The Gamesman's Toolkit

David Wolfe *

Abstract

Tired of all those hand calculations? Of converting to canonical forms? Of calculating means and temperatures? Of wondering if you've goofed? Wanted to play with the overheating operator, but don't have any patience?

A toolkit written in C (under UNIX) implements virtually all of the material in finite combinatorial game theory. The toolkit is invaluable in analyzing games, and generating and testing hypotheses. In fact, several of the results presented at the 1994 Combinatorial Games Workshop were discovered using the program.

This paper introduces the features of the program in order to show you what can be done with the toolkit and to convince you it's both useful and fun to use.

1 Overview

A toolkit[1] written in C (under UNIX) implements virtually all of the material in finite combinatorial game theory [BCG82] [Con76] [BW94]. The toolkit can be used in one of two ways:

- It can be used "as is" by the gamesman who wishes to do standard algebraic manipulations on games. You might then think of the program as a games calculator.
- It can be used as a programming toolkit to, for instance, analyze a particular game. A broad base of game theoretic functions are provided, along with a parser and output routines.

Either way, the program has proved to be a versatile and powerful tool for the student and the researcher.

*Computer Science Division, UC Berkeley, CA 94720. Email: wolfe@cs.berkeley.edu

[1] The toolkit is free, and is available by sending e-mail to `wolfe@cs.berkeley.edu`. I've chosen to distribute it by e-mail so I can maintain a mailing list for notifying you of significant improvements in the program. Enjoy!

2 As a games calculator

Figure 1 shows a sample run of the program. Most commonly, the gamesman enters a game expression, and the computer converts the expression to canonical form. The sample run best demonstrates how to represent a game expression (typing `help` in the toolkit brings up on-line help as well.) However, these few general guidelines will help:

- The caret, "`^`", represents $\uparrow$; the letter vee, "`v`", is reserved for $\downarrow$. Triple-up can be typed as either "`^^^`" or "`^3`".
- You can assign variables to games and later reuse them. For example, you can set "`g = 2|1`" and later reuse `g` in another expression.
- Subscripts in games notation are typically enclosed by brackets for the toolkit. For example, the input format for $+_2$ (i.e, tiny-2) is `+[2]` and the input format for g_3 (i.e, g cooled by 3) is `g[3]`.
- Similarly, superscripts are enclosed by angle brackets, `<` and `>`.
- The `%` represents the heating (integral) symbol. So "`%[1*]<1>g`" denotes $\int_{1*}^{1} g$ (i.e, g overheated from $1*$ to 1) [Ber88] [BW94, p. 214].

3 As a programmer's toolkit

The most common use of the program as a toolkit is to analyze a particular game. You may wish to implement the rules to that game, and have the computer evaluate positions of that game. Unfortunately, I had only an experienced programmer in mind when I wrote the program. So, if you are experienced (or patient) you should find the toolkit quite versatile.

Several people have already written programs which use the toolkit to analyze a game. Konane (programmed by Michael Ernst), Toads & Frogs (Jeff Erickson) and Domineering (David Wolfe) come with the toolkit. Dan Garcia has written a wonderful interface using `tcl` for playing domineering on an X window [Gar].

To get you started, Figure 2 summarizes the most useful files and functions you'll need. Figure 3 contains a sample program which evaluates positions in Wyt Queens (or Wythoff's game) [BCG82, p. 61].[2]

On first reading, you may wish to ignore that part of the code related to the hash table. The hash table is used for efficiency, avoiding the exponential blowup of reevaluating the same positions recursively over and over. The lines of code which relate to the hash table are marked with an "`-`", and can be omitted from the program if efficiency is not an issue.

[2] Wyt Queens is played by on quarter-infinite chess board, where a queen (or, in general, several queens moving independently) is placed on a square. A move consists of moving the queen (or a queen) either north, west, or northwest any number of squares. The last player to move wins.

`unix% games`	The program is executed from the UNIX prompt
`Type 'help' and...`	You are advised to type `help` and `help help`.
`> * ? ^`	How does $*$ compare with $\uparrow$? "`> `" is the prompt.
`<>`	Answer: they are incomparable
`> * ? ^^`	How does $*$ compare with $\Uparrow$?
`<`	Answer: $*$ is less than $\Uparrow$
`> -2 + 3/8 + vvv + *6`	Compute $-2 + \frac{3}{8} + \Downarrow\!\!\!\downarrow + *6$ (`v3` is short for `vvv`)
`-13/8v3*6`	Answer: $-\frac{13}{8} + \Downarrow\!\!\!\downarrow + *6$
`> +[2],*|-2`	Find the canonical form of $\{+_2, * \mid -2\}$;
`0,*|-2`	brackets indicate subscripts so `+[2]` is tiny-two
`> $[1/2]`	Cool the last result (\$) by $\frac{1}{2}$
`-1/2|-3/2`	$\{0, * \mid -2\}$ cooled by $\frac{1}{2}$ is $\{-\frac{1}{2} \mid -\frac{3}{2}\}$
`> g = $`	Assign a variable g to the last result, denoted \$
`g = -1/2|-3/2`	
`> mean g`	Calculate g's mean and temperature
`-1`	
`> temp g`	
`1/2`	
`> (1|||0||-1|-3)[1/2]`	Cool $1\|\|\|0\|\|-1\|-3$ by $\frac{1}{2}$
`1/2||0|g`	Note the machine remembers that g is $\{-\frac{1}{2} \mid -\frac{3}{2}\}$
`> no ups`	Please do not print out $\uparrow$ in canonical forms
`> ^*`	What's the canonical form of $\uparrow *$?
`0,*|0`	It's $\{0, * \mid 0\}$
`> domino`	Enter a domineering position
`Enter domin...`	
`xxxx`	
`  xx`	
`xxxx`	The position is ended with an extra carriage return
`-3/2|-4`	The 10 box position has value $\{-\frac{3}{2} \mid -4\}$

Figure 1: *A sample run of the program* `games` *which demonstrates a few of its features. The table is in two columns: On the left is the computer dialogue, and on the right appear comments. What the computer prints is in* `typewriter font` *and what the user types is* `underlined`.

gameops.h: Performs operations on games (game_type)	
init()	must be called once to initialize the game routines
make(list_type, list_type)	constructs a game from two lists of options and converts it to canonical form; destroys its arguments
zero	the zero game
num(Q_type), up(int), star(int)	construct games consisting numbers, ups and star
plus(game_type, game_type)	Add two games, etc.
minus(game_type, game_type)	
negative(game_type)	
eq(game_type, game_type), ge(","), ...	compare games (equal, $\geq$, etc.)
is_int(game_type), is_num(game_type)	TRUE if game is an integer, number
list_type left_options(game_type)	game's list of left options
list_type right_options(game_type)	game's right options
output.h: Output routines for games and lists	
game_printf(game_type)	output a game plus a newline
game_print(game_type)	output a game w/o the newline
game_sprintn(game_type)	print a game to a string w/o newline, maximum of n characters
list.h: Manipulates lists of integers or games (list_type)	
list_make()	returns an empty list
list_insert(list_type, int or game_type)	insert an element into a sorted list
list_prepend(list_type, int or game_type)	prepend to an unsorted list
int or game_type list_nth(list_type)	find the nth element of a list
list_copy(list_type)	return a copy of a list
list_free(list_type)	destroy a list and free its space
rational.h: Manipulates low precision rational numbers (Q_type)	
Q(int, int)	construct a rational p/q
int Q_p(Q_type), Q_q(Q_type)	numerator or denominator of a rational
hash.h: Maintains a hash table keyed by an (int, list_type) pair	
boolean hash_test(int, list_type)	TRUE if the hash table contains entry
game_type hash_get_last()	get the last entry tested positive
hash_put(int, list_type, game_type)	put a value into the hash table; destroys its list argument

Figure 2: *Most common functions used by a programmer of the toolkit.*

```
/* A programming example: The game of Wyt Queens */
#include "games.c"       /* Includes all the needed .h files */
#define QUEENS_KEY 1001  /* Hash table keys below 1000 are reserved. */

game_type queens (int x, int y) {
    list_type posn_as_list, left, right;
    game_type g;
    int n, min;

-   /* Encode the position as a list of integers, in order to use the hash
-      table to store computed positions and avoid recomputing positions */
-   posn_as_list = list_make();
-   list_prepend (posn_as_list, x);
-   list_prepend (posn_as_list, y);
-   if (hash_test (QUEENS_KEY, posn_as_list)) { /* If pos'n previously computed */
-       list_free (posn_as_list);               /* Free space used by the list */
-       return hash_get_last();                 /* Return computed value */
-   }

    /* Position wasn't already computed, so evaluate the position recursively. */
    left = list_make();
    for (n=0; n<x; n++) list_insert (left, queens (n, y)); /* Horizontal moves */
    for (n=0; n<y; n++) list_insert (left, queens (x, n)); /* Vertical */
    min = ( x<y ? x : y);
    for (n=1; n<=min; n++) list_insert (left, queens (x-n,y-n)); /* Diagonal */
    right = list_copy (left);       /* Right's options are the same as Left's. */
    g = make (left, right);         /* Construct the game's canonical form.
                                       Lists left and right are destroyed. */

-   /* Store the value of the position in hash table.  The posn_as_list is
-      destroyed and freed by hash_put() by, so no need to list_free() it. */
-   hash_put (QUEENS_KEY, posn_as_list, g);
    return g;
}

#define MAXBUFF 10
void main() {
    char s[MAX_BUFF];

    init();                                  /* initialize the toolkit! */
    game_sprintn (s, queens(2,6), MAXBUFF-1); /* Store position in string s */
    printf ("A queen at location (2,6) has value %s\n", s);
}
```

Figure 3: *Sample main() program. Lines marked with an "-" are for efficiency.*

4 Acknowledgement

Elwyn Berlekamp first encouraged me to write the toolkit in a course at UC Berkeley in Spring of 1989. Many people have since contributed ideas and code to the program, including Dan Calistrate, Raymond Chen, Jeff Erickson, Michael Ernst, Dan Garcia, Yonghoan Kim, David Moews, and Martin Mueller.

References

[BCG82] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*. Academic Press, New York, 1982.

[Ber88] Elwyn R. Berlekamp. Blockbusting and domineering. *Journal of Combinatorial Theory*, 49(1):67–116, September 1988.

[BW94] Elwyn Berlekamp and David Wolfe. *Mathematical Go: Chilling Gets the Last Point*. A K Peters, Ltd., Wellesley, Massachusetts, 1994.

[Con76] John H. Conway. *On Numbers and Games*. Academic Press, London/New York, 1976.

[Gar] Dan Garcia. A graphical, X-based front-end for domineering. (A tcl interface which runs on top of the games toolkit.) Available from Dan Garcia by e-mail: `ddgarcia@cs.berkeley.edu`.