Documentation for WinBy

Alan Roytman and John Lo

2006-12-14

Contents

1 Overview

WinBy is a solver and gameplay feature where the perfect computer tries to maximize the winning score or (if it is a losing position) minimize the number of points the opponent receives. This is in contrast to the typical mode of play, where the perfect computer either wins as quickly as possible or loses as slowly as possible (i.e., remoteness). Currently, WinBy is implemented for the standard solver along with the variable slices aware solver. If the module uses the goAgain solver, the WinBy feature will not work. In addition, the module should use generic_hash so the solver is aware of whose turn it is while it is propagating values up the game tree. For all games that support WinBy, the user may decide whether he or she wants the computer to play according to the WinBy optimization or the remoteness value stored in the database. Should the player desire to know the WinBy values for all moves from a given position, it has been added to "ps" output and can be viewed during gameplay. Currently, WinBy is implemented for mothello and miceblocks.

2 Design Decisions

WinBy values are of type WINBY and are type defined to be integers. This is important, because the values are allowed to be negative! In fact, WinBy is *defined from player one's perspective*. This has a few consequences for both the module and the solver. For example, WinBy values for Othello are always calculated as (# black pieces - # white pieces), no matter whose turn it is or whether the value is positive or negative.

The ramifications for the solver come into play when WINBY values are being propagated up the tree. If it is player one's turn for a given position, player one wants to *maximize* over all possible WINBY values of that position's children (i.e., get as close to positive infinity as possible). On the other hand, if it is player two's turn for a certain position, then that player wants to *minimize* over all possible WINBY values of that position's children (i.e., get as close to negative infinity as possible). These are two properties that follow from the definition of WinBy. As a result, the solver must know whose turn it is before it can do any bubbling up intelligently. This is precisely the reason the module needs to use generic_hash, since the solver cannot extract whose turn it is from the hashed POSITION.

The module's WinBy function is only called on positions which are primitives. All other positions have their WinBy values defined recursively. For example, if the initial position has a WINBY of 8, then this implies that there is a path from root to leaf where all intermediate nodes and the leaf node have WinBy values of 8. Actually, this path might very well be the choice of moves executed during a match if two perfect opponents are playing each other. Here is why: if it is player one's turn for a given POSITION and it has a WINBY of 8, then that implies there exists a child with a WINBY of 8 and it is the maximum possible value. Similarly, if it is player two's turn, then that implies

there exists a child with a WINBY of 8 and it is the minimum of all its children. Thus, for any given position and any player, *the child with the same* WINBY *value as the parent is the best move that player can make.*

Currently, memdb and bpdb are the only database files that support WinBy. For memdb, WINBY values are used instead of the existing MEX values since there is no game that requires both pieces of data. This creates a limitation on how large or small a WINBY value can be, since MEX values are only given 5 bits. That is, a WINBY value is only allowed to range from -16 to 15; anything else is ill defined.

3 Function Names

The following are the main functions/globals associated with WinBy. All other functions are specific to the database or solver implementation.

`globals.h:`

```
BOOLEAN gWinBy;
```

This boolean variable determines whether or not the computer should play by the WinBy or remoteness optimization. The default value is TRUE, which implies that any game which supports WinBy will play by the WinBy optimization. The user can make the value FALSE inside Gamesman's configuration menu by pressing "w".

```
WINBY (*gPutWinBy)(POSITION);
```

This is an API function which allows the module to announce to the core that it supports the WinBy feature. This function pointer needs to point to whatever function the module uses to compute the WINBY value of a primitive position. The function takes in a hashed POSITION, and returns the WINBY value of that POSITION, always from player one's perspective.

`db.h:`

```
void WinByStore (POSITION pos, WINBY winBy);
```

This is the generic WinBy DB function that writes the WINBY value for the POSITION in the database. Precisely how it accomplishes this task is specific to what database is being used. In the case of memdb, the MEX value is overwritten, and in the case of bpdb, a WINBY slot is added.

```
WINBY WinByLoad (POSITION pos);
```

This is the generic WinBy DB function that loads the WINBY value for the given POSITION. In the case of memdb, it just reads the same bits that MEX normally occupies. In the case of bpdb, it reads from the WINBY slot. Sign extension is taken into account for the case of negative WINBY values.

gameplay.c:

```
MOVE GetWinByMove (POSITION position, MOVELIST* genMoves);
```

Given a POSITION and a MOVELIST generated from the POSITION, this function returns a MOVE to a child with the same WINBY value as its parent.

4 Module Writers

Module writers need to do three things in order to properly implement the WinBy feature. The first is to write a function that computes the WINBY value for a given primitive POSITION *from player one's perspective*. This function should take as input a POSITION and return a WINBY. Second, the module needs to set the global function pointer gPutWinBy to the address of the function that computes WINBY values. Third, to properly display the value in PrintPosition, the module should call GetPrediction. If WinBy is activated during gameplay, GetPrediction appropriately returns information regarding how much the player can win (or lose) by as well as how many moves need to be made (i.e., remoteness). If the user decides that he or she wants the computer to play by remoteness, then no WinBy information will be returned in the prediction string. That is it! Here is an example of all three steps, taken straight from mothello.c:

```
InitializeGame() {
  ...
  gPutWinBy = &computeWinBy;
  ...
}

WINBY computeWinBy(POSITION position) {
  int i;
  int whitetally = 0, blacktally = 0;
  char *board = getBoard(position);
  for(i = 0; i < (OthCols * OthRows); i++) {
    if(board[i] == WHITEPIECE)
      whitetally++;
    if(board[i] == BLACKPIECE)
      blacktally++;
  }
  return blacktally - whitetally;
}

void PrintPosition (POSITION position, STRING playerName, BOOLEAN usersTurn) {
  ...
  sprintf(prediction,"| %s",GetPrediction(position,playerName,usersTurn));
  ...
}
```

5 Future Goals

In the future, WinBy should be made independent of generic_hash. Also, it would be nice to have a larger range of WINBY values (as of now, the range is -16 to 15). Lastly, we would like to eventually add a feature that allows the computer to minimize the WINBY value (but still win), in the interest of giving the human player false hope.