is that the program will play inappropriately against an opponent of greatly different ability[27]. This is not a cause for great concern, since the primary aim of current research into computer game-playing is to devise programs that play as strongly as possible in even games.

The easiest approximation to address in the calibration process is the complete neglect of the opponent's time. Since this is readily available, to completely ignore it may seem rather slack but is in fact so standard as to

[27]For example, suppose a strong Chess program offered a beginner a queen's odds. If the utility function did not take the opponent into account, it would suggest that a loss was almost certain. This would cause a problem similar to the horizon effect; the program would play increasingly desperate and risky moves, in the hope that it had underassessed its chances. By contrast, a human expert would expect the beginner to make simple errors, and play accordingly.

pass without comment in the discussion of computer game-playing[28]. Nevertheless, it is of use in determining human play, a fact which will be agreed by any games player who has experience of playing under serious time controls. The most immediate use of this information would seem to be its (cautious) application to the area of time control, where it could be added in as another factor to the rather ad hoc algorithms that are currently the norm. It seems reasonable to vary the value of time in sympathy with the amount of time the opponent has. The justification for this is that if the opponent has a lot of time, this gives him the potential to think deeply about the position and so cause complications (by playing trick moves etc.) which will cause us to run short of time. Conversely, if the opponent has very little time, then either he is in time trouble — which suggests we should look for such complications — or else he has correctly judged that he will not need much more time, in which case we have probably been too frugal up to now if we have a lot of time left.

A more robust use of the opponents' time would be to take it into account when calibrating the evaluation functions. Attempting to equate a board position with a win probability ignores the aspect of time control altogether, limiting the effectiveness of time control. A more thorough approach would use a population of games to derive a function $U(\mathbf{L}, \tau_{\text{Us}}, \tau_{\text{Opponent}})$, which might then be used for time control as well as move selection. The modelling process can be aided by certain theoretical knowledge of the properties of

[28]I am not aware of any game-playing programs which use this information.

$U()$. We require a smooth function which is increasing in τ_{Us}, decreasing in τ_{Opponent} and — drawing upon the thoughts presented in Section 5.2 — that is subharmonic in τ_{Us} and superharmonic in τ_{Opponent}. From a theoretical point of view such a reasoned approach to time control would be greatly preferable to the current crude approximations; from a practical point of view, it would require a considerable increase in the computing power needed to calibrate the utility function, would lead to a more complicated program and might decrease the speed slightly, so computer game-playing may take some years to reach the point where it becomes a priority.

Of the game-playing algorithms devised so far, BP has come closest to defining a satisfactory model of utility. Inevitably, some sacrifices have been made to expediency. One of these is the oversimplified time control mechanism highlighted above, which, by the authors' own admission has 'an annoying sickness' [80].

Another shortcoming of BP is the strategy for leaf expansion. The 'gulp' idea is an important way of cutting down meta-calculation costs. However, the notion of 'gulp size' is a rather imperfect implementation of it. No justification is presented for why the gulp should be a fixed proportion of the information, much less a constant one. If QSS really is a good approximation to 'utility', a desirable goal would be to expand those leaves with the largest QSS values, *throughout the entire game*. This would suggest that an appropriate mechanism of leaf selection would be to expand in one gulp all those leaves with a QSS value above a certain threshold. This threshold

should be dynamic, changing in response to the findings of search. If time ran short, if the game looked like taking longer than expected, or if the position suggested that there would be many leaves with above average QSS ahead, the threshold should increase. Conversely, it would decrease in the opposite circumstances — the guiding principle should at every stage be the aim of maximising expected 'utility/unit of search'.

6.6 Trends in Computer Game-Playing

At the risk of stating the obvious, I hope that the reader is at this point completely convinced of the importance of quantifying the uncertainty around a position's static evaluation. This remark is by no means as trite as it may appear to the reader with a statistical background, since this realisation has taken the artificial intelligence community a very long time[29]. Almost all the top game-playing programs have no such element, being basically alpha-beta search engines with a selection of refinements such as those reviewed in Section 1.1.

The explanation for this is simple. Research into game-playing has been largely empirically-driven, and algorithms have not been developed *in vitro*. The meta-calculation costs associated with the processing of probability distributions are very considerable as compared with point estimates, while

[29]If the research published on tree search and game-playing is any indication, it is only just becoming widespread, almost 50 years after the notion of computer game-playing was first mooted by Shannon [76].

considerable computing power is required to realise the benefits of greater selectivity. This alone, apart from their greater intricacy, is sufficient to explain why probabilistic selective search techniques have not, in the past been in the mainstream of research. If they *had* been developed twenty years ago, they would not have held their own against the sheer speed of non-selective search algorithms.

One notable exception to the early concentration on 'brute force' methods was the work of the statistician Good. His 1968 "Five Year Plan for Automatic Chess" [30] would more realistically have been entitled a "Twenty Year Plan"; even in the late 1980's, when Russell and Wefald introduced the MGSS* algorithm, they found it only to be comparable with fixed-depth alpha-beta search[30]. The recent results of the BP program seem very encouraging, and so it seems as if, another 10 years on, selective searching has finally come of age. Since the advantage of being selective increases with the amount of computing power available, it will surely not be long before even the most powerful alpha-beta program, running on specialist hardware, will succumb to the greater intelligence of game-playing algorithms based on selective search. If the reader will indulge me in a little prediction, I suggest that by 2020, all computer game-playing programs will be Bayesians.

[30]Baum and Smith [6] have suggested that even this may have been an overoptimistic assessment of its performance.

6.7 Conclusion

This work has attacked the problem of computer game-playing from both ends. At one extreme are the models introduced, which have a provably optimal solution, at least in some circumstances. At the other, I have included some musings about the problem which inspired their development, that of how a statistically-based game-playing program might work. As we have seen, this problem does not have an optimal solution — in the classical sense — so while I may be disappointed with the size of the gap between the two extremes of study, I make no apology for its existence. I do hope that my work has gone some way towards enlightening the reader about what may be achieved by wider application of dynamic stochastic control methods to the tasks of tree search and game-playing.

It seems appropriate to conclude with the remark that, both as a games player and as a researcher, I find it a reassuring thought that it is impossible to define an 'optimal' strategy for game-playing, and so this problem is one which will remain permanently open.

A Summary of Notation

$d_i = (p_i,\ t_i, f_i)$	Details of node type i.
f_i	Offspring distribution of node type i.
$\mathbf{L} = (L_1, \ldots, L_n)$	Search information.
$L_i = (h_i, v_i)$	Leaf i, with height h_i value v_i.
$M(P)$	Retirement reward function.
P	Overall probability that an object exists.
$p_i = 1 - q_i$	Probability node type i contains an object.
$p_i^* = 1 - q_i^*$	Corrected probability of node type i.
$p_i' = 1 - q_i'$	Probability of an object not in node i or its descendants.
$p_i'' = 1 - q_i''$	Probability of an object amongst the offspring of node i.
$\pi_i(\mathbf{x})$	Action taken by policy π at time i from state $\mathbf{x}$.
t_i	Expected time to search node type i.
t_i^*	Corrected expected time of node type i.
$V()$	Optimal payoff function.
$V_\pi()$	Payoff function from following policy π.
$\mathbf{X}_\tau^\pi$	State reached by policy π at time τ.
Y_i	Difference between the value of leaf L_i and its daughters.
$\emptyset_i = p_i/t_i$	Reward rate of node type i.
$\emptyset_i^* = p_i^*/t_i^*$	Corrected reward rate of node type i.
$\langle \mathrm{F} \rangle$	No object is found, there are no descendants.
$\langle \mathrm{T} \rangle$	An object is found.

B Implementation of PCN*

```
// PCN1.cpp     Last Modified: 23/10/96   Probabilistic Conspiracy Numbers.
// (c) Robin Upton 1996
// Available from www.RobinUpton.com/research/phd/pcn1.cpp

class CPVector {
double _CP[NoDiscreteValues];
public:
double CP(int) const;
void Set_Cell(int, double);
CPVector();
};
class Node {
unsigned long _seed; /* Used to generate determined pseudo-random descendants. */
BoardPosn* _position;
long _depth;
int _score;
CPVector* _CPV;
Node* _parent;
int _childno; /* How this node is ordered in parent's child[ ] array. */
Node* _child[MaxNoChildren]; /* Children ordered by score. */
long depth() const;
  int childno() const;
void Add_child(Node*);  /* used in Expand(). */
void Number_children(); /* used in Expand(). */
void Recalculate_CPvector();
void Updated_child(int);  /* Percolates changes up the tree. */
int isbetter(int, int) const;  /* > for MAX nodes, < for MIN nodes. */
int ispreferable_to(Node*) const; /* Compares score, and if necessary CPV().*/
public:
unsigned long seed() const;
BoardPosn* position() const;
int score() const;
CPVector* CPV() const;
double CP(int) const;
Node* up() const;
Node* child(int) const;
Node(BoardPosn*,  Node*, unsigned long);
```

```
void Con_search(int); /* Search and try to change score to (int) or more. */
void Expand(); /* Expands node with simple 1-ply search. */
void PCN_Expand(); /* PCN updating that follows a call to Expand(). */
int player() const;
};

extern long Search_Time=DefaultSearchTime;
extern unsigned long No_Nodes_Searched=0;

void main(int argc, char *argv[] ){
 unsigned root_seed=1;           /* Game tree variables. */
 Node *best_kid, *pcn_root;      /* PCN* variables.*/
 long Last_Search_Time;
 double cons_prob, most_likely_cp;
 int best_nmrv, new_minrootvalue, overall_best_cn, best_cn_this_child, child_no;
pcn_root=new Node(new BoardPosn(ROOTSCORE),0,root_seed);/*Initialise the pcn search t
if (argc>1) Search_Time= atoi(argv[1]);   /* Read the command line argument. */
Last_Search_Time= Search_Time;
pcn_root->PCN_Expand();
        do {/* Select the most likely strategy, S.*/
   best_kid=pcn_root->child(0);
   most_likely_cp=best_cn_this_child=0;
   for (new_minrootvalue=pcn_root->child(1)->score();
new_minrootvalue<=1+pcn_root->score(); new_minrootvalue++)
        {/* Look at all new_minrootvalues between 1st & 2nd best node's scores. */
          cons_prob=child_no=0;
  /* Search through the children other than the best one. */
                 while (pcn_root->child(++child_no)!=0)
if (pcn_root->child(child_no)->CP(new_minrootvalue)>cons_prob) /* Pick the best CP. *
           { best_cn_this_child=child_no;
              cons_prob=pcn_root->child(child_no)->CP(new_minrootvalue); };
          cons_prob*=best_kid->CP(new_minrootvalue-1);/* Root must also be degraded.
          if (cons_prob>most_likely_cp)
           {/* Update the details of the best conspiracy. */
             overall_best_cn=best_cn_this_child;
             best_nmrv=new_minrootvalue;
             most_likely_cp= cons_prob; }
                 }/* We now know that the best conspiracy is one in which
  /*  pcn_root->child(0)              conspires to worse than `best_nmrv`
```

```
  /*  pcn_root->child(overall_best_cn) conspires to better than or equal to ‘best_nmr
   if (most_likely_cp > 0)
    {pcn_root->child(overall_best_cn)->Con_search(best_nmrv);
          best_kid->Con_search(best_nmrv-pcn_root->player());}
/* Use ’best_kid’, in case the children have been reordered. */
   else  {FILE* DebugF=fopen(DEBUGFILE,"a");
                fprintf(DebugF,"Search ended due to impossibility of conspiracy.\n"
                fclose(DebugF); break; /* Exit if the game is over. */ }
}
while (Search_Time>0);
}

void Node::Number_children()
 {int c=0; for (c=0; c<MaxNoChildren && child(c)!=0; c++) child(c)->_childno=c; }
void Node::Recalculate_CPvector() {
double best_prob_yet, prob_so_far;
int childno, cellscore;
/* Calculate the CPV cells for the ’desired’ changes. */
for (cellscore=MINVALUE+(player()==MAX)*(NoDiscreteValues-1);
cellscore!=score(); cellscore-=player())
{best_prob_yet=0; childno=0;
  do best_prob_yet=__max(best_prob_yet, child(childno)->CP(cellscore));
  while (child(++childno)!=0);
  CPV()->Set_Cell(cellscore,best_prob_yet);}
/* Calculate the CPV cells for the ’undesired’ changes. */
for (cellscore=MINVALUE+(player()==MIN)*(NoDiscreteValues-1);
cellscore!=score(); cellscore+=player())
{prob_so_far=1; childno=0;
  do prob_so_far*= child(childno)->CP(cellscore);
  while (child(++childno)!=0  && isbetter(child(childno)->score(),cellscore));
 CPV()->Set_Cell(cellscore, prob_so_far);}
CPV()->Set_Cell(score(),1); /* Set CPV cell for the null change.*/
}

void Node::Updated_child(int child_no){/*called when _child[child_no] is updated.*/
Node* temp;
int newscore=child(child_no)->score();
 while (child_no>0 && (isbetter(newscore, child(child_no-1)->score())))
   /* Promote child(child_no).*/
```

```
   { temp=child(child_no-1);
     _child[child_no-1]=child(child_no);
     _child[child_no]=temp; /* Swap the two node pointers around. */
     child(child_no-1)->_childno--;
     child(child_no)->_childno++; /* Update the _childno records. */
     child_no--; };
     while ((child(child_no+1)!=0)&&(isbetter(child(child_no+1)->score(),newscore)))
/*  Demote child(child_no).*/
{temp=child(child_no+1);
_child[child_no+1]=child(child_no);
_child[child_no]=temp; /* Swap the two node pointers around. */
child(child_no)->_childno--;
child(child_no+1)->_childno++; /* Update the _childno records. */
child_no++; };
     _score=child(0)->score(); /* Update the new score. */
     Recalculate_CPvector();
     // It is possible to reduce the complexity of Recalculate_CPVector()
     // here, by making use of the parameter passed.
     if(up()!=0)up()->Updated_child(childno());/* Percolate changes up the tree. */
}

int Node::isbetter(int i1, int i2) const { return player()*i1 > player()*i2; }
int Node::ispreferable_to(Node* N) const {
double margin=0; /* How much better "this" is than 'N'.*/
int celldiff=0;
if (isbetter(score(), N->score())) return 0;
if (isbetter(N->score(), score())) return 1;
do /* Scores equal, so we compare the cells of the CPV... */
{celldiff++;
  if (score()+celldiff<=MAXVALUE)
margin-=player()*(CP(score()+celldiff)-N->CP(score()+celldiff));
  if (score()-celldiff>=MINVALUE)
margin+=player()*(CP(score()-celldiff)-N->CP(score()-celldiff));
  if (margin>0) return 1;else if (margin<0) return 0;}
while (score()+celldiff>MAXVALUE && score()-celldiff<MINVALUE);
return 1;/* Arbitrarily break ties against 'N' if we've no more cells to compare.*/
}

unsigned long Node::seed() const { return _seed; }
```

```
BoardPosn* Node::position() const { return _position; }
int Node::score() const {return _score; }
CPVector* Node::CPV() const { return _CPV; }
double Node::CP(int cell) const { return CPV()->CP(cell); }
Node* Node::up() const { return _parent; }
Node* Node::child(int childno) const { return _child[childno]; }
Node::Node(BoardPosn* b, Node* parent, unsigned long newseed ) {
int i;
No_Nodes_Searched++; /* Keep track of #nodes expanded. */
        Search_Time--; /* Keep track of amount of time used. */
        _seed=newseed;
_position=b;
_score=b->evaluate();
_parent=parent;
if (up()==0) _depth=0; else _depth=1+up()->depth();
        _CPV=b->get_ppd(player());
for (i=0; i<MaxNoChildren; i++) _child[i]=0;
}

void Node::Con_search(int target) {
/* Picks conspiracy most likely to achieve score 'target' or more. */
int child_no=0;
 if (score()!=target) /* The target is not achieved - so conspire... */
   {if (child(0)==0) PCN_Expand();  /* We are at a leaf - expand. */
   else if (isbetter(target, score()))
   /* Desirable change. Pick the node with the highest consp. prob. */
   {while (CP(target) > child(child_no)->CP(target))   child_no++;
   child(child_no)->Con_search(target);}
   else /* Undesirable change. */
   {do child(child_no)->Con_search(target);
   /* All nodes with a score > target need to conspire. */
   while (isbetter(child(child_no)->score(), target) && child(++child_no)!=0);};
   }
}

void Node::PCN_Expand(){/* Expands node and recalculates CPVector as appropriate. */
 Expand();                                    /* First expand the node. */
 Number_children();                           /* Order them appropriately. */
 _score=child(0)->score();                    /* Get the new score first... */
```

```
 Recalculate_CPvector();                        /* then the new CPVector. */
 if (up()!=0) up()->Updated_child(childno());  /* Percolate stuff up the tree. */
}

void Node::Expand() { /* Expands node, with a 1-ply search. */
int child_no=0;
BoardPosnArray* childboards=position()->get_kids(BRANCHINGFACTOR, seed());
while (child_no<MaxNoChildren && childboards->posn(child_no)!=0)
{Node* new_child=new Node(childboards->posn(child_no),this,next_seed(seed(),child_no)
  Add_child(new_child); /* Add the extra child.*/
  child_no++; }
}

int Node::player() const { return 1-(int)(2*(depth()%2));}
double CPVector::CP(int cellscore) const { return _CP[cellscore-(MINVALUE)];}
void CPVector::Set_Cell(int cellscore,double value){_CP[cellscore-(MINVALUE)]=value;
CPVector::CPVector() {int i; for (i=0; i<NoDiscreteValues; i++) _CP[i]=0; }
```

C Model Enumeration Program

```
// XGTSM.cpp Last Modified: 7/5/97
// (c) Robin Upton 1997
// Available from www.RobinUpton.com/research/phd/xgtsm.cpp

// Model Default Parameters
#define default_K      10.0  #define default_Va       1.0
#define default_Vb      1.0  #define default_Pa       0.1
#define default_Pb1     0.1  #define default_Qb       0.1
#define default_U_threshold      0.5 /* This is not relevant for U_type=1. */
// Internals.
#define U_type    2        /* Selects type of retirement utility function. */
#define tolerance  0.00000000001     /* Needed to avoid round-off errors. */
#define noactions  3        /* Search A, Search B, Retire */
#define maxsize    51
#define infinity   999999
void calculate_V(int, int);
double U(int, int);
static double result[maxsize][maxsize][noactions];
static char policy[maxsize][maxsize]; static char pdisplay[noactions+2];
double K=default_K; double U_threshold=default_U_threshold;
double Va=default_Va; double Vb=default_Vb;
double Pa=default_Pa; double Pb1=default_Pb1;
double Qb=default_Qb; double Qa,Pb2;
int main(int argc, char* argv[])
{int a_target, b_target, i, j, k;
  time_t ltime;  time( &ltime );
  //********* SET UP THE GLOBALS TO THE RIGHT VALUES *******
  pdisplay[0]=46; /* '.' ~ Retiring is optimal.            */
  pdisplay[1]=97; /* 'a' ~ Searching A is optimal.         */
  pdisplay[2]=98; /* 'b' ~ Searching B is optimal.         */
  pdisplay[3]=61; /* '=' ~ both searches are equally good. */
  pdisplay[4]=32; /* ' ' ~ policy not defined.             */
  for (i=0; i<maxsize; i++) for (j=0; j<maxsize; j++)
for (k=0; k<noactions; k++) result[i][j][k]= -1;
  for (i=0; i<maxsize; i++) for (j=0; j<maxsize; j++) policy[i][j]= 4;
  for (k=0; k<noactions; k++) result[0][0][k]=0; policy[0][0]=0; /* Set up (0,0). */
```

```
  // ***************** INPUT MODEL PARAMETERS **************
  if ((argc <=2)||(argc>10)||(a_target=atoi(argv[1]))<0||(b_target=atoi(argv[2]))<0)
   {cout<<"Usage: "<<argv[0]<<
" <#A nodes> <#B nodes> \n[[[[[[[[K] U_threshold] Va] Vb] Pa] Pb1] Qb]\n";return 1;}
  if (a_target+b_target>=maxsize){cout<<
"#A nodes + #B nodes may not exceed "<<maxsize-1<<".\n"; return 1;}
     switch (argc)
  { case 10: Qb=atof(argv[9]);   case 9:  Pb1=atof(argv[8]);
    case 8:  Pa=atof(argv[7]);   case 7:  Vb=atof(argv[6]);
    case 6:  Va=atof(argv[5]);   case 5:  U_threshold=atof(argv[4]);
    case 4:  K=atof(argv[3]);    default:/* No parameters set. */;  }
  Qa=1-Pa;
  Pb2=1-Pb1-Qb;
  if (Qa<=0 || Pb2<0 || Va<0 || Vb<0|| K<0|| Pa<0||Pb1<0||Qb<0 )
{cout<<"Bogus parameters!"; return 1;}
  // *************** OUTPUT MODEL PARAMETERS ***************
printf("=============================================================================\n")
printf("Output from %s run on %s", argv[0], ctime( &ltime ) );
printf("=============================================================================\n")
printf("K=%.3f, Va=%.3f, Vb=%.3f, Pa=%.3f, Pb1=%.3f, Qb=%.3f\n",K, Va,Vb,Pa,Pb1,Qb);
printf("Utility function type=%i",U_type);
if (U_type>1) printf("  U_threshold=%.3f",U_threshold);
printf("\n=============================================================================\n
calculate_V(a_target, b_target);
  // ****** GRAPHICAL DISPLAY OF THE OPTIMAL POLICY *********
  printf("\n    ");  for (i=0; i<=a_target+b_target; i++)
{if (i>10) printf ("%i", i/10); else printf(" ");  }
  printf("\n    ");  for (i=0; i<=a_target+b_target; i++) printf ("%i", i%10);
  for (j=0;j<=b_target;j++)
   { printf("\n%2i: ", j);
      for (i=0; i<=a_target+b_target; i++) printf("%c", pdisplay[policy[i][j]]);
   }
  printf("\n"); return 0;
}

void calculate_V(int a, int b) {
  if (a+b>0)
  {/* Recur if we've not already calculated the values we need. */
  if (a>0 && policy[a-1][b]==4)  calculate_V(a-1, b);
```

```
  if (b>0 && policy[a+1][b-1]==4)  calculate_V(a+1, b-1);
  if (b>0 && policy[a][b-1]==4)  calculate_V(a, b-1);
  /* Set the result values. */
  result[a][b][0]=U(a, b);
        if (a==0) result[a][b][1]=infinity; /* Value from searching A. */
           else result[a][b][1]=Va+Qa*result[a-1][b][__min(2, policy[a-1][b])];
  if (b==0) result[a][b][2]=infinity; /* Value from searching B. */
   else result[a][b][2]=Vb+Pb2*result[a+1][b-1][__min(2, policy[a+1][b-1])]
+Qb*result[a][b-1][__min(2,policy[a][b-1])];
  /* Set the optimal choice of action. */
  if (result[a][b][0]<__min(result[a][b][1], result[a][b][2])) policy[a][b]=0;
         else if (result[a][b][1]+tolerance < result[a][b][2]) policy[a][b]=1;
       else if (result[a][b][1] > tolerance  +result[a][b][2]) policy[a][b]=2;
        else policy[a][b]=3; /*  Search 'A' or 'B' */
}}

// Utility Function...     (Must have U(Having found an object for sure)==0.
double U(int a, int b) {
  double prob_of_failure=(pow(Qa,a))*(pow(Qb+Pb2*Qa,b));
  /************ Utility Functions I Have used  **********************/
  switch (U_type+0){case 1: return K*__min(prob_of_failure, 1-prob_of_failure);
                    case 2: return K*(prob_of_failure<U_threshold);
                    case 3: return K*(prob_of_failure>U_threshold);
}
```

References

[1] J. D. Allen, A note on the computer-solution of connect-four, *in: Heuristic Programming in Artificial Intelligence: The First Computer Olympiad (eds. D. N. L. Levy & D. F. Beal)*, Ellis Horwood, Chichester, England, (1989), 134-135.

[2] L. V. Allis, H. J. van den Herik & I. S. Herschberg, Which games will survive?, *in: Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad (eds. D. N. L. Levy & D. F. Beal)*, Ellis Horwood, Chichester, England, (1991), 232-243.

[3] L. V. Allis, M. van der Meulen & H. J. van den Herik, Proof-number search, *Artificial Intelligence* **66** (1994), 91-124.

[4] L. V. Allis, H. J. van den Herik & M. P. H. Huntjens, Go-Moku solved by new search techniques, *in: Proceedings AAAI Fall Symposium Games: Planning and Learning (eds. S. Epstein & R. Levinson)*, AAAI Technical Report FS-93-02, (1993).

[5] T. Anantharaman, M. Campbell & F. Hsu, Singular extensions; adding selectivity to brute force searching, *Artificial Intelligence* **43** (1990), 99-109.

[6] E. B. Baum & W. D. Smith, A Bayesian approach to relevance in game playing, *Artificial Intelligence* **97** (1-2) (1997), 195-242.

[7] D. F. Beal, An analysis of minimax, *in: Advances in Computer Chess 2 (ed. M. R. B. Clarke)*, Edinburgh University Press, (1980), 103-109.

[8] D. F. Beal, Benefits of minimax search, *in: Advances in Computer Chess 3 (ed. M. R. B. Clarke)*, Pergamon Press, (1980), 17-24.

[9] D. F. Beal, Experiments with the null move, *in: Advances in Computer Chess 5*, Elsevier Science Publishers, Amsterdam, (1989), 103-109.

[10] D. F. Beal, A generalised quiescence search algorithm, *Artificial Intelligence* **43** (1990), 85-98.

[11] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, (1957).

[12] H. J. Berliner, The B* tree search algorithm: a best-first proof procedure, *Artificial Intelligence* **12** (1979), 23-40.

[13] H. J. Berliner, Backgammon computer program beate world champion, *Artificial Intelligence* **14** (1980), 205-220.

[14] H. J. Berliner, Computer chess at Carnegie-Mellon university, *in: Advances in Computer Chess 4 (ed. D. F. Beal)*, Pergamon Press, (1986), 166-180.

[15] H. J. Berliner, G. Goetsch, M. S. Campbell & C. Ebeling, Measuring the performance potential of chess programs, *Artificial Intelligence* **43** (1990), 7-20.

[16] H. J. Berliner & C. McConnell, B* probability based search, *Artificial Intelligence* **86** (1996), 97-156.

[17] D. Bertsimas & J. Niño-Mora, Discrete sequential search, *Information and Control* **8** (1965), 159-162.

[18] W. L. Black, Discrete sequential search, *Information and Control* **8** (1965), 159-162.

[19] A. de Bruin, W. Pijls & A. Plaat, Solution trees as a basis for game-tree search, *ICCA J.* **17** (4) (1994), 207-219.

[20] M. Buro, Probcut: an effective selective extension of the $\alpha-\beta$ algorithm, *ICCA J.* **18** (2) (1995), 71-76.

[21] M. S. Campbell & T. A. Marshland, A comparison of minimax tree search algorithms, *Artificial Intelligence* **20** (1983), 347-367.

[22] B. V. Dean, Stochastic networks in research planning, *in: Research Program Effectiveness (eds. M. C. Yovits et al.)*, Gordon and Breach, New York (1966).

[23] M. R. Garey, Optimal task sequencing with precedence constraints, *Discrete Mathematics* **4** (1973), 37-56.

[24] R. Gasser, Es ist entschieden: Das Mühlespiel ist unentschieden, *Informatik Spektrum* (5) **17** (1994), 314-317.

[25] J. C. Gittins, Bandit processes and dynamic allocation indices, *JRSS Series B* **41** (1979), 148-177.

[26] J. C. Gittins, *Multi-Armed Bandit Allocation Indices* (1989), Wiley, New York.

[27] K. D. Glazebrook, Stochastic Scheduling with Order Constraints, *Int. J. Systems Sci.* **7** (1976), 657-666.

[28] K. D. Glazebrook & R. Garbe, Reflections on a new approach to Gittins indexation, *J. O. R. S.* **47** (1996), 1301-1309.

[29] K. D. Glazebrook & J. C. Gittins, On single-machine scheduling with precedence relations and linear or discounted costs, *Operations Research* **29** (1) (1981), 161-173.

[30] I. J. Good, A five year plan for automatic chess, *Machine Intelligence* **2** (1968), 89-118.

[31] G. J. Hall, Sequential search with random overlook probabilities, *Annals of Statistics* **4** (1976), 807-816.

[32] H. Horacek, Reasoning with uncertainty in computer chess, *Artificial Intelligence* **43** (1990), 37-56.

[33] L. R. Harris, The heuristic search under conditions of error, *Artificial Intelligence* **5** (1974), 217-234.

[34] T. Ibaraki, Generalization of alpha-beta and SSS* search procedures, *Artificial Intelligence* **29** (1986), 73-117.

[35] W. B. Joyce, Organization of unsuccessful R & D programs, *IEEE Trans. Eng. Managm.* **18** (1971), 57-65.

[36] J. B. Kadane, Quiz show problems, *Journal of Math. Anal. Appl.* **27** (1969), 609-623.

[37] J. B. Kadane & H. A. Simon, Optimal strategies for a class of constrained sequential problems, *Annals of Statistics* **5** (1977), 237-255.

[38] H. Kaindl, Dynamic control of the quiescence search in computer chess, *European meeting on Cybernetics & Systems Research* **6**, Vienna, (1982), 973-978.

[39] I. Karatzas & S. E. Shreve, Connections between optimal stopping and singular stochastic control I: Monotone follower problems, *SIAM J. Control Optimality* **22** (1984), 856-877.

[40] I. Karatzas & S. E. Shreve, Connections between optimal stopping and singular stochastic control II: Reflected follower problems, *SIAM J. Control Optimality* **23** (1985), 433-451.

[41] I. Karatzas & S. E. Shreve, Equivalent models for finite-fuel stochastic control, *Stochastics* **18** (1986), 245-276.

[42] F. P. Kelly, Discussion of bandit processes and dynamic allocation indices, *JRSS Series B* **41** (2) (1979), 167-168.

[43] F. P. Kelly, A remark on search and sequencing problems, *Mathematics of Operations Research* **7** (1982), 154-157.

[44] D. E. Knuth & R. W. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975), 293-326.

[45] R. E. Korf, Generalized game trees, *Proceedings IJCAI-89*, Detroit, MI, (1989), 328-333.

[46] R. E. Korf, Linear-space best-first search, *Artificial Intelligence* **62** (1993), 41-78.

[47] R. E. Korf & D. M. Chickering, Best-first minimax search, *Artificial Intelligence* **84** (1996), 299-337.

[48] V. Kumar & L. N. Kanal, A general branch and bound formulation for understanding and synthesizing and/or tree search procedures, *Artificial Intelligence* **21** (1,2) (1983), 179-198.

[49] K. Lee & S. Mahajan, The development of a world class Othello program, *Artificial Intelligence* **43** (1990), 21-36.

[50] D. A. McAllester, Conspiracy numbers for min-max search, *Artificial Intelligence* **35** (1988), 287-310.

[51] T. A. Marshland, A. Reinefeld & J. Schaeffer, Low overhead alternatives to SSS*, *Artificial Intelligence* **31** (1987), 185-199.

[52] L. G. Mitten, An analytic solution to the least cost testing sequence problem, *J. Indust. Eng.* **11** (1960), 17.

[53] P. Nash, Optimal allocation of resources between research projects, *Ph.D. Thesis*, Cambridge University, (1973).

[54] D. S. Nau, Pathology on game trees: a summary of results, *Proceedings of the first annual conference on artificial intelligence*, (1980), 102-104.

[55] D. S. Nau, The last player theorem, *Artificial Intelligence* **18** (1982), 52-65.

[56] D. S. Nau, Decision quality as a function of search depth on game trees, *JACM* **30** (1983), 687-709.

[57] D. S. Nau, Pathology on gametrees revisited, and an alternative to minimaxing. *Artificial Intelligence* **21** (1983), 221-244.

[58] M. Newborn, A hypothesis concerning the strength of chess programs, *ICCA J.* **8** (4) (1985), 209-215.

[59] N. J. Nilsson, Searching problem-solving and game-playing trees for minimal cost solutions, *in: Information Processing 68, Proceedings of the IFIP Congress 1968 (ed. A. J. H. Morrell)*, North-Holland, Amsterdam, (1969), 137-153.

[60] A. J. Palay, The B* tree search algorithm — new results, *Artificial Intelligence* **19** (1982), 145-163.

[61] A. J. Palay, Searching with Probabilities, *Ph.D. Thesis*, Department of Computer Science, Carnegie-Mellon University, (1983).

[62] A. J. Palay, *Searching with Probabilities*, Pitman, Marshfield, MA, (1985).

[63] J. Pearl, Asymptotic properties of minimax trees and game-searching procedures, *Artificial Intelligence* **14** (1980), 113-138.

[64] J. Pearl, On the nature of pathology in game searching, *Artificial Intelligence* **20** (1983), 427-453.

[65] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, (1984).

[66] A. Plaat, J. Schaeffer, W. Pijls & A. de Bruin, Best-first fixed-depth minimax algorithms, *Artificial Intelligence* **87** (1996), 255-293.

[67] R. L. Rivest, Game tree searching by min/max approximation, *Artificial Intelligence* **34** (1988), 77-96.

[68] I. Roizen & J. Pearl, A minimax algorithm better than alpha-beta? yes and no, *Artificial Intelligence* **21** (1,2) (1983), 199-220.

[69] S. M. Ross, *Introduction to Stochastic Dynamic Programming*, Academic Press, London, (1983).

[70] S. Russell & E. Wefald, On optimal game-tree searching using rational meta-reasoning, *Proceedings IJCAI-89*, Detroit, MI, (1989), 334-340.

[71] S. Russell & E. Wefald, Principles of meta-reasoning, *Artificial Intelligence* **49** (1991), 361-395.

[72] S. Russell & E. Wefald, *Do the Right Thing — Studies in Limited Rationality*, MIT Press, Cambridge, Mass., (1991).

[73] J. Schaeffer, Conspiracy numbers, *Artificial Intelligence* **43** (1990), 67-84.

[74] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu & D. Szafron, A world championship caliber checkers program, *Artificial Intelligence* **53** (1992), 273-289.

[75] J. Schaeffer, Personal communication, 12th February 1998.

[76] C. E. Shannon, Programming a computer for playing chess, *Philos. Magazine* **41** (1950), 256-275.

[77] H. A. Simon & J. B. Kadane, Optimal problem-solving search, *Artificial Intelligence* **6** (1975), 235-247.

[78] J. J. Slagle & J. K. Dixon, Experiments with some programs that search game trees, *JACM* **16** (2) (1969), 189-207.

[79] W. E. Smith, Various optimizers for single stage production, *Naval Research Logistics Quarterly* **3** (1956), 54-66.

[80] W. D. Smith, C. Garret, E. Baum & R. Tudor, Experiments with a Bayesian game player, (1996), *(Submitted for publication)*, also at http://www.neci.nj.com:80/homepages/eric/eric.html

[81] G. C. Stockman, A minimax algorithm better than alpha-beta?, *Artificial Intelligence* **12** (1979), 179-196.

[82] L. D. Stone, *Theory of Optimal Search*, Academic Press, London, (1975).

[83] C. W. Sweat, Sequential search with discounted income, the discount a function of the cell searched, *Ann. Math. Statist.* **41** (1970), 1446-1455.

[84] A. & B. Szabo, The technology curve revisited, *ICCA J.* **11** (1) (1988), 14-20.

[85] J. N. Tsitsiklis, A short proof of the Gittins index theorem, *Annals Of Applied Probability* **4** (1994), 194-199.

[86] J. W. H. M. Uiterwijk, H. J. van den Herik & L. V. Allis, A knowledge-based approach to Connect Four: the game is over white to move wins, *in: Heuristic Programming in Artificial Intelligence: The First Computer Olympiad (eds. D. N. L. Levy & D. F. Beal)*, Ellis Horwood, Chichester, England, (1989), 113-133.

[87] I. Wegener, The discrete sequential search problem with nonrandom cost and overlook probabilities *Mathematics Of Operational Research* (3) **5** (1980), 373-380.

[88] G. Weiss, Branching bandit processes, *Prob. Eng. Inform.* **2** (1988), 269-278.

[89] P. Whittle, Multi-armed bandits and the Gittins index, *JRSS Series B* **42** (1980), 143-149.

[90] P. Whittle, Arm-acquiring bandits, *AAP* **9** (2) (1981), 284-292.