Fuego Go: The Missing Manual

Grace I. Lin

Independent Study Quarterly Report for Dr. David Helmbold
Fall Quarter, 2009

Purpose

The purpose of this report is to provide some information on how Fuego uses UCT search to generate a move given current board position. While Fuego provides developer documentation on their website, it is not intuitive on where modifications can be made. So it is my hope that the report may also be helpful in make modifications in the future.

Fuego Documentation

Fuego functions and classes are documented using Javadoc-style Doxygen syntax. This makes it easier to navigate through the code by clicking through links. The online Fuego documentation (http://www.cs.ualberta.ca/~games/go/fuego/fuego-doc/) reflects the current implementation. Users can also generate a local version of the Doxygen-style documentation reflecting the downloaded version.

Portability

- Standard C++
- External library: Boost
- C POSIX library – using C calling conventions

Code Naming Conventions

- Member variables use prefix m_
- Static variables use s_
- Global variables use g_

Fuego Libraries and Applications

Fuego is composed of five libraries and two applications. Figure 1 shows the module dependencies between them. For example, the library GtpEngine does not depend on any other module, while the library GoUct depends on three libraries: Go, SmartGame, and GtpEngine.

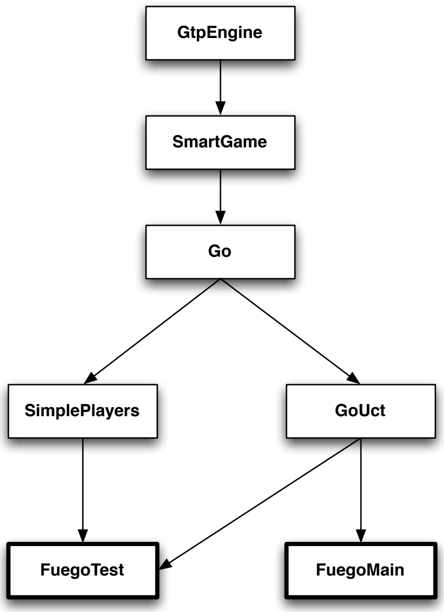


Figure 1: Fuego Libraries and Applications

Five libraries:

1. GtpEngine – implementation of Go Text Protocol (GTP); game-independent
2. SmartGame – utility classes/code can be shared between different 2-player board games
3. Go – Go specific classes
4. SimplePlayers – players with simple algorithms
5. GoUct – Go UCT player

Two applications:

1. FuegoTest– GTP interface with commands for testing purposes
2. FuegoMain– GTP interface to GoUctPlayer

Running Fuego in Command Line

Once Fuego is downloaded and compiled, we can play the game in the command line using provided GTP commands. The list of GTP commands can be found in each of the five classes:

- GtpEngine
- SgGtpCommands
- GoGtpEngine
- GoBook
- GoUctCommands

Reference: [*http://www.cs.ualberta.ca/~games/go/fuego/fuego-doc/fuegomain-doc/index.html*](http://www.cs.ualberta.ca/~games/go/fuego/fuego-doc/fuegomain-doc/index.html)

Here are some useful commands to get started: (% indicates the UNIX prompt; // indicates my comment)

% *fuego* // starts Fuego
% *go_board* // print info about current game board
% *komi <n>* // set komi
% *genmove <b/w>* // generate and play a move for black or white (see below)
% *gg-undo <n>* // undo past n moves
% *play <c> <m>* // player c makes move at a point (e.g. Q4)
% *showboard* // show board with player stones
%list_*stones* <b/w> // list stones for player

The command “genmove” is of particular interest to us since it is where all the magic happens. A top-level trace of function calls of this command is provided in the following section.

Generate Move with UCT Search

One of the challenges was going through the massive C++ code written in Fuego and trying to make sense out of it. This section presents a trace of top-level function calls to show how UCT search works through the command “genmove”. Hopefully this will help in making modifications to the code in the future.

Each box represents a function call, along with the corresponding class name and brief description. It focuses on generating a move using the UCT search given the current board position. It is basically a top-level trace of the command “genmove” mentioned above.

Note that this is NOT a complete sequence of function calls. Many details were left out of the diagrams.

The actual diagram is divided into the following order: 1) GTP command, 2) top-level search, 3) play game, 4) in-tree phase, and 5) play-out phase.

1. GTP command

In Figure 2, the “genmove” command enters the code at CmdGenMove() then onto GenMove() in the Go GTP Engine class. Fuego will try to lookup a move in the Go book first. If that did not generate a move, then the engine will try to generate a move based on the selected search mode:

- No search, use policy to select a move
- UCT search (our focus)
- One-ply Monte-Carlo search

As an example, the diagram is read as follow:

- CmdGenMove() calls GoGtpEngine::GenMove()
- GoGtpEngine::GenMove() first calls LookupMove(). If it did not generate a move, then it calls GoUctPlayer::GenMove().
- GoUctPlayer::GenMove() calls DoSearch()
- DoSearch() calls Search(), which is described in the following sections

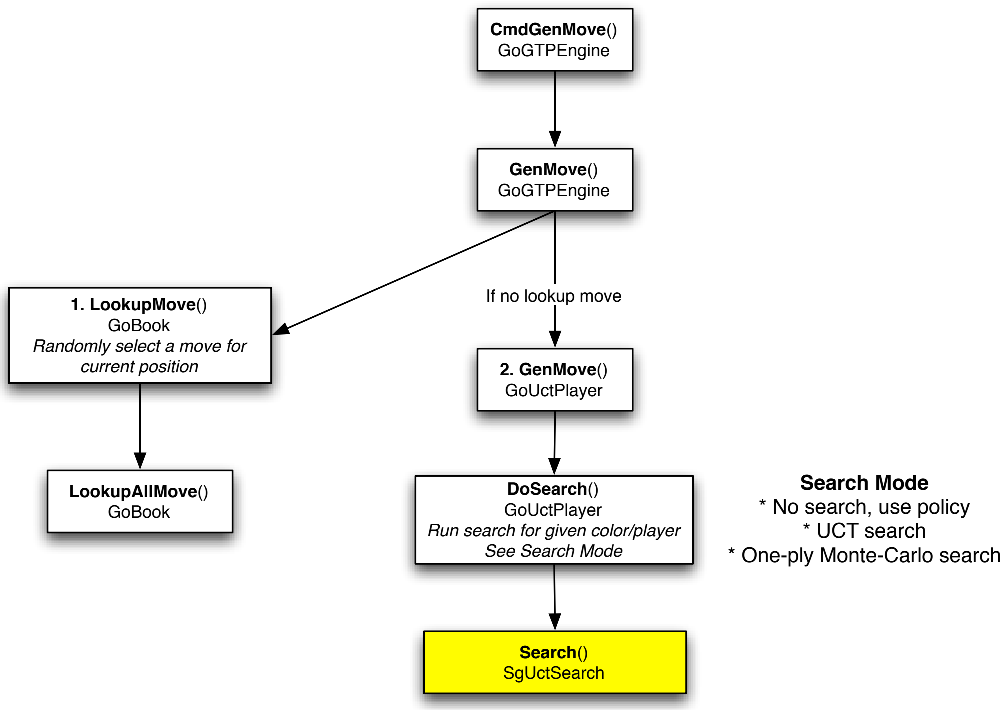


Figure 2: Generate Move (GTP Command) Diagram

2. Top-level search

Following up from the previous section, the UCT search (Figure 3) starts with game initialization and thread creation. Once the thread is started, Fuego enters a search loop to iteratively build a tree by playing games (see next section) . The loop is terminated when the tree cannot be expanded anymore. Once the search is done (play finished), prune nodes with low count, then proceed to find the best sequence of the tree.

The sequence is found by finding a best child node (representing the next move) of the current node in tree. The move selection strategy currently has four choices:

1. Select move with highest mean value (highest win-loss ratio)
2. Select move with highest count
3. Use UCT bound (combined with RAVE) : GetBound(), which is GetValueEstimate() plus UCT bias
4. Use weighted sum of UCT and RAVE value (no bias term) : GetValueEstimate()

Reference: Inside SgUctSearch class, find enumerated list SgUctMoveSelect

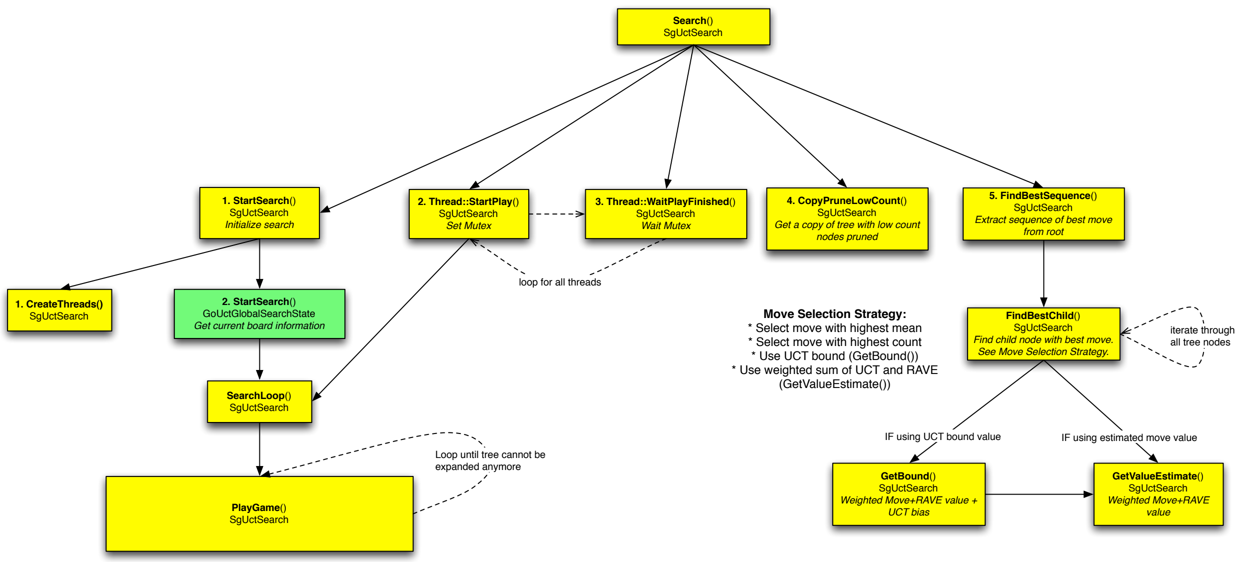


Figure 3: Generate Move (Top-Level Search) Diagram

3. Play a Game

The PlayGame() function is called repeatedly until the tree is fully expanded, as shown in

Figure 4. In includes a in-tree phase (PlayInTree()) and a play-out phase (PlayoutGame()). These simulated moves will be "undone" since they are not real moves. All the information generated remains. The game is terminated after two passes. It is scored with the Tromp-Taylor rule (a Chinese scoring method that assumes alls tones on the board are alive). Once the current game is finished, update the tree, RAVE values, and statistics.

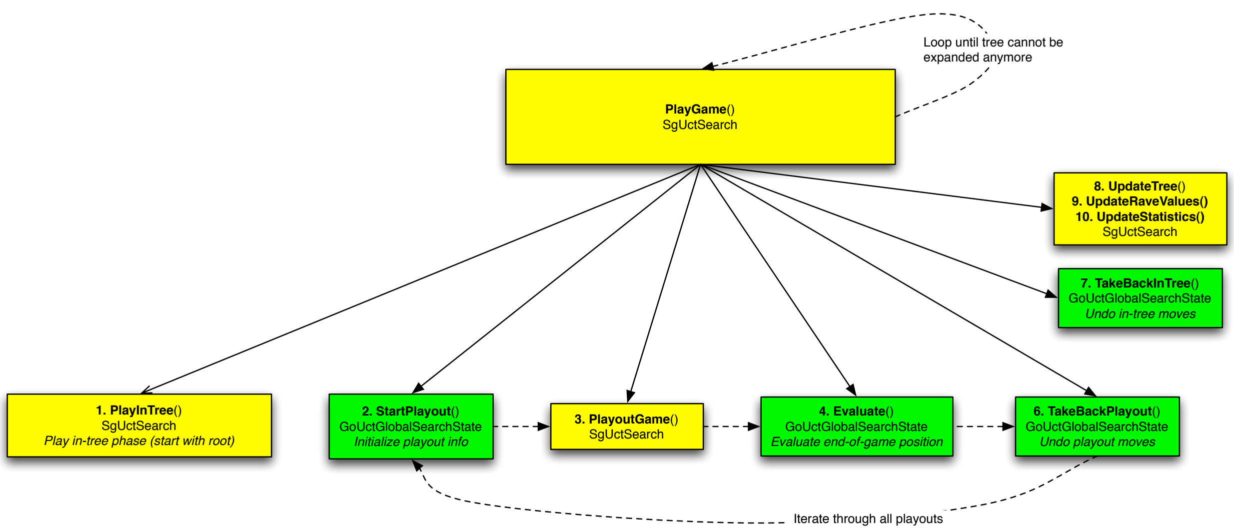


Figure 4: Generate Move (Play a Game) Diagram

4. In-Tree Phase

The in-tree phase (Figure 5) expands nodes until there is a proven win/loss. It generates legal moves, create children nodes, select the best child base on UCT bound (calls GetBound()), and finally executes the move. The loop continues until the last move produces and win or loss of the game.

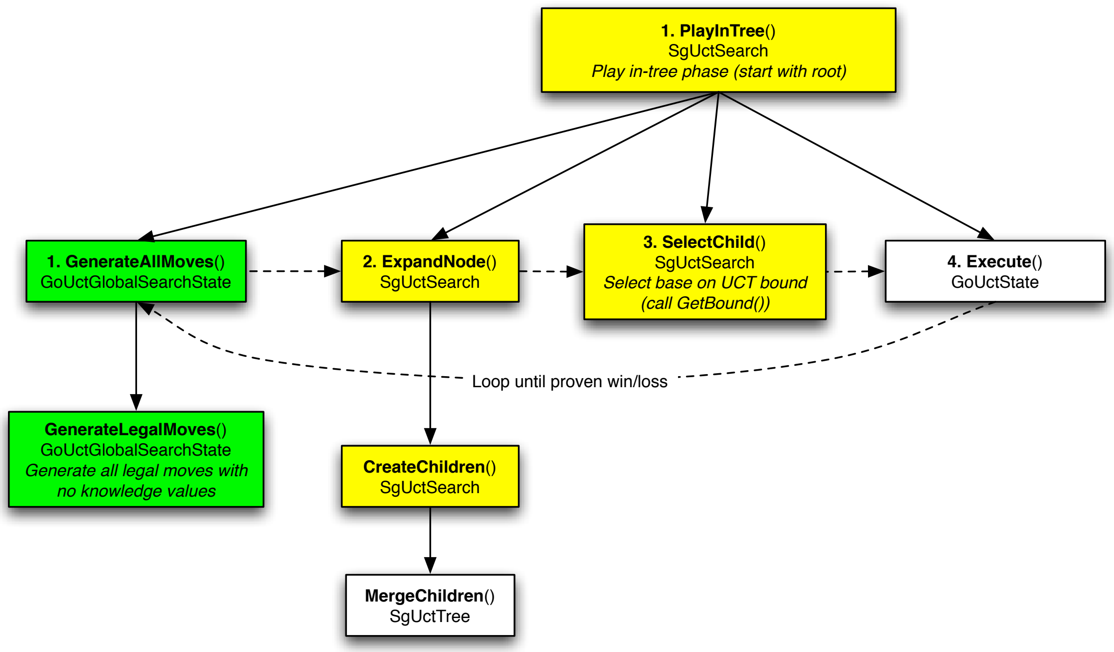


Figure 5: Generate Move (In-Tree Phase) Diagram

5. Play-out phase

Outside of the UCT tree, the play-out phase (Figure 6) tries to generate play-out moves based on the play-out policy. The play-out move is generated until a NULL move is generated (i.e., after a pass move was played). The play-out policy generates a move in the following order (from highest to lowest priority):

1. Nakade heuristic move
2. Atari capture move
3. Atari defense move
4. Low liberty move
5. Pattern move
6. Capture move
7. Random move
8. Pass move
9. NULL move

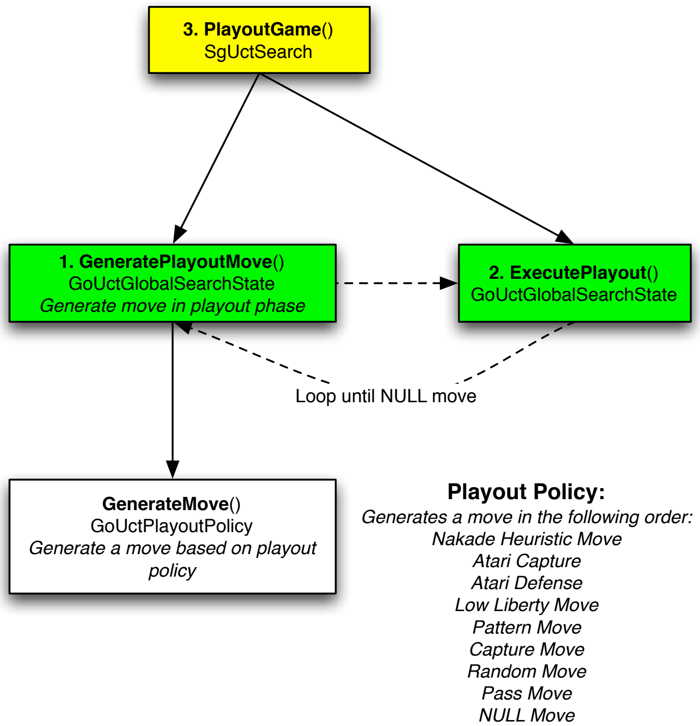


Figure 6: Generate Move (Play-Out Phase) Diagram

Update RAVE Values for Both Players

Reference: SgUctSearch::UpdateRaveValues()

RAVE store weighted game result to moves (tree nodes). It gives more weight to moves that are closer to the position that is currently being updated with RAVE statistics. Skip RAVE update is not currently supported in the in-tree phase.

According to the Fuego documentation, the weight function linearly decreases from 2 to 0 as the move gets further away from the position where RAVE statistics are stored. Here is the pseudocode:

Let *len*=length of sequence of current play (include both in-tree and playout sequences)
For position *i* in end position to first position of in-tree sequence
    Iterate through all children nodes (subsequent moves)
        Let *mv*=move of child node
        Let *first*=first time *mv* played in sequence (of the current player color)
        Calculate ***weight=2-(first-i)/(len-i)***
        Update child node's RAVE value = *weight* * *game result of playout(s)*

After analyzing the equation, it is unclear how *weight* could be less than 1. In order for *weight* <1, the ratio *-(first-i)/(len-i)* will have to be >1, which means *first>len*. This does not seem to make sense since *len* is the length of the sequence, and *first* has to be within the sequence. After playing with the code for a while, the weight never seems to go below 1. The documentation might have a typo, or maybe more analysis/playing with code is needed. Perhaps it is meant for future expansion?

Another interesting observation was that the position *i* seems to stay constant at 1 for some reason. This did not seem to make sense to me since in the code, position *i* decrements from the 2nd to last position in the playout sequence (which includes the in-tree sequence) until position 0. More experiments will be needed to convince myself that 1 makes sense.

Selecting the Best Move

As described earlier, Fuego allows for different options in picking the best move:
Reference: SgUctSearch::FindBestChild)

1. Move with highest mean value (average game result)
2. Highest move count (number of times the move leading to this position was chosen)
3. UCT bound (UCT bound formula) – see UCT Bound Formula section below
4. Weighted sum of UCT and RAVE value → see Estimator Weights in UCT Search section below

Item 3 and 4 are both described in the following sections.

Estimator Weights in UCT Search

The two estimators are the regular move value and the RAVE value. They are assumed to be uncorrelated. The weight of estimator *i* is described as follow:

$$w_i = \frac{1}{Z}\frac{1}{MSE_i}, \quad Z = \sum_i \frac{1}{MSE_i}, \quad MSE_i = \frac{C_{\mathrm{var}}}{N_i} + C_{bias}^2$$
$$where\ i = \{Move, RAVE\}\ Estimator$$

Reformulate to get the un-normalized weight. The variance and bias become constants (at least in Fuego) that describe the initial steepness and the final asymptotic value of the un-normalized weight

$$Z * w_i = \frac{1}{MSE_i} = \frac{1}{\frac{C_{\text{var}}}{N_i} + C_{bias}^2} = \frac{\frac{1}{C_{\text{var}}}}{\frac{1}{N_i} + \frac{C_{bias}^2}{C_{\text{var}}}} = \frac{C_{initial}}{\frac{1}{N_i} + \frac{C_{initial}}{C_{final}}}$$

with

$$C_{initial} = \frac{1}{C_{\text{var}}}, \quad C_{final} = \frac{1}{C_{bias}^2}$$

- *N* = sample count of the estimator (number of times the move leading to this position was chosen.)
- $C_{initial}$ = initial weight parameter when $N=1$ and $C_{final}>C_{initial}$; initial steepness
- C_{final} = final weight parameter when $N \rightarrow \infty$; final asymptotic value

The RAVE and regular move weight, as well as their relationship, are described in the following sub-sections.

1. **RAVE Weight:**

$$RAVE\ weight = \frac{C_{initial}}{\frac{1}{N_{RAVE}} + \frac{C_{initial}}{C_{final}}}$$

with

$$C_{initial} = \frac{1}{C_{\text{var}}}, \quad C_{final} = \frac{1}{C_{bias}^2}$$

In Fuego, the formula is further re-formulated as follow. By default, $C_{initialize}$ is 0.9 and C_{final} is 20,000.

$$RAVE\ weight = \frac{C_{initial}}{\frac{1}{N_{RAVE}} + \frac{C_{initial}}{C_{final}}} = \frac{N_{RAVE} * C_{initial}}{1 + \frac{N_{RAVE} * C_{initial}}{C_{final}}} = \frac{N_{RAVE}}{\frac{1}{C_{initial}} + \frac{N_{RAVE}}{C_{final}}}$$

$$= \frac{N_{RAVE}}{raveparam1 + raveparam2 * N_{RAVE}}$$

where

$$raveparam1 = \frac{1}{C_{initial}}, \quad raveparam2 = \frac{1}{C_{final}}$$

2. **Move Weight**

Bias is zero, and the variance become part of the normalization constant. This means the weight is just the sample count of the estimator.

$$
\begin{aligned}
&MOVE\ weight = N_{move} \\
&where \\
&C_{bias} = 0 = C_{final} \\
&MSE_{move} = \frac{C_{\text{var}}}{N_{move}} \\
&w = \frac{1}{Z} * \frac{1}{MSE_{move}} = \frac{N_{move}}{Z * C_{\text{var}}} \\
&Z * C_{\text{var}} * w = N_{move}
\end{aligned}
$$

3. Relationship between RAVE and move weight:

Based in the weights equation, RAVE weight will dominate initially, but eventually the regular move weight will dominate (Figure 7)

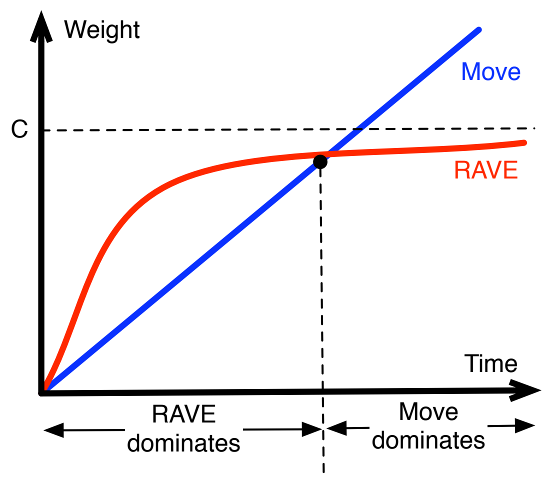


Figure 7: RAVE and Move Weights

UCT Bound Formula

The UCT bound value combines the estimated move value and the UCT bias. The estimated move value is reward for the move, and it is calculated as the weighted mean of regular move and RAVE values, using Move Weight and RAVE Weight equations described earlier. The move with the highest UCB bound is chosen as the best move.

$$UCT\ Bound = \overline{x_j} + c\sqrt{\frac{\log n}{T_j(n)}} = Estimated\ Move\ Value + UCT\ Bias$$

x_j = reward for move *j* = weighted mean of move value and RAVE value
j = move index
n = # times father node visited
$T_j(n)$ = # times move *j* has been played
C = appropriate constant (default is 0.7 in Fuego)

The Estimated Move Value and UCT Bias terms are described in the following sub-sections.

1. **Estimated Move Value**

Reference: SgUctSearch::GetValueEstimate()

Estimated move value is the weighted mean of regular move value and rave value, without RAVE bias

$$EstimatedMoveValue = \frac{Move\ Weight * Move\ Value + RAVE\ Weight * RAVE\ Value}{Move\ Weight + RAVE\ Weight}$$

where

$$Move\ Weight = N$$

$$Move\ Value = (1 - average\ game\ result)$$

$$RAVE\ Weight = \frac{N_{RAVE}}{raveparam1 + raveparam2 * N_{RAVE}}$$

$$RAVE\ Value = weighted\ average\ game\ result$$

Move Weight, RAVE Weight, and RAVE Value were described in earlier sections.

The average game result is the win-loss ratio of the node. The node represents the next move to be made by the opponent, and therefore we use (*1-average game result*) to minimize the win.

In case of unexplored moves (i.e., neither estimator has a sample count yet), use a pre-defined parameter value (*m_firstPlayUrgency*; default=10000). It may be set to a small value to encourage early exploitation.

2. **UCT bias**

Reference: SgUctSearch::GetBound(Node, ChildNode)
Node = represents position
Child Node = represents subsequent move

$$UCTbias = c * \sqrt{\frac{\log(positionvisited)}{1 + moveplayed}}$$

positionvisited = # of times node was visited
moveplayed = # of times the move was played, given position visited
c = 0.7 by default (as described in the original UCT paper)

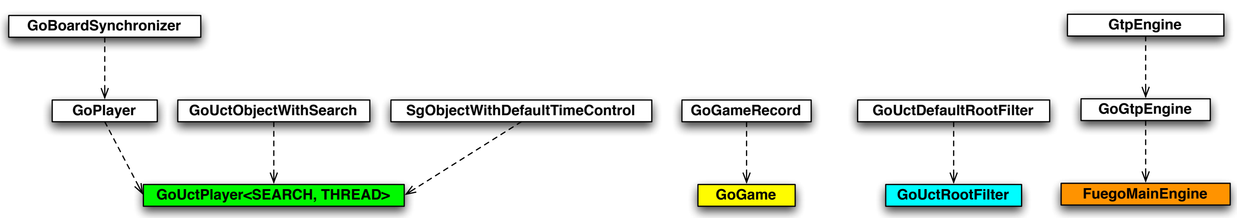
The denominator is added with 1 to avoid dividing by zero.

Other Useful Information

This section presents some helpful information in navigating through Fuego.

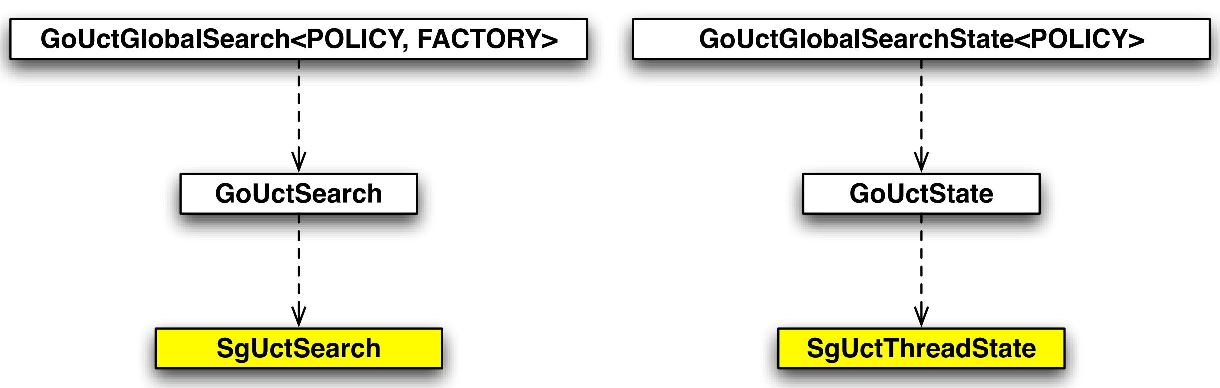
1. Inheritance Diagrams

- UCT player inherits from a regular player, along with search and timer control.
- Game inherits from game record.
- UCT root filter (in detecting ladder) inherits from the default root filter.
- Fuego main engine inherits from Go GTP engine and the default GTP engine.



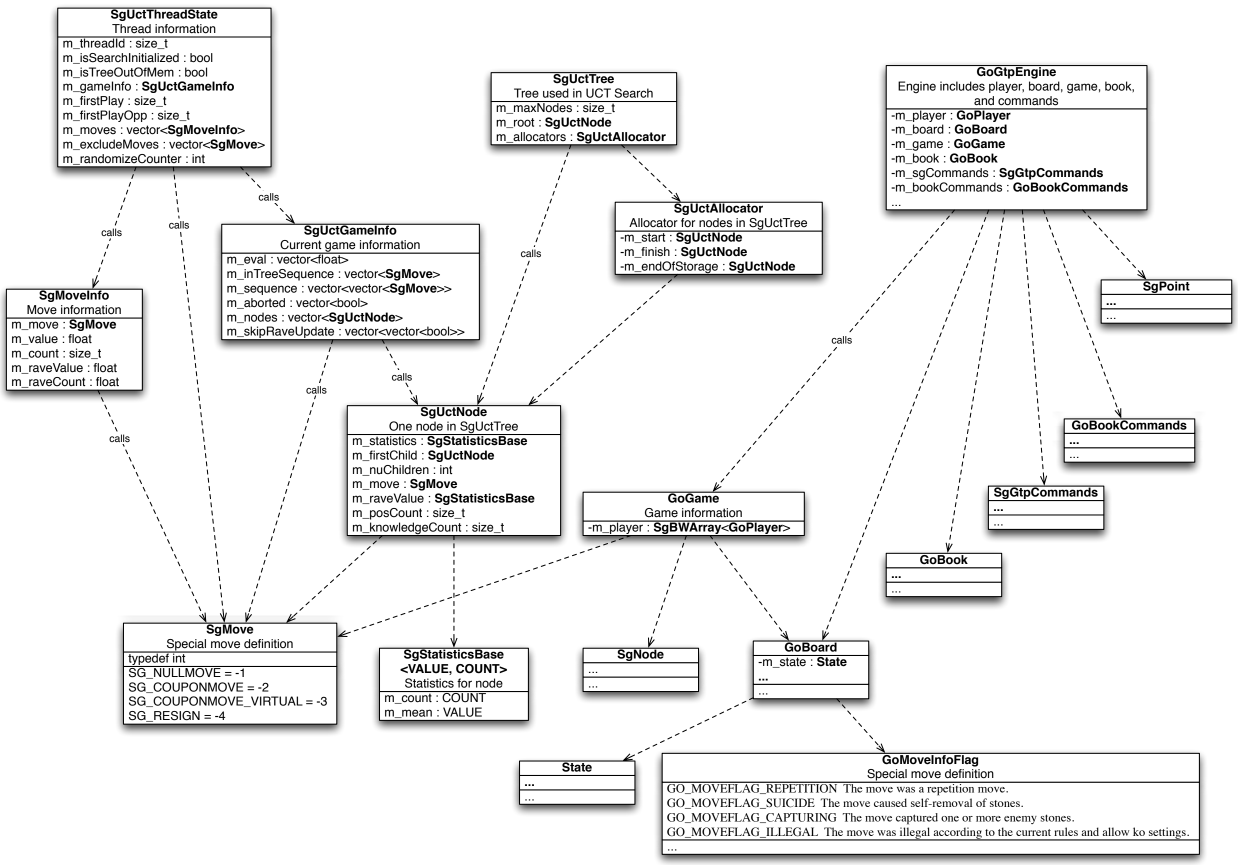
The following two class inheritances are by far the most useful in studying Fuego:

- UCT search inherits from UCT global search and Go UCT search.
- UCT thread state inherits from global and Go UCT state.



2. Class Diagrams

This is an attempt to capture relationships between some important classes (not a complete version).



3. Board representation

- 1D array
- Neighbors of a point: offset *SG_WE* and *SG_NS*

	P+SG_NS	
P-SG_WE	*P*	*P+SG_WE*
	P-SG_NS	

- Points & coordinates: *SgPointUtil::Pt, SgPointUtil::Row, SgPointUtil::Col*
  - Pt(1,1) = 21 = Location 'A1', lower left corner of board
  - *SgPoint.h* (default) point numbers

```
19|381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399
18|361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379
17|341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
16|321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339
15|301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
14|281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299
13|261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
12|241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259
11|221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
10|201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219
 9|181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
 8|161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 7|141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
 6|121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
 5|101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
 4| 81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
 3| 61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 2| 41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59
 1| 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
   [A] [B] [C] [D] [E] [F] [G] [H] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
```

4. UCT Patterns

Reference: GoUctPatterns<BOARD> class template

- Hard-coded pattern matching routines to match patterns used by MoGo
- See http://hal.inria.fr/docs/00/11/72/66/PDF/MoGoReport.pdf
- Move is always in center of pattern or at middle edge point (lower line) for edge patterns
- Patterns matched for both colors, unless specified otherwise

Notation:

```
O = White            x = Black or Empty
X = Black            o = White or Empty
. = Empty            B = Black to Play
? = Don't care       W = White to Play
```

- Patterns for **Hane**: return true if any patern is matched

```
X O X   X O .   X O ?   X O O
. . .   . . .   X . .   . . .
? ? ?   ? . ?   ? . ?   ? . ? B
```

- Patterns for **Cut1**: return true if 1st pattern is matched, but not next two

```
X O ?   X O ?   X O ?
O . ?   O . O   O . .
? ? ?   ? . ?   ? O ?
```

- Pattern for **Cut2**

```
? X ?
O . O
x x x
```

- Pattern for **Edge:** return true if any pattern is matched

```
X . ?    ? X ?    ? X O      ? X O        ? X O
O . ?    o . O    ? . ? B    ? . O W      O . X W
```

5. Playing games between two Go programs

GoGUI reference: http://gogui.sourceforge.net/doc/reference-twogtp.html

Command: gogui-twogtp

Example: A simple shell script that plays 5 games between GNU Go (black) and Fuego (white).

Note: GNU Go uses Japanese rules (territory counting) by default, while Fuego uses Chinese rules (area counting) by default. GNU Go can play at different levels up to level 10 (highest and most accurate level). By default it plays at level 10.

```
set NUMGAMES=6
set BSIZE=9
set KOMI=7.5
set FILENAME="fuego_gnugo"

gogui-twogtp -black "gnugo --mode gtp --chinese-rules" \
            -white "fuego" \
            -games $NUMGAMES \
            -komi $KOMI \
            -size $BSIZE \
            -alternate \
            -sgffile $FILENAME \
            -auto
gogui-twogtp -analyze $FILENAME.dat
```

Options used (see website reference for full list of options available):

-black	Command for the black program
-white	Command for white program
-games	Number of games to play
-komi	Set komi
-size	Board size
-alternate	Alternate colors; Black and White are exchanged every odd game; scores saved in SGF file keeps the name for Black and White given at command
-sgffile	Prefix of the SGF file(s); each game is saved with filename *prefix-n.sgf*, where *n* is the game number
-auto	Automatically play games
-analyze	Create a HTML formatted result page of the games played

The resulting files:

fuego_gnugo-0.sgf
fuego_gnugo-1.sgf
fuego_gnugo-2.sgf
fuego_gnugo-3.sgf
fuego_gnugo-4.sgf
fuego_gnugo-5.sgf
fuego_gnugo.dat
fuego_gnugo.html : summary, results, and links to all games played
fuego_gnugo.summary.dat

Area counting versus territory counting: TODO

6. Useful websites for Go information:

- Sensei's Library: http://senseis.xmp.net/ (pretty much anything we need to know about Go)
- Computer Go Resources: http://computer-go.info/
- List of computer Go tournaments: http://computer-go.info/events/index.html

Top MCTS computer Go programs:

Go Program	Recent Achievement
1. SilverStar (Japanese edition of KCC Igo)	2009 UEC Cup winner
2. Zen	2009 Computer Olympiad winner
3. CrazyStone	2007 & 2008 UEC Cup winner
4. Many Faces of Go	2008 Computer Olympiad winner
5. Fuego	November 2009 KGS winner; 1st computer program to win an official game of 9x9 Go against a 9-Dan professional player
6. MoGo	2007 Computer Olympiad winner

GNU Go is not a top-ranked program, but it is a free program with well-documented manual.

Some Terminologies

SGF file format:
Smart Go Format for computer-recorded go games

Liberty

A vacant point immediately adjacent to a stone either directly up, down, left, or right from it, or connected through a continuous string of same-colored stones to such a point.

Atari
A situation where a stone or chain or stones has only *one liberty*, and may be captured on the next move if not given one or ore additional liberties.

Self-atari / auto-atari
Placing a single stone in a position where it only has one liberty.

Komi
Black has the advantage of first move. To compensate, white can be given an agreed, set number of points (called komi) before starting the game.

Joseki
Established sequences of play considered optimal result to both players. Thousands of lines researched and documented.

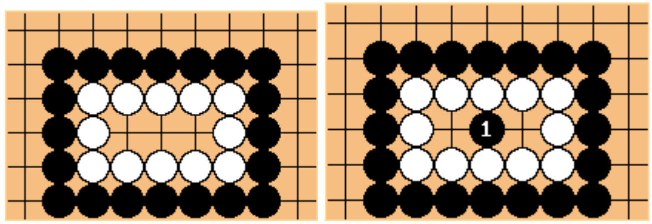
Seki
Term describe an impasse that cannot be resolved into simple life and death. For example, capturing race end in a position in which neither player can capture the other.

Life and death
A fundamental concept in Go where the status of a distinct group of stones are determined as "alive" or "captured".

Factory design pattern:
- http://en.wikipedia.org/wiki/Factory_method_pattern

Hashing of positions:
- GNU GO 14.2 Hashing of positions
- http://www.delorie.com/gnu/docs/gnugo/gnugo_169.html#IDX352
- High occurrence: previously checked position is rechecked, of ten from different branch in recursion tree → waste computing resources
- Hash (or transposition) table: Store current position, function we are in, result of search; which move made attack/defense succeed
  - Key: Go position
  - Data: results of reading for certain functions and groups

Nakade:
- "Inside move" or "move inside"
- Crucial to life and death
- Refers to a situation where a group has a single large internal, enclosed space that can be made into two eyes by the right move, or prevented from doing so by an enemy move
- Can be designated the actual move that prevents the two-eye formation



Source: http://senseis.xmp.net/?Nakade

References

M. Enzenberger and M. Müller. "Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search". Technical Report TR 09-08, Dept. of Computing Science. University of Alberta, Edmonton, Alberta, Canada, 2009.

M. Enzenberger and M. Müller. "A lock-free multithreaded Monte-Carlo tree search algorithm". Advances in Computer Games 12, Pamplona, Spain, 2009.

Fuego Developer's Documentation: http://www.cs.ualberta.ca/~games/go/fuego/fuego-doc/