The Graph-History Interaction: On Ignoring Position History

Murray Campbell

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Game playing programs make great use of the fact that the trees they search are more often graphs, containing identical nodes arrived at by different means. By saving search results in a hash table, or maintaining a graph rather than a tree, it is often possible to avoid duplication of effort. However there is a potentially harmful interaction that occurs when such techniques are used in a game that allows repetition of position. This papers describes the Graph-History Interaction, and suggests practical solutions to the problems it causes.

1. Introduction

The so-called *game trees* searched by most current programs are more properly termed *graphs*; many identical positions exist in various branches of the tree. There are two major techniques used in today's programs to take advantage of this fact: hash tables (used in full-width programs), and subtree merging (used in programs that maintain the whole tree). Hash tables in particular have proven extremely successful, typically giving speedups of 2 or more [4].

For efficiency reasons, programmers often take liberties with the definition of 'identical positions'. In many games, the identity of a position depends not only on the current configuration, but also on the history leading up to it. In chess, enpassant and castling status are considered by most programs as part of a position's identity. But other history dependent information is not used. In referring to Chess 4.5, Slate and Atkin [4] write

> Strictly speaking, positions reached via different branches are rarely truly identical, because the 50-move and three-time repetition draw rules make the identity of a position dependent on the history of moves leading to that position. This effect is small and we decided to ignore it.

This paper gives an example of a position where ignoring previous history gives an incorrect result (the Graph-History Interaction, or GHI), describes the general case of GHI[1], and shows that in the worst case there is no efficient solution. Some possibilities to reduce the occurrence of the problem are suggested.

2. An Example

Consider the position in Figure 2-1, being searched by a full-width, alpha-beta program with a hash table.

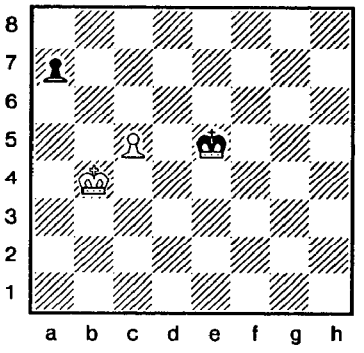

Figure 2-1: White to Play

After

1. Kb5? Ke6?
2. Ka6? Kd5
3. Kb5 Ke6

the position after move 1 has been repeated. As white has nothing better than 3. Kb5, the position after 2. Ka6 is stored in the hash table as a draw[2].

Now in the line

1. Ka5! Ke6
2. Ka6!

[1] This problem is found in all two-person games where repetition of position is possible.

[2] Throughout this paper we are dealing with repetition *within* the search tree, not to be confused with repetition with positions previously reached in an ongoing game. In the former case, a repetition is equivalent to lack of progress by either side, and is immediately scored as a draw. In the latter case, triple repetition is necessary before allowing a draw to be scored.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1985 ACM 0-89791-170-9/85/1000-0278 $00.75

reaches the identical position as above, incorrectly taking the draw value from the hash table, and missing the win after 2 ... Kd5 3 Kb5 Ke6 4 Kc6! Thus the program will fail to find the win with 1. Ka5, even though it is searching deeply enough to find it. This particular position does not cause problems for most depth-first programs, because iterative deepening avoids the difficulty. For selective search programs, however, the problem is a serious one.

3. The General Case

In Figure 3-1 we see the prototypical situation, based on an example given by Palay [3], where ignoring the position history leads to an incorrect search result. There are two possible scenarios in searching this tree.

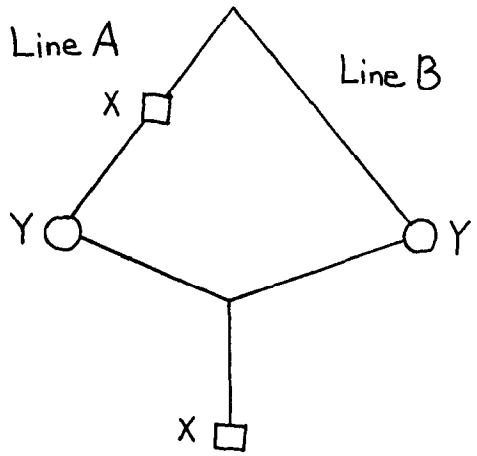

Figure 3-1: Prototypical case of GHI

1. Case Draw-First: Line A is searched first. Position Y is given a value dependent on the lower position X being a draw. When Y is reached down line B, the stored value is used, while the actual value may be altered by the fact that position X is not a draw.

2. Case Draw-Last: Line B is searched first. Position X is not considered a repetition, and position Y is given some value v. When Y is reached down line A, the stored value is used, even though the actual value may be altered by the draw that will be scored at the lower position X.

GHI only arises in searches of 5 ply or more. In shallower searches there are not sufficient moves to repeat the position (minimum of 4 ply) *and* have a position history.

It is possible to characterize the values returned by a search done when GHI exists.

Theorem 1: Let T_2 be an arbitrary game tree. Let T_1 be identical to T_2 except that T_1 has one or more node values set to 0 (the draw score), and the subtrees below these nodes are deleted. Then

1. If $V(T_1) < 0$, then $V(T_2) \leq V(T_1)$

2. If $V(T_1) = 0$, then there are no restrictions on the value of T_2

3. If $V(T_1) > 0$, then $V(T_2) \geq V(T_1)$

where V(T) gives the minimax value of tree T.

A sketch of the proof for case 1 runs as follows: Let $V(T_1)$ be $v < 0$. Consider the tree T_{min} which is that part of T_1 necessary to prove the value v (i.e. the tree that would be searched by alpha-beta given perfect ordering). At nodes with Max (the maximizing player) to move, the maximum of all alternatives is at best v (< 0). Thus none of these nodes could have value 0. At nodes with Min to play, the minimum of all alternatives is v. If one of the alternatives has value 0, the only change that would allow it to affect the value of the parent is to create a new minimum (since we are at a minimizing level). Thus T_2, which does not contain any artificial draw nodes, has value $\leq v$.

Proof of the other cases is similar.

Theorem 1 essentially states the restrictions on using a value returned by a search when the subtree contains a repetition of a position prior to the search. There is a companion to this theorem, which involves inverting the roles of T_1 and T_2. As will be seen later, Theorem 2 is much less useful than Theorem 1.

Theorem 2: Let T_1 be an arbitrary game tree. Let T_2 be identical to T_1 except that T_2 has one or more node values set to 0 (the draw score), and the subtrees below these nodes are deleted. Then

1. If $V(T_1) < 0$, then $V(T_1) \leq V(T_2) \leq 0$

2. If $V(T_1) = 0$, then $V(T_1) = V(T_2)$

3. If $V(T_1) > 0$, then $0 \leq V(T_2) \leq V(T_1)$

From these two theorem can be derived the following results:

- Case Draw-First does not present difficulties for a depth-first search if the value established by line A is less than the draw value.

- Case Draw-Last does not present difficulties for a depth-first search if the value established by line B is greater than or equal to the draw value.

4. Solutions

The problems that arise when ignoring position history were first described in Palay's thesis [3]. The difficulty arose in the context of a B* [1] program that maintained the entire tree that was being searched. Identical nodes reached via different paths had their subtrees merged, setting the stage for possible GHI. Since GHI situations only arose in 3 (out of 300) test positions, Palay did not implement a solution. Since abandoning a graph representation for a tree is unacceptable for performance reasons, Palay suggested the continued use of a graph, trying to detect the problem situation. When GHI is recognized, the graph is split into two independent sub-graphs at the same point that they were originally merged. Another method, used by Thompson[3], selects the next node to expand by doing a full tree search (i.e. with history). Nodes are evaluated without using history.

Things are much more difficult in depth-first programs that only maintain a tree in a hash table. Let us consider each case separately.

[3] Ken Thompson, personal communication, 1985

Case Draw-First: The program can detect when it is within a draw cycle, and avoid storing the history dependent information in the hash table (a simple implementation of such a scheme is sketched in the next section). Only a small number of positions are involved (a draw cycle of 4 ply in length only affects 4 positions), so very little information is lost. But the results of Theorem 1 say we can do even better than this. While within a draw cycle, the following set of rules can be used to store values in the hash table[4]:

- values less than the draw value can be stored, but only as upper bounds
- values greater than the draw value can be stored, but only as lower bounds
- values equal to the draw score can never be safely stored
- if a value to be stored is supposed to be a (say) lower bound, but the above rules state it is an upper bound, the value cannot be safely used

Case Draw-Last: There appears to be no way to detect this case. The search of line B has no reason not to store it's value of position Y in the hash table. When line A runs across position Y, the value in the table appears perfectly acceptable, and line A is scored incorrectly. Now we see why Theorem 2 is not useful. It is not possible to know in advance when the theorem is applicable.

David Slate's current program, Nuchess, adopts the simple technique of never storing a value equal to the draw score[5]. The intuition here is that difficulties most likely arise when the drawing line is on the principal variation. This would have sufficed to deal with the position in Figure 2-1. However this method does not solve Draw-First situations in general. In addition, genuine scores equal to the draw score (by coincidence, or through stalemate or 3-fold repetition with actual positions already reached (i.e. outside this particular search)) are not saved.

In some informal tests, the following four versions of a chess program were tested:

1. no action taken
2. draw scores not stored
3. no values stored on draw cycle
4. values stored in draw cycle according to full rules described above

Each of the last 3 methods would deal with the difficulty that arose in Figure 2-1, although option 2, as noted above, does not solve the general case of Draw-First GHI. No significant difference was noted between any of the programs on a number of test positions, though option 2 performed less well in one position where a large number of nodes had value 0 simply because the evaluation function scored the positions as even.

[4]These rules assume a minimax-style implementation. Negamax-based programs [2] must be suitably altered.

[5]David Slate, personal communication, 1984

5. Implementation

It is very inexpensive to implement options 3 and 4 in an alpha-beta based program. Whenever the current position at depth **CurrentDepth** is a repetition of a position at depth **OriginDepth**, all the positions currently on the search stack at depths **OriginDepth** through **CurrentDepth** inclusive must be flagged in some way. In flagged positions, either no value is stored in the hash table (option 3), or the value is stored subject to the previously mentioned rules (option 4). It should be noted that if the origin of a draw cycle is at depth 0, i.e. the root, GHI cannot occur (as there is no history prior to the cycle). In this case it is not necessary to flag the positions on the draw cycle.

This scheme correctly deals with the Draw-First case of GHI, and requires minimal space and time overhead.

6. Conclusions

The decision to ignore parts of a position's history were consciously made in order to allow hash tables to work effectively. Even though deeper searches (i.e. at least 5 ply) are becoming more common as programs and technology improve, GHI does appear to occur relatively infrequently. The key in avoiding most occurrences of GHI appears to be iterative deepening.

Of the two types of GHI, Draw-Last remains with no solution in depth-first programs. Complicated solutions for the Draw-First case are unlikely to be acceptable in programs striving for maximum nodes per second. Tests show that both options 3 and 4 are easy to implement and have minimal impact on program performance, while guaranteeing correctness.

GHI problems occur much more frequently in selective search programs, and require some solution in order to achieve reasonably general performance. Both Palay's and Thompson's approaches seem to be acceptable.

References

[1] Berliner, Hans J.
The B* tree search algorithm: A best first proof procedure.
Artificial Intelligence (12), 1979.

[2] Knuth, D. and Moore, R.
An analysis of alpha-beta pruning.
Artificial Intelligence 6:293-326, 1975.

[3] Palay, Andrew J.
Searching with probabilities.
PhD thesis, Carnegie-Mellon University, 1983.

[4] Slate, D. and Atkin, L.
CHESS 4.5 - The Northwestern University chess program.
In Frey, P. (editor), *Chess Skill in Man and Machine*,
chapter 4. Springer-Verlag, 1977.