GLOBAL POSITIONING SYSTEM
(GPS)

GamesCrafters Research Group

University of California, Berkeley

TABLE OF CONTENTS

1. History

The idea of GPS was developed by Eric Siroker in Spring 2005.

2. Overview

This document is intended to introduce the particular feature of the GAMESMAN solver, Global Positioning System (GPS), made by the GamesCrafters Research Group at the University of California, Berkeley. This is especially intended for a fellow GamesCrafters programmer who wants to understand the concept of GPS in order to improve it or apply that knowledge to anything/anywhere he/she pleases.

Please note that GPS may only be implemented on non-loopy games, and any non-loopy game should implement this feature (see *Usage and Implementation*). The time saving bonus by cleverly allocating a global variable is undoubtedly a win. See games like tic-tac-toe or quarto for examples of GPS implementation.

3. Prerequisite Knowledge

To understand how it works, you must first understand how the current system solves games. The current system is a stateless solving system, which solves a game completely by doing depth first search on the game tree.

The recursive call follows these steps:
We start by checking if the current board is already hashed[1] (in case there are multiple ways to reach the same board). If it is already hashed, we already know the result. If it is not already hashed, we need to check if it's a primitive[2] (win, loss, or tie). If it is a primitive, we have reached the end of our game tree and we can return a value of the board. If not then examine its children to see if the current board is a winning or losing one[3] by generating a list of all possible moves and doing one move at a time creating a new board on which we must recurse (depth first search).

The numerical superscripts above refer to these three functions:

[1]`generateMoves`, [2]`doMove`, [3]`primitive`

These functions work as follows:

`MOVELIST generateMoves(board);`	$\rightarrow$	returns a list of moves
`POSITION doMove(board, move);`	$\rightarrow$	returns a board
`VALUE primitive(board);`	$\rightarrow$	returns win, lose, or tie

By analyzing the function calls, we see what each function must do during its execution:

`generateMoves;`	YES unhash, NO hash
`doMove;`	YES unhash, YES hash
`primitive;`	YES unhash, NO hash

We represent a game board and game moves as hashed values and pass these hashed values to our functions. Move hashing and unhashing is relatively quick, so we will consider those as taking no time at all; however, boards can sometimes be quite large and require a significant amount of time to hash or unhash.
(Question: As the size of the board grows, what is most of the time of our solver spent doing?)

Looking at our recursion, we have to generate the moves, do the moves, check if it's a primitive and repeat. Any operation that takes in (a hashed representation of) a board needs to unhash it to see what the current board is before it can perform its operation.

As stated above, during each recursion, `generateMoves`, `doMove`, and `primitive` all need to unhash the boards (we are unhashing the same board 3 times!), and `doMove` needs to hash the resulting board at the end of performing the move.

There must be a better way to do this. Enter GPS.

4. Usage

GPS may only be used on non-loopy games. This is because only one board is stored at a time. If running a depth first search on a loopy game, the same boards will recur, never reaching the end of recursion and returning a value for that board, resulting in an infinite loop.

4.1 Intended Usage

For a non-loopy game, GPS saves us a tremendous amount of time by eliminating the need for unhashing so many times. If we keep a global variable that can store an entire board, all functions can just access that board (eliminating it from the arguments passed in) resulting in these prototypes:

```
generateMoves();
doMove(move);
primitive();
```

With this scheme, we can reference the global board without passing around hashed values and, by doing so, we've eliminated the unhashing done in each of those functions! Now, instead of passing around hashed values, we just reference the global variable and analyze it. But this presents a major problem: if we have one board, we can only represent one board at a time, then how do we move up and down the game tree?

We must be able to progress and regress through games by modifying the global variable to explore the entire game tree.

To move forward, `doMove` must now modify that global board for the next board, effectively becoming `doMove!` (not pronounced do-move, but pronounced do-move-bang). `doMove!` is now a mutator.

To move backward, we will need an undo function that undoes what `doMove!` just did. `undoMove!` must restore the board to what it was before the move was done and is a mutator just like its counterpart `doMove!`.

We will now require the following set of functions:

```
generateMoves();
doMove!(move);
primitive();
undoMove!(move);
```

GPS has eliminated the need to unhash the same board numerous times. Unfortunately, doMove! must hash the resulting board and update the table of already hashed boards, so GPS does not help us here, but it does not hurt us either. GPS has drastically reduced the amount of time needed to run the solver by reducing the number of times we unhash a board.

4.2 Additional Usage Possibilites

Now we get even more creative. Since we can support a global variable for the board, why not store more information? We can store any additional variables that will improve our solve time.

For example, instead of having `generateMoves` search the board over and over again for free moves, why not just save a list of free moves along with the board. `generateMoves` now can just return that variable instead of searching a board to generate the moves, and `doMove!` just needs to do an additional task of updating the list of moves when it is updating the board.

We can store a list of previously executed moves up to the current board so `undoMove!` will not need any arguments at all. `doMove!` just needs to do an additional task of updating the list of moves when is is updating the board.

We can even add fields that contain information to eliminate redundant searching. In the game connect four, we need to search for four pieces in a specific configuration, namely a row, column or diagonal. If a portion of the board is already completely filled and no winning configurations are present, why re-search that section for a winning configuration everytime we call `primitive`? We can store a variable which indicates which areas not to re-search, which `doMove!` must update when it is updating the board.

The list of additional uses for GPS is infinite and it is up to the programmer to decide how to use it. However, the programmer should keep in mind the drawback of GPS, which is with the every new field introduced, the more complicated `doMove!` must be. `doMove!` must be able to update all these fields leading to a more complex `doMove!` function.

It is wise to remember “bigger ain’t better.”

5. Implementation

Each module will need to support a solver that can run without GPS. In order to implement GPS, the modified functions must be used (see *Usage*). It is silly to write seven functions and reproduce large portions of code that both styles of solving use, so most likely the programmer will use a function that decides whether or not to solve with GPS. The solver now will also need to know the differences between running the solver with GPS and without.

5.1 Example: Tic Tac Toe

Tic Tac Toe is an example where, depending on whether GPS is used or not, the function behaves in two very different ways.

```
POSITION DoMove(POSITION position, MOVE move)
{
  if (gUseGPS) {
    gPosition.board[move] = gPosition.nextPiece;
    gPosition.nextPiece = gPosition.nextPiece == x ? o : x;
    ++gPosition.piecesPlaced;

    return BlankOXToPosition(gPosition.board);
  }
  else {
    BlankOX board[BOARDSIZE];
    PositionToBlankOX(position, board);
    return position + g3Array[move] * (int) WhoseTurn(board);
  }
}
```

5.2 Example: Quarto

Quarto is an example where much of the code is repeated whether GPS is being used or not.

```
POSITION DoMove (POSITION position, MOVE move)
{
  QTBPtr board;
  int piece, slot;
  POSITION newposition;
  /* Use GPS board if GPS solving, otherwise use unhashed position */
  board = gUseGPS ? GPSBoard : unhash( position );
  slot = GetMoveSlot( move );
  piece = GetMovePiece( move );
  board->piecesInPlay += ( piece == GetHandPiece( board ) ) ? 0 : 1;
  board->slots[slot] = GetHandPiece( board );
  SetHandPiece( board, ( piece == board->slots[HAND] ) ? EMPTYSLOT : piece );
  board->squaresOccupied += ( slot == HAND ) ? 0 : 1;
  newposition = hash(board);
  /* Deallocate board if not using GPS */
  if (!gUseGPS) {
    FreeBoard(board);
  }
  return newposition;
}
```