# EXPERIMENTS IN FORWARD PRUNING WITH LIMITED EXTENSIONS

*Chun Ye and T. Anthony Marsland*

Computing Science Department
University of Alberta
Edmonton

ABSTRACT

Several heuristics have been developed to reduce the effort needed to select a move in chess. Some of them dynamically change the order in which moves are considered during the search, others assess the opponent's threats by making a null move. There are many more possibilities including forward-pruning mechanisms recognizing when it is futile to pursue a line of play along the full length of a given path. This paper considers a spectrum of simple heuristics and strategies and applies them to Chinese chess. It also tries to provide quantitative data to support the use of not only these standard techniques but also of some newer ideas involving attack-evading moves and strictly-forced moves. The effect on these methods of the significantly different draw-by-repetition rules for Chinese chess is also considered.

## 1. INTRODUCTION

Over the years, several move-ordering heuristics, search-depth extension methods and memory functions have been developed for computer chess programs. Some of them have been widely implemented and extensively studied (Marsland, 1986), others have been tried and found effective, but only limited quantitative supporting data was provided (Birmingham and Kent, 1977), and finally a few are still quite new (Beal, 1989; Anantharaman *et al*., 1988). In this paper many of the methods with which there is only modest working experience (or little published data) have been implemented in the Chinese chess program, *Abyss*, and data gathered to help make an authoritative comparison.

Chinese chess, a game much like chess but played on a board with 10×9 intersections, was chosen not only to avoid replication of some earlier work, but also to see how well the techniques developed for computer chess carry over to a related domain. Unlike chess, there are seven piece types, with different restrictions on their moves. For example, the King/General and Queen/Guard must stay in the "palace" (a 3×3 region), the Bishop/Elephant must stay on the home side of the "river" (a space diving the board into two halves, and taking one move to cross). Also, after crossing the river the Pawn/Soldier can move both forward and sideways, but does not promote to a more valuable piece when it reaches the back rank. Of the remaining pieces, the Rook/Warrior and Knight/Horse are substantially the same as their chess counterparts (a Horse may be blocked, however), while the Cannon is much like a Rook, but captures with an inline jump reminiscent of a checkers (draughts) move. One convenient English-language reference for the rules and examples of play is Lau's (1985) book.

Despite the obvious differences in the board and the pieces, the basic ideas in the two games are the same: win pieces and mate the King! In the eyes of humans, playing strategies developed for the one game carry over naturally to the other, but with two important differences. In Chinese chess a stalemate does not exist: if you cannot move you are lost. There is also an important difference in the notion of draw by repetition. Although the same idea is present in Chinese chess, it is moderated by the restriction that the drawing cycle (threefold repetition of position) exclude a sequence of threats. Thus a draw by perpetual check is not allowed. But draw cycles are surprisingly common in Chinese chess, and since their outcome may seriously affect some of the heuristic methods employed, a detailed description of the algorithm used to detect illegal repetitions in *Abyss* is provided in the Appendix. This is done not only for completeness, but also as a first step towards a full algorithmic

treatment of this seemingly complex draw-by-repetition mechanism that is a fundamental component of Chinese chess.

## 2. MOVE-ORDERING HEURISTICS

It is well-known in game-tree search that the most effective way to reduce the size of an $\alpha$-$\beta$ search is to improve the move-ordering mechanisms. The simplest scheme is to consider capture moves before all others, and so use of this simple heuristic is here taken for granted and forms the basic reference program (`+cap`) for comparison purposes. Two other important move-ordering heuristics are: first try the move held in the transposition-table entry before doing either a move generation or considering capture moves (Slate and Atkin, 1977), and use the history heuristic (Schaeffer, 1983, 1989) to order the (remaining) moves, so that those which frequently achieve a cut-off elsewhere in the tree are tried before the others. It is now well established that these three methods combine together well, achieving an average search reduction of more than 50%, as Table 1 shows. In the Depth = 5 data, the 32K-entry transposition table accounts for 75% of the search reduction and the history heuristic for the remaining 25%, over use of capture moves alone. However, in these experiments, use of the transposition table was not restricted to providing a preferred move, but included the normal search savings that come from the recognition of transposition of moves. Our failure to isolate this component for the results in Table 1 may be considered a minor flaw.

**Table 1: Experiments with Interior Move Orderings**

| Orderings | Depth = 3 | | | Depth = 4 | | | Depth = 5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nodes | ratio | hit % | nodes | ratio | hit % | nodes | ratio | hit % |
| `+cap` | 263085 | 1.00 | 18 | 1978554 | 1.00 | 30 | 8883580 | 1.00 | 34 |
| `+cap+tra+his` | 158359 | 0.60 | 18 | 788646 | 0.40 | 30 | 4238259 | 0.48 | 34 |

The data in Table 1 was obtained by searching the 50 tactical Chinese chess positions listed in Ye's thesis (Ye 1992, Appendix, 2). Following the normal convention, if *Abyss* selects the first move of the winning combination it was accepted as "solving" the problem. For the 50 problems this success rate is listed as the "hit %" in Table 1 and elsewhere. A more stringent requirement that the correct principal variation also be identified yields a consistently lower figure of about 66% of the hits for 5-ply searches (Ye, 1992). Omitted here is data pertaining to the use of a refutation table (a table showing an adequate continuation for every move in the original position under study). Its incremental effect is small when an appropriately sized transposition table is used. Also, even when deep searches overload the transposition table, the entries corresponding to the principal refutations can be protected from being overwritten from one iteration to the next. Strangely, there has not yet been a discussion of the added management costs and the merits and benefits that accrue to this refinement. With this addition the transposition table effectively makes the refutation table redundant, even on the deepest searches.

To be truly effective the transposition table must not be overwritten excessively. In particular, the entries corresponding to critical nodes of the tree should not be overwritten. This can be done through careful table management techniques, or by choosing a table size that is appropriate for the average size of the trees being searched. The effectiveness of the table can be captured experimentally by measuring the percentage of the time that valid entries are overwritten during the search process. Figure 1 shows the data for 5-ply searches with table sizes varying from 2K entries to 1024K entries (a typical entry might require 8-10 bytes). By accepting a 10% overwrite rate one would be happy with a 32K-entry table, which was our choice here.

### 2.1. The Null-Move Heuristic

In addition to its move re-ordering property a transposition table automatically yields a search reduction, because results from traversing one subtree can be retrieved and re-used later. Another search reduction technique, experimented with by Ken Thompson and others in the 1970s, is the *null-move* heuristic (Goetsch and Campbell, 1990); it is also a means of improving search speed with little risk. *Abyss* tries a null-move search at the internal nodes with a depth reduction of 1 ply before it starts searching legal moves. If the value returned

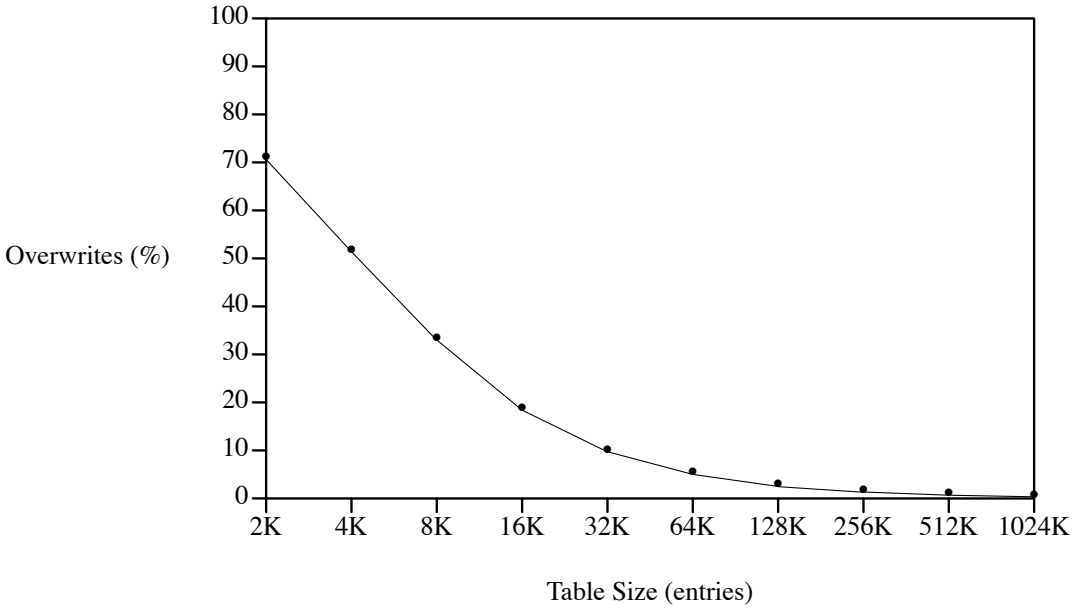


**Figure 1:** Overwrite Percentages versus Transposition-Table Size.

exceeds the $\beta$ bound, the value is accepted as a true cutoff; otherwise, it is used to improve the $\alpha$ bound. The heuristic is not used during the endgame or if the King is in check.

It is straightforward to add the null-move heuristic to any version of the $\alpha$-$\beta$ search algorithm, and it can be applied either at all interior nodes or only at those nodes up to the frontier (the last layer before the search horizon). In our experiments, the latter choice was shown to be better in terms of node search reductions, and without loss of move quality. Clearly there is no point in doing a null-move search at the frontier since no new information can be gained.

### 2.2. The Futility Cutoff

Other ideas for forward pruning have appeared many times, e.g., the *Gamma Algorithm* by Newborn (1975, pp. 177-178), the *razoring* technique by Birmingham and Kent (1977) and some special forward-pruning methods used in *Chess 4.5* (Slate and Atkin, 1977). All these heuristics are generally applicable but using them involves some risk. A safer variation is the heuristic which Jonathan Schaeffer calls the *futility cutoff* (Schaeffer 1986, pp. 33-34). A futility cutoff differs from the earlier heuristics in the following ways:

- First, use of a futility cutoff is restricted to the frontier nodes in the search tree.
- Second, both the material merit and maximum positional value are used to decide whether to stop the search.
- Finally, the search does not necessarily stop when the decision criterion above is met. In particular checking moves are not forward pruned this way, nor are those capturing moves for which the material merit alone yields a score within the current window.

In other words, the futility cutoff is a lower risk transformation of nodes near the frontier into tip (horizon) nodes when certain criteria are met. Although this heuristic is often mentioned, previously only Schaeffer (1986) provided quantitative data to show its effectiveness.

To illustrate use of the futility cutoff in the $\alpha$-$\beta$ algorithm, Figure 2 presents C-like pseudo code for one implementation. Function *value(side, position)* returns the static value (material balance) of *position* plus a maximum positional score for *side* which is used to determine whether to apply the futility cutoff. Figure 2 also illustrates how a legal draw by repetition is distinguished from an illegal repetition, as specified in Figure A1 of

the Appendix. The experimental results for the futility-cutoff heuristic (`-null+fcut`) are given in Table 2. The combination of both the futility cutoff and the null-move heuristic (`+fcut+null`) is also included. The basis for comparison (`-null-fcut`) is *Abyss* using the three move-ordering methods: transposition-table move, captures, and the history heuristic, i.e., the (`+cap+tra+his`) entry from Table 1.

**Table 2: Null Move and Futility Cutoff Heuristics**

| Experiments | Depth = 3 | | | Depth = 4 | | | Depth = 5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nodes | ratio | hit % | nodes | ratio | hit % | nodes | ratio | hit % |
| `-null-fcut` | 158358 | 1.00 | 18 | 788646 | 1.00 | 30 | 4238259 | 1.00 | 34 |
| `+null-fcut` | 133220 | 0.84 | 18 | 706844 | 0.90 | 30 | 2563303 | 0.60 | 30 |
| `-null+fcut` | 115169 | 0.73 | 18 | 706245 | 0.90 | 30 | 2392147 | 0.56 | 34 |
| `+null+fcut` | 104234 | 0.66 | 18 | 641469 | 0.81 | 30 | 1887356 | 0.45 | 30 |

As can be seen from Table 2 the futility cutoff is the preferred choice, since the success rate at Depth = 5 for the null-move heuristic is poorer. This deterioration in success rate may come partly from the nature of these two heuristics, but it could also depend on the test suite we chose; using different test positions may yield other results. The two heuristics (`+null+fcut`) combine well, giving greater savings without degrading the performance further. For this reason, both the null move heuristic and the futility cutoff are included in the current version of *Abyss* and form the new reference program for all the remaining experiments described here. Note that the null-move heuristic is a search-reduction technique, and so there is some risk in its use (i.e., it may show a lower success rate). Also, in our implementation the null-move heuristic was applied recursively, thus increasing the chance of error. Nevertheless, on balance Goetsch and Campbell (1990) believe that the technique is worth trying, and so do we.

## 3. EXTENSIONS WITH DOMAIN-SPECIFIC KNOWLEDGE

No matter how fast computers become there will always be some search horizon beyond which moves must be neglected. To reduce the damage from this neglect the notion of *selective extensions* has been introduced, specifically increasing the search by an extra ply (plies) when certain criteria are met.

For instance, chess-playing programs usually extend the search by an extra ply when the side to move is in check, since being in check can be a serious threat. This is one example of using domain-specific knowledge to extend the search. Other approaches include extending on recaptures (Ebeling, 1987, pp. 101-102), on pawn moves to the 6th and 7th rank in chess (Kaindl 1982, Scherzer *et al*., 1990), on moves near the territory of the opponent's King (Anantharaman, 1991), on strictly forced moves (say if one side has a single legal move) and on certain evading moves to bring to safety a piece under attack (an *ad hoc* heuristic tried in *Abyss*). The latter two have not yet been adequately assessed in the computer-chess literature, but are nonetheless worth considering. Here, we provide the descriptions of these heuristics with experimental results, and share our experience in implementing them in the *Abyss* Chinese chess program.

Note that the implementation of these heuristics is simple and straightforward. Suppose we have a function called *forcing(move, position)* which specifies whether the *move*, if made on the current *position*, meets the definition of a *forcing move* (either a check evasion, a recapture, a king threat, an attack-evading move or a strictly-forced move). The $\alpha$-$\beta$ search algorithm can be easily modified to suit this extension strategy, along the lines shown in Figure 3. So the remaining task is to assess forcing moves to decide when they should cause a search extension.

It is remarked that the function *forcing()* invoked in Figure 3 has been tailored to the check evasion case, so that *forcing(move, position)* returns **true** if *move* is a check, causing all nodes expanded in this layer (where the opponent's King is in check) to be searched without a depth reduction. Of course, this is only one way of adding the extension heuristics on forcing moves. The five forcing moves assessed are:

(a) Check Evasion

```
int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, rept, num_moves, p, score, lower, fcut;
  struct { int from, to; } moves[MAX_MOVES];

  rept = repetition(side, position);   /* Is the position duplicated? */
     /* 0, no repetition; -1, illegal repetition; 1, draw repetition. */
  if (rept ≡ 1) return(DRAW);          /* A legal draw */
  if (rept ≡ -1) return(ILLEGAL)       /* An illegal repetition */
  if(depth ≤ 0)                        /* A tip node? */
    return(evaluate(side, position, α, β));

  /* decide whether we should apply futility cutoff;
   * value() is material balance plus maximum positional score.
   */
  fcut = (depth ≡ 1) && (value(side, position) ≤ α);

  num_moves = generate(side, position, moves);
  score = -∞; p = 0;

  while(p < num_moves && score ≡ -∞) {
    successor = nextposition(moves[p], position);
    if(!fcut || value(side, successor) > α || check(!side, successor))
      score = -pvs(!side, successor, -β, -α, depth-1);
    p++;
  }

  while(p < num_moves) {
    if(score ≥ β)
      goto out;
    successor = nextposition(moves[p], position);
    if(!fcut || value(side, successor) > α || check(!side, successor))
    {
      lower = max(α, score);
      merit = -pvs(!side, successor, -lower-1, -lower, depth-1);
      if(merit > score)
        if(merit > α && merit < β && depth > 1)
          score = -pvs(!side, successor, -β, -merit, depth-1);
        else
          score = merit;
    }
    p++;
  }

out:
  return(score);
}
```

**Figure 2:** Principal-Variation Search using a Futility Cutoff.

(b) Recaptures

(c) King Threats

(d) Attack-Evading Moves

(e) Strictly-Forced Moves

```
int pvs(side, position, α, β, depth)
  int side, position[90], α, β, depth;
{
  int successor[90], merit, ndepth, num_moves, p, score, lower;
  struct { int from, to; } moves[MAX_MOVES];

  if(depth ≤ 0)
    return(evaluate(side, position, α, β));

  num_moves = generate(side, position, moves);
  successor = nextposition(moves[0], position);
  if(forcing(moves[0], position))
    score = -pvs(!side, successor, -β, -α, depth);
  else
    score = -pvs(!side, successor, -β, -α, depth-1);

  for(p = 1; p < num_moves; p++) {
    if(score ≥ β)
      goto out;
    lower = max(α, score);
    successor = nextposition(moves[p], position);
    if(forcing(moves[p], position))
      ndepth = depth
    else
      ndepth = depth - 1;
    merit = -pvs(!side, successor, -lower-1, -lower, ndepth);
    if(merit > score)
      if(merit > α && merit < β)
        score = -pvs(!side, successor, -β, -merit, ndepth);
      else
        score = merit;
    }
  }

out:
  return(score);
}
```

**Figure 3:** Principal-Variation Search with Extension on Forcing Moves.

Because of its simplicity and efficiency, *check evasion* is perhaps the most commonly found feature in chess programs. A checking move usually forms a major threat, therefore a one ply deeper search might reveal some tactics that are beyond the original horizon. The situation is of course substantially the same in Chinese chess, so one can expect a similar benefit from adding such a heuristic.

Equally, since capturing is the essence of tactics in chess (and Chinese chess), and a capture search (Bettadapur and Marsland, 1988) forms the kernel of quiescence search, some captures are more or less forced. For example, it might be worthwhile to extend one more ply on *recaptures*, as defined by Ebeling (1987, pp. 101-102), with the hope that some deep tactics can be revealed. Note that extending all recaptures could be expensive since a capture does not restrict the move choices by the opponent, therefore care is necessary to avoid a search explosion. In *Abyss*, we adopt the same rule as in *Hitech* (Ebeling, 1987; Berliner and Ebeling, 1989); only recaptures that bring the material merit into the window of the initial root value are considered.

It has already been mentioned that the *king-threat* heuristic can be used to alleviate the problem that computer chess programs normally lack knowledge about long-term threats against the King (Anantharaman, 1991). In Chinese chess, the King is perhaps more vulnerable to attack than in chess, since it is confined to only nine squares (called the *palace*). On the other hand, the need to retain an adequate mating force to assault the King

is more important, because there is no pawn promotion rule. In fact, there is an adage in Chinese chess which says: "three pieces beside [the opponent's palace] wins the game;" On balance, Chinese chess motivates use of the king-threat heuristic more strongly than chess.

Since the King is confined to the palace, that region can be used to identify the squares around one side's King, but such a treatment is too static and might cause evaluation inaccuracies. In *Abyss* we adopt a definition (Anantharaman, 1991) equivalent to that used in *Deep Thought*. We consider the King in jeopardy if the number of squares around one's King (exactly one square away) that are under the opponent's attack exceeds a certain threshold (3 is chosen); we then extend the search. Several experiments were done. Some were aimed at deciding how many (1 or 2) plies we should extend when a king threat is detected. Others were used to determine whether to extend the positions where the attacked squares around one's King exceed the threshold or only consider it when such a position occurs after making a move. The best results seem to come from using the king-threat extension only for one ply, and only considering the situation where the attacked squares around the King exceed a threshold after one side has made a move. Using this implementation, move quality was maintained, with only a moderate increase in node expansions. Therefore, this method is chosen in *Abyss* for the experiments involving king threats.

### 3.1. Attack-Evading and Strictly-Forced Moves

Another possibility is to extend the search for moves that bring to safety a piece under attack. This may be viewed as a generalization of the *check-evasion* heuristic, since otherwise the piece under attack will be captured by the opponent on the next move, resulting in a material loss. Moving the piece to safety (to a square where it is no longer under the opponent's attack or is protected by pieces of its own side) can be considered the equivalent of a forced move.

Another reason for using this *ad hoc* extension heuristic in *Abyss'* search algorithm stems from consideration of the repetition rule of Chinese chess. *Null moves* can be used to detect threats, but not all threats identified by the *null move* follow the rules of Chinese chess, and the expense of detection is high. Therefore, some simplification to restrict the type of moves which are considered "threats" is made in *Abyss*. As a byproduct of detecting an *attack-evading* move, we gain information distinguishing a legal repetition from an illegal one.

In *Abyss*, a simplified version of this heuristic is included; it only considers moving a major piece out of attack, since including all types of pieces proved to be too expensive. Two experiments were conducted: in the first, we determine whether there is at least one opponent's piece under attack after one side has made a move (including *discovering* attacks). In the second, we restrict ourselves to those pieces attacked during the last move by the opponent. Although the former implementation yields a better performance in terms of hit rate, we believe that it is not cost-effective since the search ratio is too high. Thus, for further experiments here, we consider only the pieces attacked by the piece that moved last.

There is still another heuristic which can be used for extensions: strictly-forced moves. This heuristic has been implemented in the chess-playing program *Touch*†. If there is only one legal move in a position, it is of course forced and a less turbulent and more reliable measure of the position may be found by extending the search with an extra ply. Moreover, such an extension may also be useful in situations where one side can make a move which leads to a decisive advantage (such as a mate threat) but the opponent can temporarily "thwart" this threat by some delaying move such as a check. Without these extension moves, one can be "fooled" into missing the threat. Note that positions where there is only one legal move are common when the King of the side to move is in check. Therefore a total of two extra plies will be searched when strictly-forced moves are combined with the *check-evasion* heuristic, allowing a deeper search in a forced line without incurring too much extra cost.

### 3.2. Combining Extensions with Domain-Specific Knowledge

The experimental results of different combinations of search extensions with domain-specific knowledge have been provided in Table 3, based on the key of extension combinations given in Table 4. To decide the relative importance of each extension heuristic (and their combinations), we must compare not only the performance

† Private communication, J.W.H.M. Uiterwijk, 3rd Computer Olympiad, Maastricht, The Netherlands, August 1991.

measures, but also the search efficiency. Since the importance grows as the hit % increases, but is inversely related to the node ratio, a function defined as hit % rate (H) over node ratio (R) was used to measure the relative importance (*RI*) of each extension heuristic (but note, R is computed relative to the node count of Experiment No. 1). From this an average weighted relative importance (*Ave. RI*) is computed, made up of the H/R values for depths 3, 4 and 5 in proportion of 1:2:5, respectively. This *ad hoc* biasing of the average RI to the results from the greater depths serves as a good discriminator between the various contributions of the knowledge extensions, as Table 3 shows.

**Table 3: Experimental Results for Different Knowledge Extensions**

| Expt. No. | Depth = 3 | | | Depth = 4 | | | Depth = 5 | | | Ave. RI |
|---|---|---|---|---|---|---|---|---|---|---|
| | nodes | R | H | nodes | R | H | nodes | R | H | |
| 1 | 104234 | 1.00 | 18 | 641469 | 1.00 | 30 | 1887356 | 1.00 | 30 | 28.50 |
| 2 | 121842 | 1.17 | 32 | 659071 | 1.03 | 38 | 2684653 | 1.42 | 50 | 34.65 |
| 3 | 111711 | 1.07 | 22 | 661337 | 1.03 | 28 | 2290755 | 1.21 | 34 | 26.93 |
| 4 | 144948 | 1.39 | 24 | 802759 | 1.25 | 30 | 2890065 | 1.53 | 34 | 22.05 |
| 5 | 125612 | 1.21 | 18 | 705624 | 1.10 | 32 | 2420428 | 1.28 | 34 | 25.73 |
| 6 | 107812 | 1.03 | 18 | 678275 | 1.06 | 30 | 2013468 | 1.07 | 36 | 30.29 |
| 7 | 127350 | 1.22 | 32 | 658167 | 1.03 | 36 | 3062946 | 1.62 | 48 | 30.54 |
| 8 | 155909 | 1.50 | 30 | 869377 | 1.36 | 42 | 4493750 | 2.38 | 52 | 23.88 |
| 9 | 167284 | 1.60 | 34 | 759998 | 1.18 | 38 | 3510887 | 1.86 | 50 | 27.51 |
| 10 | 136747 | 1.31 | 30 | 854828 | 1.33 | 42 | 3645802 | 1.93 | 50 | 26.95 |
| 11 | 167878 | 1.61 | 30 | 893620 | 1.39 | 40 | 5309546 | 2.81 | 52 | 21.09 |
| 12 | 166342 | 1.60 | 34 | 784635 | 1.22 | 36 | 4040137 | 2.14 | 46 | 23.47 |
| 13 | 144298 | 1.38 | 30 | 855107 | 1.33 | 42 | 4467591 | 2.37 | 50 | 23.80 |
| 14 | 214200 | 2.05 | 32 | 1033796 | 1.61 | 40 | 6248260 | 3.31 | 52 | 17.98 |
| 15 | 190384 | 1.83 | 30 | 1104580 | 1.72 | 46 | 6693856 | 3.55 | 54 | 18.24 |
| 16 | 195093 | 1.87 | 32 | 1140422 | 1.78 | 42 | 7364632 | 3.90 | 52 | 16.37 |
| 17 | 249328 | 2.39 | 32 | 1632788 | 2.55 | 46 | 11574022 | 6.13 | 58 | 12.10 |

**Table 4: Key to Different Combinations of Knowledge Extensions**

| Expt. No. | Combinations |
|---|---|
| 1 | no extensions (`+cap+tra+his+null+fcut`) (Table 2, line 4) |
| 2 | check evasions |
| 3 | recaptures |
| 4 | king threats |
| 5 | evading moves |
| 6 | strictly-forced moves |
| 7 | check evasions + recaptures |
| 8 | check evasions + evading moves |
| 9 | check evasions + king threats |
| 10 | check evasions + strictly-forced moves |
| 11 | check evasions + recapture + evading moves |
| 12 | check evasions + recapture + king threats |
| 13 | check evasions + recapture + strictly-forced moves |
| 14 | check evasions + recapture + king threats + evading moves |
| 15 | check evasions + recapture + strictly-forced moves + evading moves |
| 16 | check evasions + recapture + strictly-forced moves + king threats |
| 17 | check evasions + recapture + strictly-forced moves + evading moves + king threats |

### 3.3. Interpretation

Although the criterion used here to calculate the relative merits of the heuristics is crude, these statistics provide evidence to support use of certain extension heuristics. From Table 3, we see that check evasion is the heuristic with the highest relative merit, both among those with a single extension heuristic and also when the extensions are combined. Using more than two extension heuristics leads to far more nodes being searched without improving the success rate significantly. Also, we see that when pairing extension heuristics, the combination of check evasions plus recaptures performs best, although the other two combinations, check evasions plus king threats and check evasions plus strictly-forced moves, are not far behind (based on average *RI*s). Of course, the strictly-forced move heuristic alone costs little in node expansions, but does not in itself lead to an improved hit rate.

From our experiments in the application of domain-specific knowledge we offer three conclusions. First, check evasion (Experiment 2) always performs well and should be included in all chess-playing programs. Second, combining two extension heuristics yields better correct-move hits, but it is perhaps not worth going beyond two, since then the increase in nodes searched spoils the performance (e.g., Experiment 17). Third, when combining two extension heuristics, the best choice is to use check evasions plus recaptures (Experiment 7, already the most popular choice in chess-playing program design), although further experiments may be necessary to see whether the two other choices, check evasions plus king threats (Experiment 9) and check evasions plus strictly-forced moves (Experiment 10), also adequately improve the strength of play.

We note one final point. Although some of these extensions search more nodes by a factor of two or three (though with a much improved success rate), the node counts are still below those in the basic reference program (cf. Table 1). Thus, any combination of extensions is still effective relative to a program which uses captures-first as its sole move-ordering mechanism.

## 4. CONCLUSION

In this paper we have compared several basic search-extension mechanisms designed for computer-chess programs and applied them to the Chinese-chess variant, which has different board geometry, additional piece types and move restrictions and a different draw-by-repetition rule. Aside from providing useful quantitative data showing clearly the effect on nodes (positions) visited during tree searches to nominal depths of 3, 4 and 5 ply, so that the potential benefits of different combinations of heuristics can be clearly seen, the paper also shows how these heuristics can have both beneficial and harmful effects on the search quality (the success rate). Although the heuristics show the benefits of deeper searches, there comes a stage when some may lose there effectiveness (the transposition table and history heuristic, for example). Also, this study was done only on a node-count basis. Had it been possible to obtain accurate timing measurements, then no doubt implementation techniques would play a greater role and could lead to slightly different interpretations from those assuming that the time taken is directly proportional to the number of nodes (positions) visited during the search. Thus the results presented here are only a guide to the potential benefits of some search-extension heuristics, since their times of execution have not been measured. Some of the heuristics mentioned here might not be worth implementing at all when other search-extension heuristics, such as singular extensions, are included. One might note that a strictly-forced move is subsumed as a special case of a PV-singular move since the only move is also the significantly best move (Anantharaman *et al*., 1988). Whether the strictly-forced-moves extension becomes redundant when applying PV-singular extensions is beyond the scope of this paper.

## 5. REFERENCES

Anantharaman, T.S. (1991). Extension Heuristics. *ICCA Journal*, Vol. 14, No. 2, pp. 47-65.

Anantharaman, T.S., Campbell, M.S. and Hsu, F.-h. (1988). Singular Extensions: Adding Selectivity to Brute-Force Searching. *ICCA Journal*, Vol. 11, No. 4, pp. 135-143. Republished (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99-110.

Beal, D.F. (1989). Experiments with the Null Move. *Advances in Computer Chess 5* (ed. D.F. Beal), pp. 65-79. Elsevier Science Publishers, B.V., Amsterdam. Revised as Beal (1990). A Generalized Quiescence Search Algorithm. *Artificial Intelligence*, Vol. 43, No. 1, pp. 85-98.

Berliner, H.J. and Ebeling, C. (1989). Pattern Knowledge and Search: The SUPREM Architecture. *Artificial Intelligence*, Vol. 38, No. 2, pp. 161-198. Revised as Berliner, H.J. and Ebeling, C. (1990). Hitech. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J.Schaeffer), pp. 79-109. Springer-Verlag, New York.

Bettadapur, P. and Marsland, T.A. (1988). Experiments in Chess Capture Search. *Int. J. Man Machine Studies*, Vol. 29, No. 5, pp. 497-502.

Birmingham, J.A. and Kent, P. (1977). Tree-Searching and Tree-Pruning Techniques. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 89-107. Edinburgh University Press, Edinburgh.

Ebeling, C. (1987). *All the Right Moves: A VLSI Architecture for Chess*. MIT Press, Cambridge, MA.

Goetsch, G. and Campbell, M.S. (1990). Experiments with the Null-Move Heuristic. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 159-168. Springer-Verlag, New York.

Jacobs, N.J.D. (1989). Xian, a Chinese Chess Program. *Heuristic Programming in Artificial Intelligence 1: the first computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 104-112. Ellis Horwood, Chichester.

Kaindl, H. (1982). Dynamic Control of the Quiescence Search in Computer Chess. *Cybernetics and Systems Research* (ed. R. Trappl), pp. 973-977. North-Holland, Amsterdam.

Lau, H. (1985). *Chinese Chess*. Charles E. Tuttle Co. Inc., Rutland, VT.

Marsland, T.A. (1986). A Review of Game-tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19.

Newborn, M.M. (1975). *Computer Chess*. Academic Press, New York.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19.

Schaeffer, J. (1986). *Experiments in Search and Knowledge*. Ph.D. thesis, University of Waterloo, Waterloo. Also (1986) as *Tech. Rep.* 86-12, Computing Science Department, University of Alberta, Edmonton.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, Vol. 11, No. 11, pp. 1203-1212.

Scherzer, T., Scherzer, L. and Tjaden, D. (1990). Learning in Bebe. *Computers, Chess, and Cognition* (eds. T.A. Marsland and J. Schaeffer), pp. 197-216. Springer-Verlag, New York.

Slate, D.J. and Atkin, L.R. (1977). CHESS 4.5 - The Northwestern University Chess Program. *Chess Skill in Man and Machine* (ed. P.W. Frey), pp. 82-118. Springer-Verlag, New York.

Tsao, K., Li, H. and Hsu, S. (1990). Design and Implementation of a Chinese Chess Program. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 108-118. Ellis Horwood, Chichester.

Ye, C. (1992). *Experiments in Selective Search Extensions*. M.Sc. Thesis, Computing Science Department, University of Alberta, Edmonton.

Zobrist, A.L. (1970). A New Hashing Method with Applications for Game Playing. *Tech. Rep.* 88. Computer Sciences Department, University of Wisconsin, Madison. Reprinted (1990) in *ICCA Journal*, Vol. 13, No. 2,

pp. 69-73.

## APPENDIX

### 1. The Repetition Rule in Chinese Chess and its Effects

Chinese chess has a significantly different repetition rule from that for chess. For instance, repetition by perpetual check is considered a draw in chess, but such a repetition is not allowed in Chinese chess. In general, the rules of Chinese chess disallow the use of certain repetitions after a threat move (even so there are exceptions). Three types of moves are considered as threats: checking moves, moves that threaten to win material, moves that threaten mate. Nevertheless, the rule allows certain repetitions via a threat, provided the current position is a repetition and is reached by a threat move as well (again there are exceptions to this).

Since the repetition rule of Chinese chess is so complicated, none of the current Chinese chess programs can claim that they handle all situations correctly. Tsao *et al*. (1990) proposed a means for their program *Chess Master* to handle most of the commonly occurring situations. The commercial Chinese chess program *Xian* (Jacobs, 1989) guarantees never to make an illegal repetitive move, but still lacks the knowledge to handle cases when a repetition would be legal, and in some cases it allows the opponent to make an illegal repetitive move. Another program, *Surprise*, a participant at the 3rd Computer Olympiad, allows certain illegal repetitions when it finds that all alternatives are significantly worse (as evidenced by some pre-tournament testings and casual plays against *Surprise* during the 3rd Computer Olympiad).

### 1.1 A Repetition-Check Algorithm

In *Abyss*, we tried a more general repetition-detection algorithm, differentiating between detection at the root and at interior nodes. The rationale behind this is to use a more strict rule for the root node and a more lenient one at internal nodes.

For internal nodes, some approximation is made and only check repetition and some simple piece-winning threats are considered. Two positions are considered identical if their transposition-table *locks* (Zobrist, 1970; Marsland, 1986) are the same. A stack (sequential table) is used to store all such *locks* from the first move in each game, but no count of the number of repetitions is kept. If a repetition under this definition is detected, we determine not only whether the move reaching this position is a check, or a threat to win a lone piece, but also that the previous move is not such a simple threat. If the preceding move was not a threat, then a threatening move leading to a repeated position is assigned an *illegal* score (almost as poor as a *mate* score). Any other combination of moves to a repeated position is given a *draw* score. In both cases there is no further search. The reason for using an *illegal* score is because the definition of repetition here is *approximate*, and so we might miss the only possible defense if an *illegal* score is no better than a *mate* score.

Nevertheless, at the root node an illegal move will be disallowed, so a more strict repetition-check algorithm is adopted. For each move being considered at the root, we check backwards to see if this position is being repeated for the third time. If so, and the move that reaches this position falls in the category of *threat* (defined below), we backup to see if the previous position is also a repetition (not necessarily for the third time) and whether the move reaching that position is also a *threat*. If the test of the second position fails (the position is not repeated or the move to it is not a threat), we assign the value of the current move as *forbidden*, a score worse than *mate*. In all other cases when a threefold repetition is detected, a *draw* score is assigned. We think such an approach can correctly handle many of the difficult repetition conditions in Chinese chess. Also, it provides a conceptually sound base for further improvements and should be able to encompass even more repetitions as the definition of a threat is refined.

### 1.2 The Definition of Threat

In *Abyss*, a *threat* is defined as a:

*Checking threat*

If a move delivers check, it is a threat. This is the simplest case.

*Mate threat*

If after one side has made a move, the opponent can be mated by a series of checking moves, the first move is considered to be a *mate threat*. In *Abyss*, an approximation is adopted. We search to a depth of 3 ply, considering only checking moves and replies to check, and assume that there is a mate if a mate score is returned for this search. To reduce the search cost to a reasonable level, detection of a mate-in-$n$ ($n > 3$) threat is not considered at present, although according to rules, all mate threats should be equally treated. For faster hardware, however, it may be better to search to unlimited depth, terminating only when a repetition (of any kind) or a mate is seen or when there are no more checking moves.

*Piece-winning threat*

Again some simplifications are made, and we consider only those moves that threaten to win an unprotected piece (all minor and major pieces and Pawns beyond the river). The expense of *threat* detection is not as large as it seems, because the operations above are carried out only when a third-time-repetition is seen, and the test is only done once during the search.

Figure A1 summarizes how we distinguish a legal repetition from an illegal one. Further work is required to consider even earlier positions once a repetition is found. At the moment we can handle some difficult repetition situations, e.g., two threats over two threats (a draw) or two threats over one threat (a loss). Also, because of time limits, search to some predefined fixed depth might be required to detect whether a side has a *piece-winning threat*. By restricting the moves to only captures, checking moves and replies to checks (an extended quiescence search), we can do a search after making a null move (in this case making two consecutive moves for the side causing the repetition). If the value returned exceeds a certain amount (the *threat margin*), the first move can be thought of as a *threat* and can be treated accordingly.

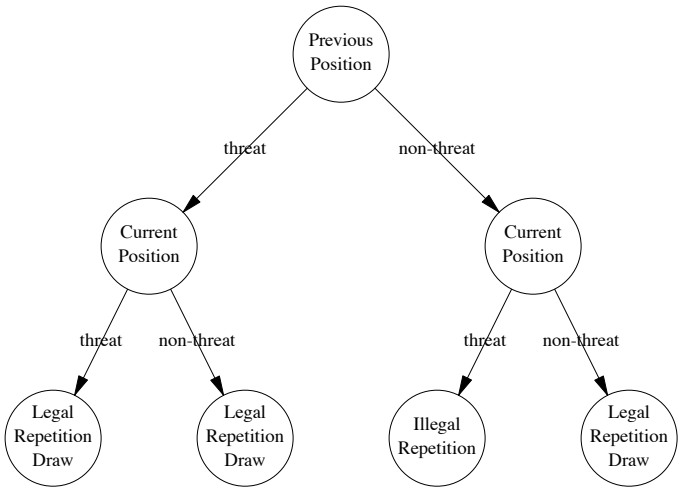


**Figure A1:** Legal and Illegal Repetitions.