Theory of Computer Games: Concluding Remarks

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu

Abstract

- **Practical issues.**
  - **Smart usage of resources.**
    - ▷ *Time*
    - ▷ *Memory*
    - ▷ *Coding efforts*
    - ▷ *Debugging efforts*
  - **Putting everything together.**
    - ▷ *Software tools*
    - ▷ *Fine tuning*
  - **How to know one version is better than the other?**
- **Concluding remarks**

Using resources: time and others

- **Time is the most critical resource [Hyatt 1984] [Šolak and Vučković 2009].**
- **Watch out different timing rules.**
  - **An upper bound on the total amount of time can be used.**
    - ▷ *It is hard to predict the total number of moves in a game in advance. However, you can have some rough ideas.*
  - **Fixed amount of time per ply.**
  - **An upper bound T_1 on the total amount of time is given, and then you need to play X plys every T_2 amount of time.**

Wall clock time vs CPU time

- **A system and O.S. issue.**
  - **CPU time measures the time spent on your process.**
  - **Wall clock time is the turn around, i.e., real, time used.**
  - **In a time-sharing system, many processes are running at the same time.**
  - **Wall clock time $>>$ CPU clock time.**
  - **For tournaments, we only care about wall clock time.**

Sample code

- **Example (Unix based)**
  - ▷ ***CPU time***

```
#include <time.h>
...
double start = (double) clock();
...
double end = (double) clock();
double cpu_time_in_seconds =
                   (end - start) / (double) CLOCK_PER_SEC;
```

  - ▷ ***Wall clock time***

```
#include <time.h>
...
   struct timespec start, end;
clock_gettime(CLOCK_REALTIME, &start);
...
clock_gettime(CLOCK_REALTIME, &end);
double wall_clock_in_seconds =
               (double)((end.tv_sec+end.tv_nsec*1e-9) -
               (double)(start.tv_sec+start.tv_nsec*1e-9));
```

Commonly time-using rules (1/2)

- Assume you have a total of T time to spend.
- Related terms
  - Time has already spent
  - Planned time to spent for this ply
    - ▷ *May be larger or smaller than the actual time spent due to time controlling schemes used.*
- Estimate the total number of plys N that you need to play during a game.
  - Collect these data empirically
  - Do not be over optimistic
- Commonly used formulas
  - Fixed
    - ▷ *time: Spend $\frac{T}{N}$ time for each ply*
    - ▷ *depth: Search up to to depth D for each ply where D is estimated using $\frac{T}{N}$ time before the tournament.*
  - Dynamic
    - ▷ *Let t_i be the time you have spent at the ith ply, for $i < j$.*
    - ▷ *Plan to spend $\frac{T-\sum_{i=1}^{j-1} t_i}{N-j+1}$ time for the jth ply.*

Commonly time-using rules (2/2)

- **Advanced techniques:**
  - **The amount of time spent during each phase of the game is different.**
    - *open game: knowledge is needed more than depth; however, need some depth, say 4.*
    - *middle game: deeper depth is needed*
    - *end game: depth is on demand*
- **To avoid extreme cases**
  - **Set a minimum/maximum time to think.**
    - *This is critical when the number of plys N is going to exceed your prior estimation.*
  - **Set a minimum/maximum depth to search.**
- **Reminders:**
  - **Dynamically adjusting**
    - *When there is only one possible move, do not think.*
    - *When the score is **stable**, cut short the time to spend.*
    - *When the situation is **dangerous**, spend more time.*
  - **Watch the time spent by your opponent.**
    - *When he is going to be out of time, do not let him have a chance to use your time in doing **pondering**.*

When and how to set time usage

- **When to check the current time usage**
  - **Iterative deepening: each time entering a new depth**
  - **Using system interrupt on a fixed time interval**
  - **MCTS: each time a selection process begins**
- **Estimation of time usage**
  - **Iterative deepening**
    - ▷ ***t_i: average time, during the last few plys, spent in searching depth-i***
    - ▷ ***prediction: $t_{i+1} \sim (t_i \cdot \frac{t_i}{t_{i-1}})$, $i > 1$***
    - ▷ ***if the remaining time for this ply is less than the predicted time, then do not initiate a new depth***
  - **MCTS: an almost constant amount of time is spent when a node a expanded and simulated**

Pondering

- **Pondering:**
  - **Use the time when your opponent is thinking.**
  - **Guessing and then pondering.**
  - **System issues.**
    - ▷ ***How interrupt is handled?***
    - ▷ ***Polling every now and then or triggered by events?***
- **How pondering is done:**
  - **In your turn, keep the first 2 plys m_1 and m_2 in the PV you obtained.**
    - ▷ ***You choose to play m_1, and then it's the opponent's turn to think.***
    - ▷ ***In pondering, you assume (guess) the opponent plays m_2.***
    - ▷ ***Then you continue to think at the same time your opponent thinks as if he has played m_2.***
  - **Guessing right: If the opponent plays m_2, then you can continue the pondering search in your turn.**
  - **Guessing wrong: If the opponent plays a move other than m_2, then you restart a new search.**
- **Doing pondering requires the ability to know when a move is made by your opponent using system interrupt, or you need to check from time to time (polling).**

Comments about time usage

- **Thinking style of human players.**
  - **Using almost no time while you are in the open book.**
  - **More time is spent in the beginning immediately after the program is out of the book, and then slowly decrease the searching time.**
  - **In the endgame phase, use more time in critical positions or when you try to initiate an attack.**
- **Points to watch:**
  - **Over time: lose no matter how good you are at the moment.**
    - ▷ ***When the amount of your time left is low, speed up the search.***
    - ▷ ***When the amount of your opponent's time is low and you are more than his, spend less time and wait for his over time.***
  - **Iterative deepening helps in time planning.**
    - ▷ ***Need to set a minimum searching depth.***
    - ▷ ***Need to set a maximum searching depth to avoid buffer overflow.***

Comments

- Do not think at all if you have only one possible logical move left.
- Search only counter-checking moves if they exist.
- Does the first player really have to think for the first ply?
  - Use some open books to save time during the opening.
- When the results of the previous two iterations differ a lot, consider spending more time to verify.
- When you have searched to a certain depth and the results are stable in the previous rounds, consider to stop early.
  - Be sure to use some Quiescent search algorithm plus SEE.
  - You have searched the minimum depth.
  - The recent several depths continuously return the same best ply and almost about the same best score.
    - *Need to watch the ratio of failed low or failed high in your searching.*
    - *When your ratio of failed low is high, then you are too optimistic.*
    - *When your ratio of failed high it low, then you are too pessimistic.*

Using other resources

- **Memory**
  - **Using a large transposition table occupies a large space and thus slows down the program.**
    - ▷ *A large number of positions are not visited too often.*
  - **Using no transposition table may cause searching some critical positions too many times.**
- **CPU/GPU**
  - **Do not fork a process to search branches that have little hope of finding the PV even you have more than enough hardware.**
    - ▷ *You need to wait for its termination.*
    - ▷ *Synchronization takes resources.*
- **Other resources.**

Putting everything together

- **Game playing system**
  - **GUI.**
  - **Data structures.**
    - ▷ *Using a 2-D array to store the board and find everything by scanning the board is time consuming.*
    - ▷ *Better strategy: have a list of pieces that are still alive and a board at the same time with proper co-referencing.*
  - **Use some sorts of open books.**
  - **Middle-game searching: usage of a search engine.**
    - ▷ *Evaluation function: knowledge.*
    - ▷ *Main search algorithm: iterative deepening.*
    - ▷ *Enhancements: transposition tables, Quiescent search and possible others.*
  - **Use some sorts of endgame databases.**
- **Debugging and testing**

Board

- **Use a 1-D array for the board with an extra boarder around the board.**
  - **Example: CDC.**
  - **Array index L means a 2-D location (x, y) where $x = L\%10$ and $y = L/10$.**
    - *Can consider $x = L\&0xF$ and $y = L >> 4$ for faster arithmetics.*
  - **Boarders are at $P[0, *], P[*, 9], P[9, *], P[*, 0]$.**
- **Advanced data structure: bit boards.**
  - **Using a binary string for the board.**
- **Remark: avoid using auto-dynamic data structures unless you know them really well.**
  - **MAP/VECTOR in recent C++.**

Sample data structures for CDC

```
// boards
//       11,12,13,14,15,16,17,18
//       21,22,23,24,25,26,27,28
//       31,32,33,34,35,36,37,38
//       41,42,43,44,45,46,47,48
struct n_b{
       char inside; // 1 if in the board
       char empty; // whether it is empty
       char dark; // whether it is dark
       char color; // 0 or 1
       char piece;
       ...
      } board[(4+2)*(8+2)];

char is_inside(int index){
     return board[index].inside;
}
```

Using pre-computed tables

- **Save frequently used computation in tables.**
  - **take advantage of a larger cache in recent CPU's.**
- **Examples:**
  - **Need to check whether two pieces at L1 and L2 are adjacent.**
    - ▷ ***Slow code:***

```
x1 = L1 % 10; x2 = L2 % 10;
y1 = L1 / 10; y2 = L2 / 10;
if((abs(x1,x2)==1 && y1==y2) ||
   (abs(y1,y2)==1 && x1==x2))
then return 1; else return 0;
```

    - ▷ ***Using pre-computed tables:***

```
return adjacent[L1][L2];
```

  - **Need to check whether one piece can capture the other.**
  - **Need to check whether two locations are at the same column or row.**
  - **...**

Checking legal moves (1/2)

```
// [(14+2)*(14+2)] array: 7 types, 2 colors plus dark and empty
// upper cases are red; lower cases are black
// can_eat_by_move[ELEPHANT][rook] == 1
// can_eat_by_move[rook][ELEPHANT] == 0
// can_eat_by_move[ELEPHANT][ROOK] == 0
// can_eat_by_move[ELEPHANT][dark or empty] == 0
// adjaent[X][Y]: whether locations X and Y are inside and adjacent
// same_row_column[X][Y]: where X and Y are inside and
// in the same row or column
char can_eat_by_move[7*2+2][7*2+2];

char is_legal_by_move(int from, int to, int color){
   return is_your_piece(from,color) &&
   adjacent[from][to] &&
   (is_empty(to) ||
   can_eat_by_move[board[from].piece][board[to].piece]);
}
```

Checking legal moves (2/2)

```
// legal cannon jumps
char is_legal_to_jump(int from, int to, int color){
   return is_your_cannon(from,color) &&
   is_enemy_piece(to,color) &&
   same_row_column[from][to] &&
   there_is_a_piece(from,to);
}
```

Lists of pieces

- **Need at least two data structures for the pieces.**
  - **Given a piece type, report its properties.**
    - ▷ ***An array of pieces indexing on pieces' types.***
    - ▷ ***Sample usage: find your pieces during move generation.***
  - **Given a location, report the piece at this location.**
    - ▷ ***Board.***
    - ▷ ***Sample usage: checking high level properties such as mobility.***

Piece list

```
// plist[RED][0..num_pieces[COLOR]-1] is the list of
// COLOR pieces that are alive and revealed
struct pl{
         int where;
         int piece_type;
         ...
         } plist[2][16];
int num_pieces[2]; // number of revealed and alive pieces

// remove the ith piece of color
void remove_piece(int i, int color){
   num_pieces[color]--;
   if(num_pieces[color] > 0){
     // swap the last piece to the ith location
     plist[i] = plist[num_pieces[color]];
   }
}
```

How moves are done

```
#define LEFT -1
#define RIGHT +1
#define DOWN +10
#define UP -10

#define move(IDX,DIR) (IDX+DIR)

// location i can move move_num[i] directions
// which are in move_dir[i][0..move_num[i]-1]
int move_dir[(4+2)*(8+2)][4];
int move_num[(4+2)*(8+2)];

// location i has a cannon
// it can jump jump_num[i] directions
// which are in jump_dir[i][0..jump_num[i]-1]
int jump_dir[(4+2)*(8+2)][4];
int jump_num[(4+2)*(8+2)];
```

Move generation

```
for(i=0;i<num_pieces[color];i++){
    from = plist[i].where;
    for(j=0;j<move_num[from];j++){
      to = from+move_dir[j];
      if(is_legal_by_move(from,to,color)){
        if(is_capture(from,to,color))
              generate_capture(from,to,color);
        else  generate_move(from,to,color);
      }
    }
    if(is_legal_to_jump(from,to,color)){
       for(j=0;j<jump_num[from];j++){
         to_dir = jump_dir[j];
         if(to = find_jump(from,to_dir,color))
           generate_jump(from,to,color);
      }
    }
}
```

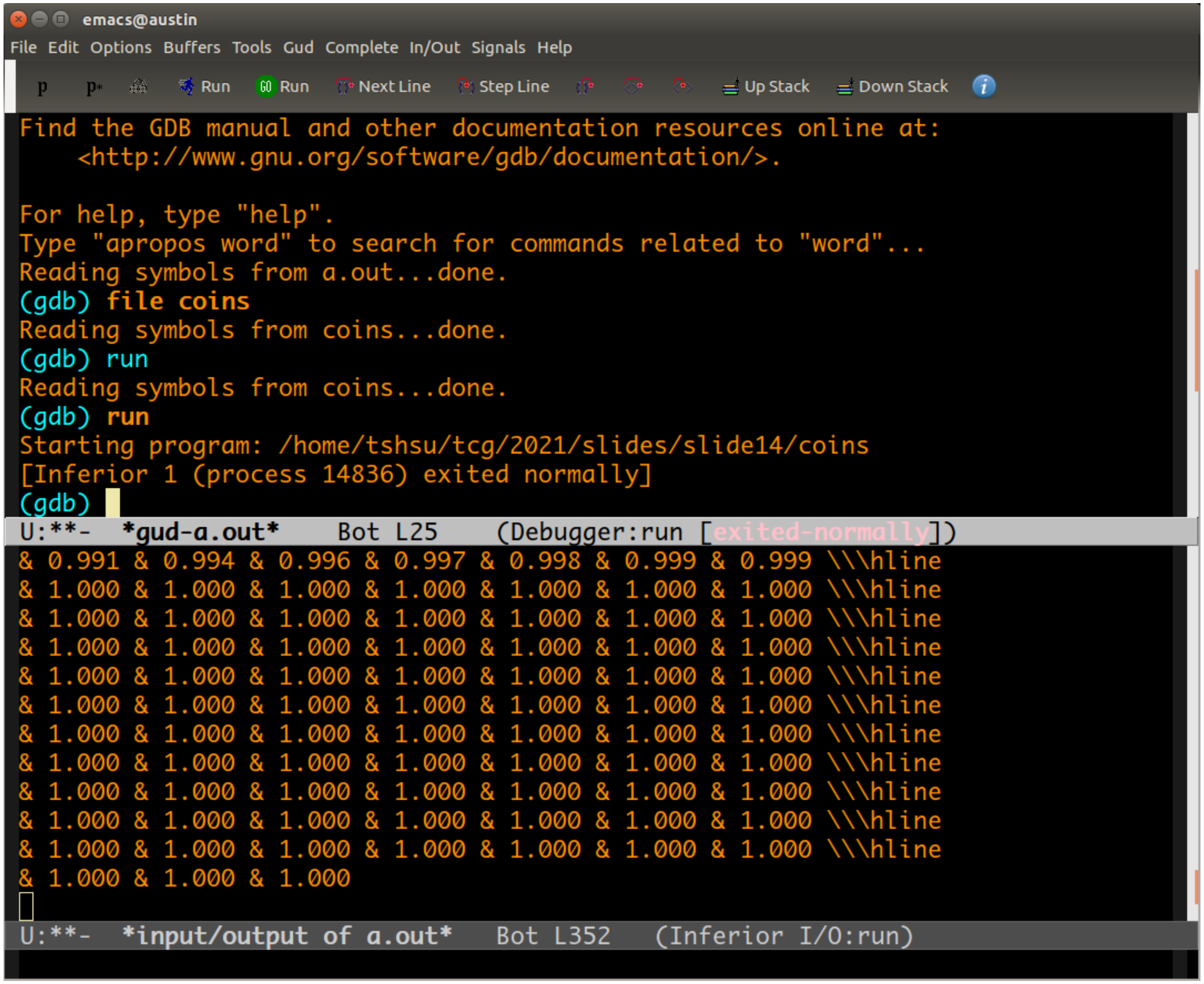
Software tools

- Using make to do a better software project management.
- Using svn or other version control tools to do code maintaining.
- Using compiler optimization switches to speed up.
  - CPU-dependent instructions
  - gcc -O2 main.c
  - gcc -O3 main.c
    - ▷ *Object code may not be stable using aggressive optimization flags.*
- Using gdb (GNU based) or other debugging tools to debug.
  - gdb a.out
- Using gprof (GNU based) or other profiling tools to find out the bottleneck of your code execution.
  - gcc -pg coins.c
  - ./a.out
  - gprof a.out gmon.out
- Using an Integrated Development Environment (IDE)
  - For Windows based systems, a good IDE is Dev C++.
  - Cross-platform: CODE::Blocks, VS code.
  - For Unix-based systems, emacs or vim can be set as an IDE.

Makefile example

```
all:    LezGo.c board.h gtp.h gostring.h UCT.h board.c gtp.c gostring.c UCT.c
        g++ -O3 -lm LezGo.c board.h gtp.h gostring.h UCT.h hash.h board.c gtp.c gostring.c UCT.c hash.c -o LezGo.exe
UCT:    LezGo.c board.h gtp.h UCT.h board.c gtp.c UCT.c liberty.h liberty.c
        g++ -O3 -lm -DUCT LezGo.c board.h gtp.h UCT.h board.c gtp.c UCT.c
 liberty.h liberty.c -o LezGo-UCT.exe

LX-all: LezGo.c board.h gtp.h gostring.h UCT.h board.c gtp.c gostring.c UCT.c
        gcc -O3 -lm LezGo.c board.h gtp.h gostring.h UCT.h hash.h board.c gtp.c
gostring.c UCT.c hash.c -o LezGo

LX-UCT: LezGo.c board.h gtp.h gostring.h UCT.h board.c gtp.c gostring.c UCT.c
        gcc -O3 -lm -DUCT LezGo.c board.h gtp.h gostring.h UCT.h hash.h board.c
 gtp.c gostring.c UCT.c hash.c -o LezGo-UCT

prof:   LezGo.c board.h gtp.h gostring.h UCT.h board.c gtp.c gostring.c UCT.c
        g++ -O3 -g -pg -lm -DUCT LezGo.c board.h gtp.h gostring.h UCT.h hash.h

board.c gtp.c gostring.c UCT.c hash.c liberty.h liberty.c -o LezGo-prof

debug: LezGo.c board.h gtp.h gostring.h UCT.h board.c gtp.c gostring.c UCT.c
        g++ -g -lm -DUCT LezGo.c board.h gtp.h gostring.h UCT.h hash.h board.c
gtp.c gostring.c UCT.c hash.c liberty.h liberty.c -o LezGo-prof

clean:  LezGo
        rm -rf LezGo
```

gdb example


emacs@austin
File Edit Options Buffers Tools Gud Complete In/Out Signals Help
Run
Run
Next Line
Step Line
Up Stack
Down Stack
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) file coins
Reading symbols from coins...done.
(gdb) run
Reading symbols from coins...done.
(gdb) run
Starting program: /home/tshsu/tcg/2021/slides/slide14/coins
[Inferior 1 (process 14836) exited normally]
(gdb)
U:**- *gud-a.out* Bot L25 (Debugger:run [exited-normally])
& 0.991 & 0.994 & 0.996 & 0.997 & 0.998 & 0.999 & 0.999 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\hline
& 1.000 & 1.000 & 1.000
U:**- *input/output of a.out* Bot L352 (Inferior I/O:run)

Profiling

```
tshsu@austin:~/tcg/2016/slides/slide13$ gprof a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 57.71      1.45     1.45 300000000     4.83     4.83  coin
 32.46      2.26     0.81 150000000     5.43    15.09  pair_toss
  9.62      2.50     0.24                             main
  0.80      2.52     0.02                             frame_dummy
```

Call graph

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.40% of 2.52 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.2    0.24    2.26                 main [1]
                0.81    1.45 150000000/150000000     pair_toss [2]
-----------------------------------------------
                0.81    1.45 150000000/150000000     main [1]
[2]     89.6    0.81    1.45 150000000         pair_toss [2]
                1.45    0.00 300000000/300000000     coin [3]
-----------------------------------------------
                1.45    0.00 300000000/300000000     pair_toss [2]
[3]     57.4    1.45    0.00 300000000         coin [3]
-----------------------------------------------
                                                 <spontaneous>
[4]      0.8    0.02    0.00                 frame_dummy [4]
-----------------------------------------------
```

Code for the sample profile (1/4)

```
// find the marginal pdf of a trinomial distribution

#include <stdio.h>
#include <stdlib.h>

//#define MAX_TRIALS 1000000000 // number of trials
#define MAX_TRIALS 1000000 // number of trials
#define MIN_N 10
#define MAX_N 50
#define N_INCR 10
#define MAX_VAL (2*MAX_N+1)
int win = 1; // points for a win
int draw = 0; // points for a draw
int loss = -1; // points for a loss
// prwin: win prob, prdraw: draw prob, 1-prwin-prdraw: lose prob
double pr_win = 0.3918; // Pr of win by the first player
double pr_draw = 0.3161; // Pr of draw by the first player
long int seedval = 5431276231; // a random magic number
```

Code for the sample profile (2/4)

```
// toss a coin with 3 outcomes
int coin(double prwin, double prdraw)
{
  double t;

  if((t = drand48()) <= prdraw) return draw; // draw
  else if(t <= prdraw+prwin) return win; // win
  else return loss; // loss
}

// the score of a pair of games
int pair_toss()
{
  int score=0;

  score += coin(pr_win,pr_draw);  //first player
  score += coin(1.0 - pr_win - pr_draw,pr_draw);  //second player
  return score;
}

main()
{
  int number;
  int s;
  int n;
  int i,j;
  int values[MAX_VAL];
  int accu,val;

  srand48(seedval);
```

Code for the sample profile (3/4)

```
for(n=MIN_N;n<=MAX_N;n+=N_INCR){
  for(j=0;j<MAX_VAL;j++) values[j] = 0;
  // perform MAX_TRIALS experiments
  for(number = 0; number < MAX_TRIALS;number++){
    // perform n trials
    val = 0;
    for(i=0;i<n;i++){
      val += pair_toss();
    }
    if(val < 0) val = -val;
    values[val]++;
  }

  // print header of each line
  accu = 0;
  for(s=0;s<=n*2;s++){
    accu += values[s];
    printf("n=%3d s=%3d Pr(|Xn|<=s)=%10d/%d\n",n,s,accu,MAX_TRIALS);
  }
```

Code for the sample profile (4/4)

```
    // output distribution
    {
      int cc;
      double f1,f2;

      cc = 0;
      accu = 0;
      f2 = MAX_TRIALS;
      for(s=0;s<=n*2;s++){
        accu += values[s];
        f1 = accu;
        printf("& %4.3f ",f1/f2);
        cc++;
        if(cc % 7 == 0) printf("\\\\\\hline\n");
      }
      printf("\n");
    }
  }
}
```

Comments

- **Coding efforts.**
  - **Iterative improving.**
    - ▷ *Build a **working** version using a minimum effort.*
    - ▷ *Add features one at a time.*
    - ▷ *Always keep a working version in the process.*
  - **Build a testing script so that it will test all previous tested features when a new one is added.**
    - ▷ *A new feature may cause an old function to have new bugs.*
- **Understand your bottleneck and find the right way to remedy it.**
- **Maintain a test log to know which tricks are good and which are not.**

Testing

- **You have two versions P_1 and P_2.**
- **You make the 2 programs play against each other using the same resource constraints.**
  - **Self-play.**
- **To make it fair, during a round of testing, the numbers of a program playing first and second are equal.**
- **After a few rounds of testing, how do you know P_1 is better or worse than P_2?**
  - **How many rounds are needed to verify it?**

How to know you are successful

- **Assume during a self-play experiment, two copies of the same program are playing against each other.**
  - **Since two copies of the same program are playing against each other, the outcome of each game is an independent random trial and can be modeled as a trinomial random variable.**
  - **Assume for a copy playing first,**

$$Pr(game_{first}) = \begin{cases} p & \textbf{if win} \\ q & \textbf{if draw} \\ 1-p-q & \textbf{if lose} \end{cases}$$

  - **Hence for a copy playing second,**

$$Pr(game_{last}) = \begin{cases} 1-p-q & \textbf{if win} \\ q & \textbf{if draw} \\ p & \textbf{if lose} \end{cases}$$

Outcome of self-play games

- **Assume $2n$ games, $g_1, g_2, \ldots, g_{2n}$ are played.**
  - **In order to offset the initiative, namely first player's advantage, each copy plays first for n games.**
    - ▷ ***We also assume each copy alternatives in playing first.***
  - **Let g_{2i-1} and g_{2i} be the ith pair of games.**
- **Let the outcome of the ith pair of games be a random variable X_i from the prospective of the copy who plays g_{2i-1}.**
  - **Assume we assign a score of w for a game won, a score of 0 for a game drawn and a score of $-w$ for a game lost.**
- **The outcome of X_i and its occurrence probability is thus**

$$Pr(X_i) = \begin{cases} p(1-p-q) & \textbf{if } X_i = 2w \\ pq + (1-p-q)q & \textbf{if } X_i = w \\ p^2 + (1-p-q)^2 + q^2 & \textbf{if } X_i = 0 \\ pq + (1-p-q)q & \textbf{if } X_i = -w \\ (1-p-q)p & \textbf{if } X_i = -2w \end{cases}$$

How good we are against the baseline?

- **Properties of X_i.**
  - **The mean $E(X_i) = 0$.**
  - **The standard deviation of X_i is**

$$\sqrt{E(X_i^2)} = w\sqrt{2pq + (2q + 8p)(1 - p - q)},$$

  and it is a multi-nominally distributed random variable.
- **When you have played n pairs of games, what is the probability of getting a score of s, $s > 0$?**
  - **Let $X[n] = \sum_{i=1}^{n} X_i$.**
    - ▷ ***The mean of $X[n]$, $E(X[n])$, is 0.***
    - ▷ ***The standard deviation of $X[n]$, σ_n, is $w\sqrt{n}\sqrt{2pq + (2q + 8p)(1 - p - q)}$,***
  - **If $s > 0$, we can calculate the probability of $Pr(|X[n]| \leq s)$ using well known techniques from calculating multi-nominal distributions.**
    - ▷ ***When n is large, it is very close to a normal distribution.***

Practical setup

- **Chinese chess**
  - $w = 1$, $p \sim 0.3918$, $q \sim 0.3161$, **and** $1 - p - q \sim 0.2920$.
    - ▷ ***Data source: 63,548 games played among masters recorded at www.dpxq.com.***
    - ▷ ***This means the first player has a better chance of winning.***
  - **The mean of** $X[n]$, $E(X[n])$, **is** 0.
  - **The standard deviation of** $X[n]$, σ_n, **is**

$$w\sqrt{n}\sqrt{2pq + (2q + 8p)(1 - p - q)} = \sqrt{1.16n}.$$

  - **When** $n = 100$, $\sigma_{100} \sim 10.8$.

Results (Chinese chess) (1/3)

$Pr(\|X[n]\| \leq s)$	$s = 0$	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 5$	$s = 6$
$n = 10,\ \sigma_{10} = 3.67$	0.108	0.315	0.502	0.658	0.779	0.866	0.924
$n = 20,\ \sigma_{20} = 5.19$	0.076	0.227	0.369	0.499	0.613	0.710	0.789
$n = 30,\ \sigma_{30} = 6.36$	0.063	0.186	0.305	0.417	0.520	0.612	0.693
$n = 40,\ \sigma_{40} = 7.34$	0.054	0.162	0.266	0.366	0.460	0.546	0.624
$n = 50,\ \sigma_{50} = 8.21$	0.049	0.145	0.239	0.330	0.416	0.497	0.571

Results (Chinese chess) (2/3)

$Pr(\lvert X[n]\rvert \le s)$	$s = 7$	$s = 8$	$s = 9$	$s = 10$	$s = 11$	$s = 12$	$s = 13$
$n = 10,\ \sigma_{10} = 3.67$	0.960	0.981	0.991	0.997	0.999	1.000	1.000
$n = 20,\ \sigma_{20} = 5.19$	0.851	0.899	0.933	0.958	0.974	0.985	0.991
$n = 30,\ \sigma_{30} = 6.36$	0.761	0.819	0.865	0.902	0.930	0.951	0.967
$n = 40,\ \sigma_{40} = 7.34$	0.693	0.753	0.804	0.847	0.883	0.912	0.934
$n = 50,\ \sigma_{50} = 8.21$	0.639	0.699	0.753	0.799	0.839	0.872	0.900

Results (Chinese chess) (3/3)

$Pr(\|X[n]\| \leq s)$	$s = 14$	$s = 15$	$s = 16$	$s = 17$	$s = 18$	$s = 19$	$s = 20$
$n = 10,\ \sigma_{10} = 3.67$	1.000	1.000	1.000	1.000	1.000	1.000	1.000
$n = 20,\ \sigma_{20} = 5.19$	0.995	0.997	0.999	0.999	1.000	1.000	1.000
$n = 30,\ \sigma_{30} = 6.36$	0.978	0.986	0.991	0.994	0.997	0.998	0.999
$n = 40,\ \sigma_{40} = 7.34$	0.952	0.966	0.976	0.983	0.989	0.992	0.995
$n = 50,\ \sigma_{50} = 8.21$	0.923	0.941	0.956	0.967	0.976	0.983	0.988

Statistical behaviors

- **Hence assume you have two programs that are playing against each other and have obtained a score of $s+1$, $s>0$, after trying n pairs of games.**
  - **Assume $Pr(|X[n]| \leq s)$ is say 0.95.**
    - ***Then this result is statistically meaningful, that is a program is better than the other, because the chance of $|X[n]| > s$ only happens with a low probability of 0.05.***
  - **Assume $Pr(|X[n]| \leq s)$ is say 0.22.**
    - ***Then this result is not statistically meaningful, because $|X[n]| > s$ only happens with a a high probability of 0.78.***
- **In general, it is a rare case in a normal distribution , e.g., less than 4.55% of chance that it will happen, that your score is more than $2\sigma_n$.**
  - **For our setting, if you perform n pairs of games, and your net score is more than $2 * \sqrt{1.16} * \sqrt{n} \simeq 2.154\sqrt{n}$, then it means something statistically.**
- **You can also decide your "definition" of "a rare case".**

Practical setup for self-play with no draws

- **For self play experiments with no draws**
  - **The mean of $X[n]$, $E(X[n])$, is 0.**
  - **The standard deviation of $X[n]$, σ_n, is**

$$w\sqrt{n}\sqrt{2pq + (2q + 8p)(1 - p - q)}$$

Examples

- **For self play experiments with no draws.**
- **Example I:** $w = 1$, $p = 0.5$ **and** $q = 0$. **Then** $\sigma_n = \sqrt{2n}$.
  - **When** $n = 10$, $\sigma_{10} \sim 4.47$.
    - ▷ ***max score = 20, min score = 0.***
    - ▷ ***if score*** > 8.94, ***then the two tested programs may be different in quality.***
  - **When** $n = 100$, $\sigma_{100} \sim 14.1$.
    - ▷ ***max score = 200, min score = 0.***
    - ▷ ***if score*** > 28.2, ***then the two tested programs may be different in quality.***
- **Example II (EWN):** $w = 1$, $p = 0.6$ **and** $q = 0$. **Then** $\sigma_n = \sqrt{1.92n}$.
  - **When** $n = 10$, $\sigma_{10} \sim 4.38$.
    - ▷ ***max score = 20, min score = 0.***
    - ▷ ***if score*** > 8.76, ***then the two tested programs may be different in quality.***
  - **When** $n = 100$, $\sigma_{100} \sim 13.86$.
    - ▷ ***max score = 200, min score = 0.***
    - ▷ ***if score*** > 27.71, ***then the two tested programs may be different in quality.***

Concluding remarks

- **Consider your purpose of studying a game:**
  - **It is good to solve a game completely.**
    - ▷ *You can only solve a game once!*
  - **It is better to acquire the knowledge about why the game wins, draws or loses.**
    - ▷ *You can learn lots of knowledge.*
  - **It is even better to discover knowledge in the game and then use it to make the world a better place.**
    - ▷ *Understand the fundamental properties such as how rules and boundary affect the game behavior and use that to improve our life.*
    - ▷ *How fun is a game and why?*
- **Try to use the techniques learned from this course in other areas!**

References and further readings

- R. M. Hyatt. Using time wisely. *International Computer Game Association (ICGA) Journal*, pages 4–9, 1984.
- R. Šolak and R. Vučković. Time management during a chess game, ICGA Journal, no. 4, vol. 32, pp. 206–220, 2009.