Improving the null move heuristic invocation in chess with a data-driven approach

© May 2016 Andrew C Lea, MA (Cantab), FBCS

Heuristics are often used in AI applications that must contend with a large search space, such as those used in Chess programs to reduce the number of nodes that must be searched. As heuristics are inexact, non-optimal solutions can be returned, and in adversarial search increase the number of nodes searched, even if on balance it represents a net saving. In this paper we describe a data-mining technique to deriving a trigger-rule which determines when such a heuristic should be invoked, applying it to the null move heuristic in Chess. As a result, the null-move success ratio increased from 1 in 4 to 4 in 5.

1 Introduction

Heuristics – inexact algorithms – are used extensively in Artificial Intelligence, frequently because a search space to be explored is prohibitively large. It is, however, in the nature of heuristics that whilst they generally improve results by finding good solutions, they can impair results by alighting on local optimums which are poor globally. Even if the ratio of improvement to impairment is good, in some applications the very high number of invocations can guarantee that that a damaging number of the results will be impaired. Ideally we would like to know in advance of each heuristic call whether it will probably prove beneficial, so that we can avoid it when not. In some situations the quality of a heuristic result can be evaluated post-invocation, but even then we have still spent CPU cycles on a nugatory invocation. Finally we normally require our invocation heuristic to be cheaper than the heuristic itself.

This paper, using Chess as the test-bed, shows how data mining can be used to find simple and fast to execute trigger-rules as to whether a particular heuristic – in this example the null-move heuristic – would prove beneficial in adversarial search. This is particularly valuable since whilst powerful, the null-move heuristic is relatively costly in CPU cycles.

This technique is of general application to those many AI programs which have large search spaces to explore. As deep AI develops on ever larger datasets, this will remain a significant challenge and, in some cases, the ability to explore large search spaces may be the rate-limiting step of progress.

Since Chess is the test-bed, the first section of this paper describes how typical Chess programs operate, and the next section describes the null-move heuristic. We then describe how the trigger-rule, which determines whether the heuristic should be invoked or not, is mined by analysing the running of an instrumented Chess program; derive an actual trigger-rule; and finally consider how successful the derived rule is.

2 Chess Programming Background

This section is an overview of the framework into which the null-move heuristic sits.

2.1 Static Value

Most chess programs represent a position's merit with a positive number for good White (and bad Black) positions, negative for good Black (and bad White) positions, near zero for neutral positions, draws, or stalemates. There is generally a *static evaluation function* which evaluates a position without exploring move possibilities and returns its *static* value.

2.2 Looking ahead and mini-max

To temporarily over-simplify, to choose a move, players generally consider each possible move, look at each possible opponent's response, how one might reply to that, and so on. Most chess programs use the mini-max[1] approach to mimic this. The program assumes that White is attempting to maximise the value of a position, and Black to minimise it.

Suppose White is to play. The program looks at all White's moves, considers all Black's responses, considers all White's responses to that move, and so on down to some fixed depth, recursively building a conceptual game tree [2] of moves[1]. At this point it stops, uses the static evaluation function to decide how good the position is, and backs the value back up the tree, assuming always that White selects the most maximising move, and Black the most minimising. At the top level, it plays White's move which has the highest value.

The branching factor – the number of possible moves at each position or sub-position - of this approach is too great to make deep exploration practical. It starts at 20 and grows until the end-game when, because many pieces have now been taken, it drops back down.

2.3 Alpha-Beta

The Alpha-Beta procedure[3] is normally used to reduce the branching factor[4] and the number of positions that need to be looked at, but the number of possibilities still grows exponentially, limiting the depth to which the program can explore.

As an algorithm[5] Alpha-Beta does not change the answer generated. When the best moves are examined first, it greatly reduces the branching factor. Alpha-beta maintains two limits, which represent at any tree node the best that White is guaranteed, and the 'worst' that Black is guaranteed. If a backed up score falls outside this window, the rest of the node's moves are ignored, because they can not yield a better position.

2.4 Heuristics

To repair our earlier over-simplification, we observe that when experts play chess, although they may look ahead many moves, they only consider the most important moves, somehow sub-consciously disregarding trivial moves. Because of this, their branching factor is very low, and the tree of moves they need to explore is thin and deep.

We would like to replicate this with AI, but to do so means developing other techniques – perhaps pattern recognition [6]. These techniques are heuristics, as they may not return the correct answer: they might even prune away good moves, and lead to worse play. Effective techniques are known to improve searching: quiescent search and extensions[7] extend the search in tactical positions, or to reduce the branching factor of the game tree, such as the killer move heuristic [8], and the history heuristic [9]. In this paper, we look at one specific heuristic, the null move heuristic.

---

[1] The entire tree never exists at once in the computer's memory, only the branch currently being explored.

3 The Null Move Heuristic

3.1 Concept

When looking at a position down in the game tree, if without even making a move the side to play still does well, there is no need to explore much deeper, since this is clearly a good position. Of course, this would not actually be legal chess and worse yet, under some circumstances (zugzwang, when the side to move loses) yields exactly the wrong answer. (Verified null-move pruning [10] is a technique to avoid this problem.)

This is the null-move heuristic [11], and is used like this. Rather than look at the moves at his disposal, the side to play first explores the game tree without making a move (the null move) and to a depth two or more less than would otherwise be the case, thus making a potential saving in the number of tree nodes explored. If that search results in a value outside the alpha-beta window, then the moves available need not be explored. Since at least one of them would have been explored, and to the full depth, this yields a significant saving, but only when the null-move heuristic is successful.

3.2 Implementation

Pseudo-code to illustrate the null-move heuristic on White's turns is given below. In it we can see the null-move heuristic triggering 'cut-offs', that is curtailing the search:

```
NULL_MOVES = 2
NullMoveInvokedCounter = NullMoveSuccesfulCounter = 0

int AlphaBetaWhite(depth, alpha, beta, board)
      // Returns board value on white's go. AlphaBetaBlack is symmetric.
      // Null-move heuristic
      if not AdverseCircumstances(...) then
            Inc(NullMoveInvokedCounter)
            score = AlphaBetaBlack(depth-1-NULL_MOVES, alpha, beta, board)
            if beta <= score then
                  Inc(NullMoveSuccesfulCounter)
                  return score
      // Full scale alpha-beta search
      foundLegalMove = false
      for move in allPossibleLegalMoves(board) do
            foundLegalMove = true
            victim = MakeMove(move, board)
            If depth > 0 then
                  alpha = max(alpha,
                              AlphaBetaBlack(depth-1, alpha, beta, board))
            else
                  alpha = max(alpha, StaticValue(board))
            UndoMove(move, victim, board)
            if beta <= alpha then
                  return alpha
      next move

      if foundLegalMove then
            return alpha
      else if WhiteInCheck(board)
            return CheckMateByBlack
      else
            return Stalemate
```

Generally the null move heuristic is not invoked in adverse circumstances, such as (in the case of the chess engine used in this paper[2]):

- in check
- near the game root
- more than once in any one branch, since search depth would be cut too significantly
- in quiescent search or in search extensions, because it makes little sense to extend the search and immediately cut it back.

Other Chess programs will have different, though similar, initial criteria. These rules are driven by Chess and programming considerations, rather than by data mining.

3.3 Disadvantages

The null move heuristic is expensive in that it involves exploring the remainder of the tree, albeit to a depth of two or more less, and if it does not demonstrate that the rest of the moves need not be searched, then searched they must be. In this case, we have increased the work to be done, not reduced it. In the instrumented Chess engine, the null move counters indicated that the null move heuristic was generally not successful.

3.4 The Null Move Heuristic Trigger Rule

We would like to use the heuristic only when it will probably save us time, by returning a value indicating there is no need to search all the other moves in depth. In other words, a heuristic or rule to trigger the null-move heuristic only when it would probably be beneficial. In the next section we data-mine that trigger rule.

4 Data Mining the Null Move Trigger Heuristic

An instrumented chess engine was equipped with the null move heuristic, and whether or not the heuristic triggered a cut-off was logged. This log was then mined to discover rules which could best predict whether or not the null-move heuristic would trigger a cut-off.

4.1 Data Mining the Trigger Heuristic: Experimental Procedure

For each interior game tree node the following was logged for subsequent analysis:

1. Any reasonably cheaply obtainable parameter which might be of use to the heuristic, such as:
   - colour: of the piece last moved,
   - ply: how deep the algorithm has recursed, where one 'ply' is one player's move, as a turn is traditionally one white move and one black move,
   - alpha and beta: highest and lowest scores encountered so far, used to optimise the mini-max algorithm,
   - the destination square of last move,
   - last moved piece value,
   - material static value.
2. The value returned from the null move heuristic.
3. Whether or not the null-move heuristic indicated (by returning a score less than beta for white) that a full search was needed. This is not the same as indicating if a full search was actually needed, which would be a different experiment: our purpose here is to mine cheap indicators as to whether or not our already trusted heuristic was likely to indicate a full search could be avoided.

This logging is illustrated in bold below:

---

[2] ImpChess is written by the author, and is so-called because of its extremely small footprint.

```
int AlphaBetaWhite(depth, alpha, beta, board)
      // Null-move heuristic
      if not (WhiteInCheck(board) or OtherAdverseCircumstances()) then
            Log(depth,alpha,beta...) // values the trigger-rule might use
            Inc(NullMoveInvokedCounter)
            score = AlphaBetaBlack(depth-1-NULL_MOVES, alpha, beta, board)
            Log(score, beta <= score) // would the heuristic cut of game
            if beta <= score then // tree exploration?
                  Inc(NullMoveSuccesfulCounter)
                  return score
      // Full scale alpha-beta search as before…
```

Two example log entries are shown below. In the first entry, the heuristic determined that a cut-off should not occur, and in the second it had determined that it should:

Colour	Ply	Alpha	Beta	Last move destination	Piece value	Position static value	Heuristic score	Cut-off
white	3	0	2	d5	384	0	-127	no
white	3	0	2	d7	384	0	2	yes

To generate the data, ImpChess was set to play itself on levels 4, 5, and 6. Because of move extensions, the program often reached ply 12 on level 5, giving good depth coverage, playing 87 moves until White checkmated Black. The analysis program read the log and calculated that the null-move heuristic yielded a White cut-off on 28% of invocations, and for Black on 27%.

4.2 Data Analysis

The aim of the data-mining is to find a rule which partitions the data using the ‘cheap' parameters into two classes: one in which the heuristic indicates a cut-off, and the other in which it does not. If it were possible to perfectly partition the data, then the heuristic would no longer be needed, since it could be replaced with the cheap rule. This approach is similar to dividing a data set in inducting a decision tree[12].

For example, one candidate rule might be alpha > beta, which has two possible partitions (true and false). Analysing a log file for a game run to depth 6, yields:

Partition	No entries	No cut-offs	% cut-offs
FALSE	30,245,767	7,069,823	23%
TRUE	1,478,899	873,027	59%

This means that where alpha > beta (second row), the null-move heuristic caused a cut-off on 59%, but where it did not, there was only a null-move cut-off on 23% of occasions. This could form the basis of a moderately useful trigger-rule.

Some partitionings have more than two partitions. For example, the partition profile on piece value has four (Pawn, Knight and Bishop, Rook, and Queen):

Partition	Pawn	Knight or Bishop	Rook	Queen
No entries	6,514,280	9,806,487	9,100,365	6,303,534
% cut-off	28%	22%	27%	20%

This set of partitions would not form a useful rule, since the percent of cut-offs in each partition are too similar.

The analysis program was designed to test a variety of rules in this way. The output of the analysis program for three much more interesting rules reads:

```
Calculating profile for material+/-piece outside a-b range, which has 2
entries:
false: 4,829,907 / 27,871,354 = 17%
true: 3,112,943 / 3,853,312 = 80%

Calculating profile for a-b > 1000 AND material+/-piece outside a-b
range, which has 2 entries:
false: 7,488,669 / 31,161,719 = 24%
true: 454,181 / 562,947 = 80%

Calculating profile for material+/-pawn outside a-b range, which has 2
entries:
false: 1,935,167 / 22,925,347 = 8%
true: 6,007,683 / 8,799,319 = 68%
```

These rules yield the opportunity to invoke the null-heuristic with a probability of successful cut-off of 80%. The use of a rule derived from these are described in the next section. Fortunately, the same rule emerged from analysing the log of ImpChess running at levels 4, 5, and 6, so the rule is not strongly level dependent.

4.3 Derived Rule

The fully derived rule is shown in bold in the context of the Chess program below. The rule terms are inexpensive to execute compared to the cost of the null move heuristic.

```
int AlphaBetaWhite(depth, ply, alpha, beta, nullMovePermitted, board)
      // Null-move trigger heuristic
      if ply > 1 and depth < 0 and NullMovePermitted
            and LastPieceMoved <> King and beta-alpha <= Infinity
            and MaterialValue(brd) > beta + PawnMaterialValue
            and not WhiteInCheck(board) then
            // Null-move heuristic
            Inc(NullMoveInvokedCounter)
            score = AlphaBetaBlack(depth-1-NULL_MOVES, ply+1, alpha, ...)
            if beta <= score then
                  Inc(NullMoveSuccesfulCounter)
                  return score
      // Full scale alpha-beta similar to before...
```

The mined terms of beta-alpha <= Infinity and MaterialValue(brd) > beta + PawnMaterialValue are both understandable in terms of Chess programming, and illustrate that this approach may yield insight into the problem space.

4.4 Analysis Code

The analysis procedure was written in Python. Initially it reads in the log, and reduces the many duplicate lines (which naturally and correctly appear since the same position can be reached via many routes) to counts, simply to reduce memory consumption and increase the analysis speed.

For analysis, candidate rules are expressed as lambda expressions. The following rule would create two partitions (true and false):

```
lambda r: r.beta - r.alpha < PawnValue
```

This rule would create three partitions, with possible value -1,0,1:

```
lambda r: 1 if r.beta > r.alpha else -1 if r.beta < r.alpha else 0
```

For each candidate rule, the Python produces a mapping of each partition to its heuristic cut-off success ratio. In the code below a dictionary is a Python mapping, and a defaultdict(int) is essentially a counter. Python to produce each mapping, similar to that used, is shown below:

```
def MapPartitions(data, rule):
    "Returns a mapping of partitions to success ratio for this rule"
    # Pass one: get the set of partitions
    partitions = set()
    for line in data:
        partitions.add(rule(line))
    # Partitions now contains a set like {pawn,knight,bishop} or {0,1}
    # Pass two: for each partition, count how often it occurs,
    # and how often there is a cut-off
    cutoffs = defaultdict(int)
    total = defaultdict(int)
    # Calculate success ratio for each partition
    for line, count in data.items():
        partition = rule(line)
        if line.cutoff:
            cutoffs[partition] += count
        total[partition] += count
    return {entry: cutoffs[part]/total[part] for part in partition}
```

5 Experimental Results

Equipped with the derived rule, ImpChess was run at depths 4, 5, and 6. The experimental results are shown in the following table, with the impact of the trigger heuristic rule is shown in the last column:

Level	Moves	Result	Log file mBytes	Entries	White null-move cut-offs	Black null-move cut-offs	Triggered cut-offs
4	254	draw	44	1,520,035	24%	22%	85%
5	87	mate	310	10,421,385	28%	27%	82%
6	100	draw	885	31,724,665	22%	27%	80%

Without the trigger-rule, the null-move heuristic caused a cut-off in approximately 1 in 4 occasions, so on 3 in 4 occasions the work of the null-move heuristic was nugatory. With the null-move heuristic only being triggered according to the trigger-rule, it caused a cutoff on 4 out of 5 occasions, so that there were far fewer wasted (and expensive) invocations of the heuristic.

This trigger heuristic, combined with the null-move heuristic it governs, allows ImpChess to search approximately two ply (that is one turn) in the same time – a worth-while improvement. The trigger heuristic also appears to prevent the null-move heuristic being used when it is likely to return a wrong result, although this has not yet been verified.

6 Conclusions

Data-mining can be used to identify, and measure the success of, trigger rules to determine whether or not to invoke heuristics, leading to greater speed, by only invoking heuristics when they are likely to be beneficial.

Heuristics in complex environments can potentially be improved through data mining. This is especially useful since sometimes the thresholds the heuristics should use can be counter-intuitive or highly dependent on the details of the rest of the system, and therefore subject to change if other details of the system, including the heuristic itself, change. Such data mining may also yield insight into the problem space, as here.

7 Further Work

The planned next steps are to:

1. Instrument the chess engine to compare the heuristic's recommendation with a full, expensive, evaluation. This would enable the derived trigger-rule to avoid the heuristic when it was likely to be wrong in recommending a cut-off, and therefore increase heuristic accuracy.
2. Use decision tree induction to generate rules with greater precision.

References

[1] Claude Shannon, “Programming a Computer for Playing Chess”, 1949

[2] Ronald Rivest, “Game tree searching by min/max approximation” 1987, Artificial Intelligence

[3] Donald Knuth, Ronald Moore, “An analysis of alpha-beta pruning” 1975, Artificial Intelligence

[4] Gerard Baudet, “On the branching factor of the alpha-beta pruning algorithm” 1978, Artificial Intelligence

[5] Andeas Blass and Yuri Gurevich, “Algorithms: A Quest for Absolute Definitions”, 2003, Bulletin of European Association for Theoretical Computer Science

[6] Christopher Chabris and Eliot Hearst, “Visualisation, pattern recognition, and forward search: effects of playing speed and sight on the position on grandmaster chess errors”, 2002, Cognitive Science

[7] Kaindl “Searching to Variable Depth in Computer Chess” 1983, Proceedings of IJCAI 83

[8] Selim Akl, Monroe Newborn “The Principal Continuation and the Killer Heuristic.” 1977, ACM Annual Conference Proceedings

[9] Jonathan Schaeffer “The History Heuristic”, 1983, ICCA Journal, Vol. 6, No. 3

[10] David-Tabibi and Netanyahu, “Verified Null-Move Pruning”, 2002, ICGA Journal, International Computer Games Association

[11] Goetsch and Campbell, “Experiments with the Null-Move Heuristic”, 1990, Computers, Chess, and Cognition

[12] Safavian and Landgrebe, “A survey of decision tree classifier methodology”, 1990, NASA Technical Report