Fuegito

User Manual

Colin Hunt

2012

Table of Contents

Introduction

Fuegito is designed from the start to be a simple, universal framework for implementing games and search algorithms. Specifically, Fuegito is concerned with two-player, deterministic games that use perfect information and game-tree search algorithms. In this section we will explore the design goals of Fuegito as well as the framework's main functionality.

Design Goals

The principal design goals for Fuegito are as follows.

Keeping Similarity with Fuego

Fuegito was conceived in part to act as an analogue with Fuego[1], an orders of magnitude more complex, professional-level, game-playing program. To this end, Fuegito was designed with a similar overall structure to its modules and classes. Same or similar names have been retained as those of Fuego to reinforce this idea. All this is done to allow an easier transition for those who wish to step-up to developing for the more complex Fuego. Literally, Fuego is fire and Fuegito is little fire.

Simplicity

While Fuegito was inspired by Fuego, Fuegito is primarily concerned with simplicity in its design and ease of use. This means that some of the more powerful features of Fuego and some of the performance are sacrificed in the name of keeping things straightforward. This is done to allow Fuegito to be more accessible to an introductory level audience. Those interested in starting with game programming can start with an existing framework and quickly program up a new game or search algorithm quickly and easily.

Extensibility

Fuegito should be highly extensible and flexible. Users should be able to implement any new game, any new player, and/or any new search and use them in any combination with the existing games, players, and searches. Fuegito is meant as a starting point to build from; it should make it as easy as possible, with as few barriers as possible.

Provide a Framework for Teaching

Fuegito's simplicity and extensibility make it an ideal framework for teaching purposes. Anyone with some introductory C++ knowledge, or those who are currently learning C++, should find Fuegito accessible to them. This makes Fuegito ideal as a teaching tool in a game-programming course concerned with the fundamentals. Fuegito's extensibility means it will also be relevant in a more advanced course. An instructor or teaching assistants can more quickly program the specific games and algorithms that his or her course is concerned with in order to study them; they could also assign these tasks to the students as an exercise.

Easy to Combine Different Games and Algorithms

Fuegito uses general interfaces between all its games, players, and search algorithms. This ensures that any specific implementations will work with each other. New games can be played with any player, new players can play any game, etc. All specific functionality is encapsulated in derived classes that use a public interface. Game-specific functionality in algorithms should be provided by the game classes themselves, freeing the algorithms from those details.

Provide Reasonable Default Implementations but allow Game-Specific Overrides

Fuegito is not just a hollow framework. It provides a number of concrete implementations for games, searches, and players to enable you to immediately start playing with Fuegito. If all you want is to implement a new game, you can use existing players and game-playing algorithms with it right away. Or, if you just want to see how a game-specific enhancement improves alpha beta search with Clobber, just write the improvement and get instant feedback. Or write completely new games, searches, and players ignoring all the existing implementations.

Standalone Open Source Program Written Entirely in Standard C++

Fuegito is open source and written entirely in C++, using nothing but the Standard Library. This mirrors Fuego as well as enables Fuegito to be highly portable across multiple platforms with no external dependencies.

Main Functionality

A Framework For 2-player Deterministic Games with Perfect Information

Fuegito is concerned with modeling two-player, deterministic games with perfect information. Any game that fits these criteria should be implementable in Fuegito.

Library of Simple Standard Game-Playing Algorithms and Players

Next to the games, Fuegito is designed to implement game-tree search algorithms to operate on these games and players that use algorithms to play these games.

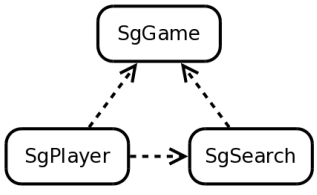
Interaction with External Users and Programs

Fuegito does not exist in isolation. It provides two interfaces that allow it to interact with the outside world. The first is Go Text Protocol, or GTP, which is a popular standard for game text interfaces[2]. The second is a simple graphical UI provided as a separate program that communicates with Fuegito via GTP commands. Please see the relevant sections for more information.

Overview of Main Classes and Their Interaction

Fuegito is divided up into three main modules and one application. The modules are grouped under the 'SmartGame' moniker that is a naming convention used in Fuego. The modules are SgGame, SgSearch, and SgPlayer. Each module defines an interface through which the other modules can interact with it. The interface for each module consists of public methods of a primary abstract base class, global constants, and custom defined classes and types.

General Overview



The dependencies between the modules form a triangle, as shown in Figure 1. SgGame is at the top, with no dependencies; SgSearch depends on SgGame; and SgPlayer depends on SgGame and SgSearch. Figure 2 shows the class relationships.

FIGURE 1: FUEGITO MAIN MODULES.

SgGame

SgGame is abstract and depends on no other classes or modules. It defines the interface for representing a game in Fuegito. Concrete game classes can inherit directly from SgGame. SgGridBasedGame is another abstract class that inherits from SgGame and provides data and methods that are general to games that use playing grids, such as a Go or Chess board. A concrete grid-style game inherits from SgGridBasedGame. An example will follow below.

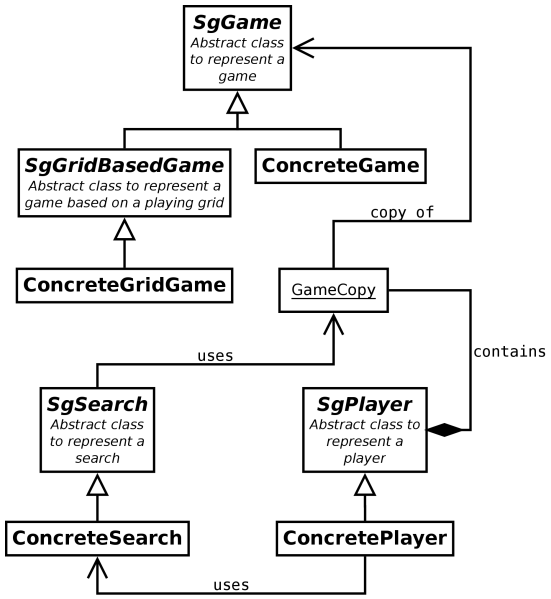


FIGURE 2: MAIN CLASS RELATIONSHIPS.

SgSearch

SgSearch is an abstract class that defines the interface for a search algorithm. Concrete searches can inherit directly from SgSearch. SgSearch contains a pointer to an external SgGame-derived object that it performs the search on. Fuegito provides a considerable number of specific implementations of different searchs.

SgPlayer

SgPlayer is an abstract class that defines the interface for a player. Concrete players can inherit directly from SgPlayer. SgPlayer contains a copy of an external SgGame-derived object that represents the game that it is playing. Concrete players can use an SgSearch object to do a search of the game to help it pick a move to play. This is done by instantiating a search object that uses the player's copy of the game, running the search, getting the result, and then destroying the search object. Because the search object is only created when it is needed, it frees the player from being restricted to one search type at a time. The player object can instantiate as many search objects of various types as it needs to aid in selecting a move. For example, a player could start off using Monte Carlo search and then switch to Alpha Beta in the end game. See the MixedMcAbPlayer class for an example of this.

Using these relationships, Fuegito allows great flexibility between games, searches, and players. Any search can be used with any game, any game with any player, and any player with any search.

TicTacToe Example

For an illustration of the class relationships on the code level, we will look at how a round of Tic Tac Toe would be played.

Suppose you elect to play a game of Tic Tac Toe against one Alpha Beta player who plays with a depth setting of 3. First, a TicTacToeGame instance needs to be created to represent the game

being played between two players. TicTacToeGame is a concrete class that inherits from SgGridBasedGame.

```
  TicTacToeGame ttt;
```

Next, an AlphaBetaPlayer instance is created and we pass the constructor our game object, which side the player is on (white or black), the depth that we want to search to, and the traditional settings for alpha and beta. Also given is a name for our player. AlphaBetaPlayer inherits from SgPlayer. Within the logic for generating a move, we would see that AlphaBetaPlayer uses an instance of AlphaBetaSearch. AlphaBetaSearch derives from SgSearch.

```
  int depth = 3;
  double alpha = -DBL_INFINITY, beta = DBL_INFINITY;
  AlphaBetaPlayer abPlayer(ttt, SG_BLACK, depth, alpha, beta, "ShallowHall");
```

For illustration purposes, we will setup a simple game loop that will handle displaying the board, getting moves from the user, asking the player to generate moves, and playing those moves in the game. The loop also checks to see when the game has ended. The FuegitoEngine class is responsible for implementing such a loop with GTP functionality.

```
while (!ttt.EndOfGame()) {
  ttt.Print(cout);
  SgMove humanMove = GetHumanMove(cin);
  ttt.Play(humanMove);
  ttt.Print(cout);
  if (ttt.EndOfGame()) break;
  SgMove aiMove = abPlayer.GenerateMove();
  ttt.Play(aiMove);
}
```

Here is what is going on:

*ttt.*EndOfGame() is tested at each iteration of the loop to detect when the game is over.

*ttt.*Print() is called to display a text-representation of the board to the standard output stream.

The user's move they wish to make is received.

*ttt.*Play(humanMove) is called to record the user's move and *ttt.*Print() is called again to display the change.

*abPlayer.*GenerateMove() is now invoked in order to get *abPlayer*'s move choice. GenerateMove() makes a copy of the outside game instance for the player that then uses that copy in conducting the search. GenerateMove() invokes the player-specific functionality by calling DerivedGenerateMove().

```
  SgMove DerivedGenerateMove() {
    AlphaBetaSearch abSearch(Board(), Color(), Hash(), Depth(), Alpha(), Beta());
    SgMove move = abSearch.GenerateMove();
    return move;
  }
```

Inside DerivedGenerateMove(), *abPlayer* creates an AlphaBetaSearch instance *abSearch* that, using getter methods, takes a pointer to the player's internal copy of the game, the player's

color, the player's hash table (not seen in this example), the search parameters, and returns a pointer to an alpha beta search object.

To actually select a move, *abPlayer* now calls *abSearch.*GenerateMove(). GenerateMove() in the search object traverses the game tree to a depth level of 3. At first this is not deep enough, so when it runs out of depth it calls *ttt.*Evaluate(). Evaluate() is a game-specific method that, in this instance, evaluates the Tic Tac Toe position that the search is currently looking at. It returns a value to the search indicating how good or bad the position is for the player.

When the search is done, the player returns the resulting move to the game loop.

*ttt.*Play(aiMove) and *ttt.*Print() are once again called to reflect the player making its move.

This process repeats until the game is over.

Details and Extensions

There are many details of the Fuegito main interfaces that need to be discussed, as well as some of the major extensions in functionality that Fuegito provides.

Main Interfaces

As mentioned previously, Fuegito provides three main interfaces. They are SgGame, SgSearch, and SgPlayer. The framework also provides its own type definitions to use with the interfaces.

Defined Types

SgGame

- **SgGameDefines.h**
  - SgBlackWhite
    An SgState type that only takes black or white values.
  - SgBlackWhiteEmpty
    SgState type that takes black, white, and empty values.
  - SgMove
    Type to represent a move as an integer.
  - SgState
    State of a game element (white, black, empty, border).
- **SgGridBasedGameDefines.h**
  - SgGrid
    A type representing grid-related values (row#, column#, #rows, #columns, etc.).
  - SgPoint
    Type that represents a point that can index into the game board.

SgSearch

- **SgSearch.h**
  - SgSearchHashTable
    Hash table used in SgSearch classes.

SgUtilities

- **SgHash.h**
  - SgHashCode
    64-bit type that represents a hash code.

- **SgTimer.h**
  - SgTimerValue
    Type to use for values with SgTimer.

SgGame Interface

The following methods define the interface for interacting with an SgGame object.

Methods

```
SgGame* Copy() const
```
Virtual copy constructor.

```
bool Play(SgMove move)
```
Plays a move specified by *move*. For turn-based games, plays the move for the current side to play and then switches the current side. Returns whether or not the move was successfully played.

```
bool Play(SgBlackWhite color, SgMove move)
```
Plays a move and color combination.

```
bool TakeBack()
```
Takes back the last played move, restoring the game to the condition it was in before the move was made. Returns whether or not the take back was successful.

```
bool EndOfGame() const
```
Returns whether or not the game is over.

```
bool HasWin() const
```
Returns whether or not there is a winner.

```
void Generate(vector<SgMove>& moves) const
```
Generates all legal moves for the current game position and stores them in the vector pointed to by moves.

```
void GenerateAll(vector<SgMove>& moves) const
```
Generates the set of all possible moves that can arise during a game and stores it in the vector pointed to by moves. Unlike *Generate()*, not all moves in this list may be legal for the current position. This is the superset for all possible move lists.

```
bool Legal(SgBlackWhite color, SgMove move) const
```
Returns whether or not move is a legal move that can be played from the current position and color.

```
SgBlackWhite GetToPlay() const
```
Returns the current side to play—black or white.

```
SgBlackWhiteEmpty GetWinner() const
```
Returns the winner of the game—black, white, or empty. Returns empty on a draw or if there is currently no winner.

```
SgHashCode GetHashCode() const
```
Returns the SgHashCode representation of the current game position. This function allows storing of game positions and information in a hash lookup table for use in search algorithms.

```
void Print(ostream& out) const
```
Writes the text representation of the current game position to the stream pointed to by *out*.

```
double Evaluate() const
```
Evaluate the current position as being good or bad for the current player. Used by search algorithms.

```
void SwitchToPlay()
```
Changes which side is currently the side to play.

Example

For an example, please see the Tic Tac Toe example in the previous chapter.

SgGridBasedGame Interface

SgGridBasedGame extends SgGame with functionality specific to games that use playing grids.

Methods

```
SgGrid GetRows() const
```
Return the number of rows in the game.

```
SgGrid GetCols() const
```
Return the number of columns in the game.

```
SgPoint GetSize() const
```
Return the size of the game board as the number of points on the board including border points. If default size settings are used, the board size may be much larger than the actual playing size.

```
SgState GetState(SgPoint p) const
```
Returns the state of the board at the point *p* as black, white, empty, or border.

```
SgState GetState(SgGrid row, SgGrid col) const
```
Same as above except the point is specified by a row *row* and column *col.*

SgSearch Interface

The SgSearch interface is very brief as it only consists of one method.

Methods

`SgMove GenerateMove()`
Instructs the search algorithm to generate and return a legal move that can be played for the search's color in the current game position. The search's color is the color it was constructed with or the current side to play for the game when it was constructed. If it cannot generate a legal move for whatever reason, it should return SG_NULLMOVE.

Example

Suppose *MySearchAlgorithm* inherits from SgSearch. In general, for any SgSearch-derived search object, the following demonstrates how to use it to perform a search of a game and get a move.

```
TicTacToeGame ttt;                          // Initialize the game.
SgBlackWhite color = SG_BLACK;              // Color to search the game for.
MySearchAlgorithm mySearch(ttt, color);     // Create the search instance and pass the
                                            //   game and color to the constructor.
SgMove move = mySearch.GenerateMove();      // Get a move using GenerateMove().
```

Specific search algorithms may define more parameters that they take in their constructor, but the basic steps are the same.

SgPlayer Interface

The SgPlayer interface is similarly brief.

Methods

`SgMove GenerateMove()`
Instructs the player to generate and return a legal move that can be played for the player's color in the current game position. If it cannot generate a legal move for whatever reason, it should return SG_NULLMOVE.

`const string& Name() const`
Returns the name of the player given when it was constructed.

Example

Suppose *MyPlayer* inherits from SgPlayer. In general, for any SgPlayer-derived player object, the following demonstrates how to use it to get a move to play in the current game.

```
TicTacToeGame ttt;                              // Initialize the game.
SgBlackWhite color = SG_BLACK;                  // Color of the player's side.
MyPlayer myPlayer(ttt, color, "MyPlayerName"); // Create the player instance and pass
                                                //   the game, color, and name to the
                                                //   constructor.
SgMove move = myPlayer.GenerateMove();          // Get a move using GenerateMove().
```

Specific players may define more parameters that they take in their constructor, but the basic steps are the same.

Extensions

There are several extensions that enhance Fuegito, including specific games, search algorithms, players, tools and utilities.

Specific Games

Fuegito provides two sample game implementations of Tic Tac Toe and Clobber.

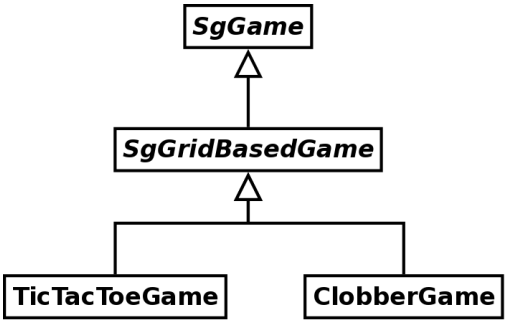


FIGURE 3: INHERITANCE DIAGRAM FOR GAMES.

Tic Tac Toe

Tic Tac Toe (or Noughts and Crosses for the Brits) provides a simple demonstration of a game that can be implemented with the Fuegito framework and used with the provided search algorithms. Its simplicity makes the game a good testing ground for new search algorithms, as correct behavior can be determined more easily.

Clobber

Created in 2001 by Michael H. Albert, J.P. Grossman and Richard Nowakowski[3], Clobber is a relatively new game about which comparatively little is understood[3]. Fuegito provides an opportunity to test the performance of a multitude of different search algorithms with Clobber to see what, if any, primary strategies emerge in play.

Configuring ClobberGame

An instance of clobber can be configured for the number of rows and columns the game board should have, as well as which side is to move first. Example:

```
int rows = 5;
int cols = 6;
SgBlackWhite toPlay = SG_WHITE;

ClobberGame clobber(rows, cols, toPlay);
```

Specific Search Algorithms

Fuegito provides several implementations of different search algorithms. There are the simple searches that provide basic functionality, and then there are Alpha Beta and Monte Carlo searches with some of their variations.

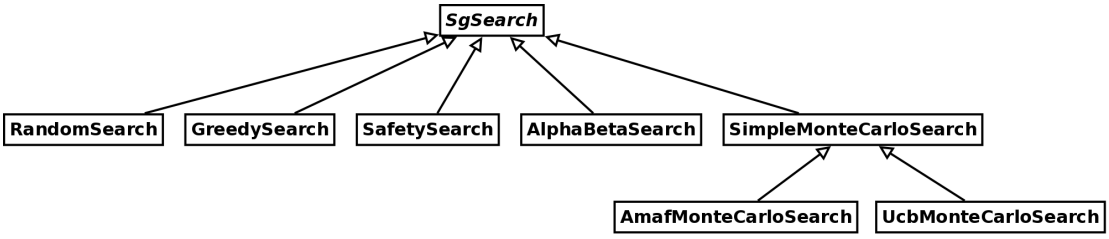


FIGURE 4: SEARCH CLASSES INHERITANCE DIAGRAM.

The Simple Searches

Three simple 1-ply or 2-ply search algorithms are provided that use very basic logic to generate a move. They can be used for simple players or as part of a more complex algorithm.

RandomSearch

A very basic 'search' method. Selects a move at random to play. Can be used as part of other algorithms when a random move is needed. See SimpleMonteCarloSearch for a sample use case.

GreedySearch

GreedySearch is greedy in the sense that it tries to find a move that will give an immediate win (performs a 1-ply win check).

The Algorithm

1. For each move in the current move list
    a. Play the move
    b. Record if won
    c. Take back the move
    d. If won:
        i. Return the move
        ii. Else go to the next move
2. If no winning moves found, return a null move

SafetySearch

SafetySearch is still a greedy search except that now it is looking for the move that will prevent an opponent from winning on the next turn (2-ply loss check).

The Algorithm

1. For each move in the current move list
    a. Play the move
    b. Do a GreedySearch for the opponent to see if they have a winning move
    c. Take back the move
    d. If the opponent has a winning move:
        i. Go to the next move
        ii. Else return the move
2. If no safe moves found, then return a null move

AlphaBetaSearch

Alpha beta search is a classic game-tree search method that has been used in games for decades[4]. It is used extensively in games such as checkers and chess, and famously in the Deep Blue chess-playing program[5]. Alpha beta is a more efficient way to compute the minimax value of a game compared to the standard Minimax algorithm. It uses parameters called alpha and beta to record the lower and upper bounds for the best value that the player can hope to achieve. If it encounters a move that allows the player to do better than or equal to beta, or for the opponent to do better than or equal to alpha, the search can stop exploring the current branch. In the former case, the opponent can force a worse move on the player; in the later, the player can avoid this line of play. This pruning of branches can significantly reduce the number of nodes that needs to be searched in a game tree.

The Algorithm

Fuegito uses the Negamax[1] implementation of Alpha Beta search.

[1] http://en.wikipedia.org/wiki/Negamax

1. Check if game is over for current position. If it is, then return negative infinity.
2. Check if we have run out of depth. If so, call the evaluation function on the current position and return the result.
3. Generate a list of all legal moves from the current position.
4. For each move in the list:
    a. Play the move.
    b. Recursively perform an alpha beta search of the new position but with alpha and beta switched and negated. Also negate the returned result.
    c. Take back the move.
    d. Check if the value returned in the recursive search is better than seen so far. If it is, record the move as the best move and the value as the best value.
    e. If the value is greater than or equal to beta, break out of the loop.
5. Return the best value and the best move.

Configuring AlphaBetaSearch

AlphaBetaSearch needs to be constructed with a game reference. All other values have default arguments. The constructor looks like:

```
AlphaBetaSearch(
  Game& game,                      // Reference to the game to be searched
  SgBlackWhite color = SG_EMPTY,   // The color of the side to search for.
                                   //   Default is the current side to play.
  SgSearchHashTable* hash = 0,     // Pointer to a hash table.  Optional.
  int depth = DEFAULT_DEPTH,       // Maximum depth to search to.
  double alpha = -DBL_INFINITY,    // Alpha parameter.
  double beta = DBL_INFINITY       // Beta parameter.
)
```

SimpleMonteCarloSearch

The SimpleMonteCarloSearch algorithm is based on the fundamental principle of the Monte Carlo method. As related to game playing, the basic idea is that when presented with a number of moves, one can find the winning probability of each move with random simulations. A simulation is nothing more than a random playout to the end of the game starting with one of the moves, with the result recorded as a win or loss. After enough simulations are run on each move, the probabilities will begin to converge and the algorithm picks the move with the highest win rate.

Default Functionality

Default functionality is provided in the SimpleMonteCarloSearch base class. It selects moves to be played in order so that each move gets an equal number of tries. During a playout moves are selected completely at random. Win rate information is only tracked for the initial moves generated in step 1 (the moves that are played first). The best move is picked by considering only the win rate, nothing more. Please see SimpleMonteCarloSearch under Implementing Your Own Search for details on how to provide custom functionality.

Configuring SimpleMonteCarloSearch

The constructor takes a game reference, and all other values have default options.

```
SimpleMonteCarloSearch(
  Game& game,                              // Reference to the game to be searched.
  SgBlackWhite color = SG_EMPTY,           // Color of the side to search for.
                                           //   Default is side to play.
  const MoveTimer* timer = 0,              // Move timer set to the max search time.
```

```
                                          //   Optional.
  int nuSimulations = 0,                  // Min number of simulations to run per
                                          //   move. Supersedes time setting.
  MonteCarloSimulationPolicy* simPolicy = 0  // Pointer to simulation policy to use.
                                          //   Default is simple random.
)
```

Variations

There are a number of variations to the basic Monte Carlo algorithm. The two examples that are implemented by Fuegito are AMAF Monte Carlo search and UCB Monte Carlo search.

AmafMonteCarloSearch

AMAF stands for All-Moves-As-First[6]. Relating to the Monte Carlo algorithm, it is a strategy that updates move statistics for moves played anytime during the playout, not just first moves. This is called the AMAF update, in contrast to the standard update. The idea is that moves played later in the game that resulted in a win could be good moves in general, and thus good moves for the current position. This is counterintuitive at first; however, experiments do confirm that this approach is beneficial in games like Go (the effect may be game-dependent, however). The primary benefit is that many more samples are generated about a move for use in the final selection policy; therefore the statistics should be more reliable.

Fuegito implements a version of the AMAF heuristic called *alpha*-AMAF. This variation does both a standard update and an AMAF update, maintaining two tables of statistics about each move. It then uses a parameter called *alpha* to combine the mean value of a move from the standard update with the AMAF value of the move. The formula for computing the value is

$$alpha \cdot AMAF\_value + (1 - alpha) \cdot standard\_value$$

Where a*lpha* takes values between 0 and 1. Note that *alpha* = 0 gives the standard algorithm and *alpha* = 1 gives the AMAF algorithm.

Configuring AmafMonteCarloSearch

AmafMonteCarloSearch's constructor takes identical arguments to SimpleMonteCarloSearch, with the addition of the alpha parameter, which defaults to 1.

```
AmafMonteCarloSearch(
  Game& game,
  SgBlackWhite color = SG_EMPTY,
  const MoveTimer* timer = 0,
  int nuSimulations = 0,
  float alpha = 1     // Alpha parameter.
)
```

UcbMonteCarloSearch

Upper-Confidence-Bound or UCB Monte Carlo search uses an upper confidence bound selection strategy for determining how to select a move to explore further[6]. The move that has been explored the most overall is then selected as the best move to be returned. The move value consists of the win rate (*Wr*) plus a variance term that gives a high-confidence upper bound on the true move value. The variance term is computed as the margin of error (*Me*) plus an adjustment (*Adj*), where the margin of error is based on a given critical value (*Cv*) that corresponds to the desired confidence interval.

$$\text{UCB} = Wr + Me + Adj$$

Where $Me = Cv\sqrt{\frac{Wr(1-Wr)}{\#plays}}$, and $Adj = \frac{1}{\sqrt{\#plays}}$.

Configuring UcbMonteCarloSearch

UcbMonteCarloSearch's constructor takes identical arguments to SimpleMonteCarloSearch, with the addition of the critical value to use.

```
UcbMonteCarloSearch(
  Game& game,
  SgBlackWhite color = SG_EMPTY,
  MoveTimer* timer = 0,
  int nuSimulations = 0,
  double criticalValue = UCB_CRITICAL_VALUE    // Critical value to use for confidence
)                                              //   bound.
```

Specific Players

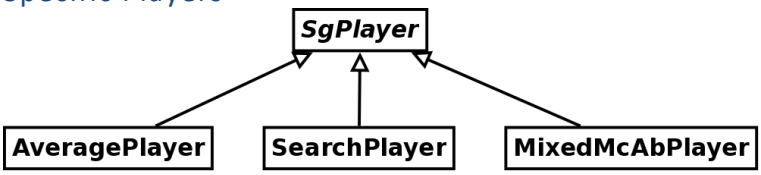


FIGURE 5: INHERITANCE DIAGRAM FOR PLAYERS.

AveragePlayer

AveragePlayer is a simple example of a player that uses multiple search techniques together to find a move to play. It first tries to do a GreedySearch to find an immediate win for itself. Failing that, it does a SafetySearch to try to avoid an immediate loss. If it still cannot find a move, it uses RandomSearch to return a random move. This results in a playing strength that is well below average in anything other than Tic Tac Toe, but do not tell it that. Perhaps in the future it will be renamed WbAveragePlayer.

MixedMcAbPlayer

This is another example of a player that uses multiple search techniques. Specifically, this player attempts to combine alpha beta and Monte Carlo search techniques into one strategy. It first will run a Monte Carlo search, which also records the depth of the game tree, and plays the returned move. On the next turn, it will check the depth of the tree against the depth setting for its alpha beta search. If the depth is shallow enough to use alpha beta, it will do so. If not, it will use Monte Carlo again. Therefore, the effect is that for the first moves of a game it plays like a Monte Carlo player, and then switches to an alpha beta player in the end stage.

Configuring MixedMcAbPlayer

MixedMcAbPlayer takes a game reference, a color for its side, a pointer to a move timer, a pointer to a hash table, the minimum number of simulations to run per move, the maximum depth to search to with alpha beta, and alpha and beta values.

```
MixedMcAbPlayer(
  const Game& game,
  SgBlackWhite color,
  const MoveTimer* timer = 0,
  SgSearchHashTable* hash = 0,
  int nuSimulations = DEFAULT_NU_SIMS,
  int depth = DEFAULT_DEPTH,
  double alpha = -DBL_INFINITY,
  double beta = DBL_INFINITY
)
```

SearchPlayer

SearchPlayer is a generic player that can use any search given to it via a search factory. It eliminates the need to create a separate player for each different search method. SearchPlayer will simply use the given search factory to create an instance of that search type and then generate and return a move using it.

Search Factories

Search factories are what allow the SearchPlayer to create the desired search type on the fly. It has a MakeSearch() method which returns an instance of the desired type. Simple searches can be added to SimpleSearchFactory, and more advanced searches can inherit from SgSearchFactory or one of its subclasses.

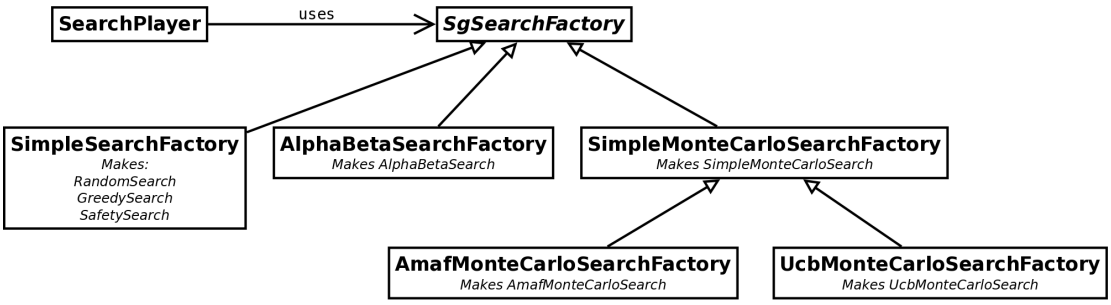


FIGURE 6: INHERITANCE DIAGRAM FOR SEARCH FACTORIES.

Modifying Search Parameters in Existing Factory Instances

One can modify an existing factory's search-specific values using accessor methods (see below). This allows changing the behavior of a search player using the search factory midgame. For example, the Depth() accessor method can be called on AlphaBetaSearchFactory which returns a reference to the depth setting, allowing it to be altered. See below for details.

SimpleSearchFactory

This type of factory can produce any basic search object that takes only the 4 standard arguments: a game reference, color to search for, hash table pointer, and move timer pointer. Its constructor takes a string representation of the search type to create. See how to extend this class to create additional search types in Using and Extending Fuegito: Implementing Your Own Search: Creating a Search Factory.

Accessor Methods

```
string& SearchType()
```
Returns a reference to the type of search being produced. Change this value to change what type of search the SearchPlayer is using.

AlphaBetaSearchFactory

This makes an instance of AlphaBetaSearch. The constructor can be passed the desired search depth along with alpha and beta values.

Accessor Methods

The following methods allow depth, alpha, and beta parameters to be read and changed.

```
int& Depth()
```

```
double& Alpha()
```

```
double& Beta()
```

SimpleMonteCarloSearchFactory

An instance of SimpleMonteCarloSearch is created with this factory. The constructor can be passed the minimum number of simulations desired.

Accessor Methods

```
int& NuSimulations()
```
Allows the minimum number of simulations parameter to be read and changed.

In general, search factories function by passing the search-specific arguments to the constructor, and then passing the factory instance to the SearchPlayer constructor. The search settings can then be modified with accessor methods.

Configuring SearchPlayer

As with other players, a SearchPlayer must have a reference to the game, and the color for the side to play for. For example, to configure an Alpha Beta SearchPlayer, assuming the game has already been created first create the search factory:

```
int depth = 5;
AlphaBetaSearchFactory abFactory(depth);
```

Then create the SearchPlayer and pass in the factory instance by reference:

```
SearchPlayer player(game, SG_BLACK, &abFactory);
SgMove move = player.GenerateMove();
```

If you wanted to change the search depth to be deeper:

```
abFactory.Depth() = 9;
SgMove newMove = player.GenerateMove();
```

The new move is now generated by searching to a depth of 9.

Move Timer

A general implementation of a simple timer is provided with the SgMoveTimer class. It can be used to control the time for an entire game, individual players, and how long searches can take.

Interface

```
SgMoveTimer(SgTimeVal time = 0)
```
Constructor. Accepts a time value in seconds to initialize the timer with. The timer is initially stopped when first constructed.

```
void Set(SgTimeVal time)
```
Sets the timer for a given time in seconds.

```
void Start()
```
Starts the timer. The timer will begin running down the time. When time 0 is reached, it will run into negative values showing how much overtime has elapsed.

```
void Stop()
```
Stops the timer.

```
SgTimeVal TimeLeft()
```
Returns the remaining time left on the timer in seconds. Negative values indicate amount of overtime that has elapsed.

Configuring a Move Timer

For example, we can set the game time for a MixedMcAbPlayer which will limit how long it has to make a move. For a 5-minute game, initialize the timer to 5 minutes, and then construct the player passing in the timer by reference:

```
SgMoveTimer timer(5 * 60);
MixedMcAbPlayer player(game, color, &timer);
```

Start the timer and ask the player to generate a move:

```
timer.Start();
SgMove move = player.GenerateMove();
```

The player uses a percentage of the remaining game time to make its move. Stop the timer again and print the time left to standard out:

```
timer.Stop();
cout << timer.GetTimeLeft() << '\n';
```

The result in this case is 210 seconds left (the player used 30% of the total time).

Hash Table

A Zobrist hash table[7] is implemented with SgHashTable, SgHashEntry, and ZobristNumbers classes. This allows searches to store and lookup current or past results, allowing the ability to recognize transpositions in move ordering and also improve move ordering—thus becoming more efficient.

SgHashTable, SgHashEntry, SgSearchHashData classes

These classes are virtually identical to Fuego's implementation of them. Please see Fuego documentation[1] for more information about how these work.

ZobristNumbers

A global singleton class that returns a Zobrist number for a given index. Used in generating a unique hash code for a game position. Currently there is a hard-coded limit of 1,500 on the index value. From the Fuego documentation:

> *The hash index ranges from [0..MAX_HASH_INDEX-1]. For board games with black and white pieces, MAX_HASH_INDEX needs to be bigger than twice the number of points on the board. [It is] up to the client to map points to this range.*

For 2-player games with more variety in piece types, the formula for computing the max hash index is $2 \cdot \#piece\_types \cdot \#board\_points$. So in TicTacToe, for example, there is only really one piece type and it can be black or white. The board contains 9 points. So the max hash index required is $2 \cdot 1 \cdot 9 = 18$. For a more complex game like Chess, there are 6 piece types and 64 board points giving a max index of $2 \cdot 6 \cdot 64 = 768$.

Configuring and Using a Hash Table

Let us use the example of the SearchPlayer to show how to have it use a hash table to improve alpha beta search. Assume we have already set up the search factory with the settings we want there and we are ready to initialize our hash table. The hash table constructor takes the desired maximum hash entries, the number of distinct piece types in the game, and an optional hash statistics class (see SgHashStatistics). We use the default hash size and number of pieces that are defined for Clobber:

```
int maxHash = CLOBBER_HASHSIZE;
int nuPieces = CLOBBER_NU_PIECES; // = 2
SgSearchHashTable hash(maxHash, nuPieces);
```

Then create the player as before, this time passing in the hash table by reference, and generate a move:

```
SearchPlayer player(game, SG_BLACK, &abFactory, &timer, &hash);
SgMove move = player.GenerateMove();
```

The player now automatically utilizes the hash table with its search as long as the search is programmed for it. Else, the table is ignored.

SgHashStatistics

Information can be gathered about the hash table using the SgHashStatistics struct. To use it, simply pass in the statistics object by reference to the hash table on construction:

```
SgHashStatistics hashStats;
SgSearchHashTable hash(maxHash, nuPieces, &hashStats);
```

Use the hash as before; during use, SgHashStatistics records data about the hash performance. To print the data, you can call PrintStats() directly or use the instance with an output stream:

```
hashStats.PrintStats(cout);
cout << hashStats;
```

What it records:

```
  /** Number of collisions on store */
```

```
  int m_nuCollisions;

  /** Total number of stores attempted */
  int m_nuStores;

  /** Number of successful stores */
  int m_nuNewStores;

  /** Total number of lookups attempted */
  int m_nuLookups;

  /** Number of successful lookups */
  int m_nuFound;

  /** Records the entry size in memory of the hash */
  int m_entrySize;

  /** The number of entries the hash contains */
  int m_hashSize;
```

Monte Carlo Simulation Policy

A simulation policy dictates the way moves will be played in the simulation of a Monte Carlo search. For example, moves could be played completely randomly or game-specific knowledge could be used to avoid blunders. Thus, a Monte Carlo search can be made to perform differently given a different simulation policy.

Interface

`SgBlackWhiteEmpty PlayGame(vector<SgMove>& playedMoves)`
Performs a simulated playout to the end of the game. Calls GenerateMove() to generate moves for the playout. Records the playout in the *playedMoves* vector reference. Returns the result of the game as black, white, or empty in the case of a draw.

`SgMove GenerateMove()`
Method to generate and return a move for use in PlayGame(). This is where bias can be introduced into the playout, or generated moves can be simple random.

Configuring and Using a Policy

The simulation policy constructor takes a reference to the game being played:

```
SimpleMonteCarloSimulationPolicy mySimPolicy(game);
```

Then the search constructor takes a reference to the simulation policy instance:

```
SimpleMonteCarloSearch mcSearch(game, color, &timer, nuSimulations,
                                &mySimPolicy);
```

The search can then be used as any other.

SgGridBasedGameRecord

This class sets up and manages the interaction of a game, two players, two search factories, and two game timers. In short, everything needed to control a game between two players. This class is used by the user interfaces in order to control the game behind-the-scenes.

GtpInterface

GTP communicates via text commands. The GtpInterface class is meant to facilitate communication between GTP and Fuegito by acting as a translator. For example, different games have different move types and thus different text representations of their moves. Subclasses of GtpInterface can implement methods to translate Fuegito values such as an SgMove value for a specific game into a text representation for GTP.

Interface

```
GtpInterface(SgGridBasedGameRecord* gameRec)
```
Constructor. Takes a reference to a game record that the interface is interpreting for.

```
bool Play(const string& move)
```
Plays a string-encoded move. Formatting is game specific. Returns whether or not it was successful.

```
bool Play(const string& color, const string& move)
```
Plays a string-encoded move with a string-encoded color. Returns as above.

```
virtual string MoveToString(SgMove move) const = 0
```
Converts an SgMove move into a string-encoded move and returns it. Game dependent.

```
virtual SgMove StringToMove(const string& strMove) const = 0
```
Converts a string-encoded move into an SgMove move and returns it. Game dependent.

```
string GenMoveWhite()
```
Generates a move for white that is played and returns it as a string.

```
string GenMoveBlack()
```
Generates a move for black that is played and returns it as a string.

```
string GetState(SgGrid row, SgGrid col)
```
Gets the state of a board point given by (*row, col*) and returns the string representation.

Using and Extending Fuegito

Fuegito is designed to be added to and extended. There are several ways that this can be done.

Choices for Extending Fuegito

1. Implement a New Game
2. Implementing a New Search Algorithm
3. Implement a New Player
4. Add Extensions and Customize Searches

Implementing Your Own Game

Fuegito has been designed to make it very easy to implement a new game that will automatically work with all existing Fuegito classes and default algorithms. Some design choices need to be made first, and then it is a simple matter of plugging the game into the framework. Additionally, there are game-specific improvements that could be implemented to enhance the performance of the game-playing algorithms with the game.

Plugging it Into the Fuegito Framework

Fuegito defines an interface with the SgGame class for implementing games that all existing player classes and search algorithms use for interaction. Creating a new game in Fuegito is as easy as inheriting from the SgGame class and overriding the 15 pure virtual methods there.

Classes To Inherit From

Fuegito provides a primary abstract class SgGame and a more specialized abstract derived class SgGridBasedGame to be inherited from for implementing a specific game.

SgGame

The SgGame class provides a general interface for implementing two-player games. The majority of its methods are undefined, and it contains no data. SgGame provides a good starting point for subclasses that implement *types* of games (grid game, card game, etc.) from which further subclasses would refine into specific games.

SgGridBasedGame

Games that are based on a set of points arranged in a grid-like pattern of some kind (Go, Chess, etc.) should inherit from the SgGridBasedGame class. This class provides predefined methods and supporting data for implementing this type of game. See the documentation on this class for more information about its features.

Methods to override

Concrete game classes must override pure-virtual methods depending on which class they inherit from in order for the game to function within the framework.

SgGame

A concrete derived class of SgGame must override all the methods defined in the class' interface as documented in Details and Extensions. Caution: there are two versions of the Play() method, and one of them has a default implementation. Thus, if the derived class does not override the defined version, it will need to bring the other one into scope with a using declaration.

SgGridBasedGame

A concrete derived class of SgGridBasedGame must override all the same methods as for inheriting from the SgGame class (see above) except for the following that SgGridBasedGame provides implementations for already.

Exempt Methods

```
SgHashCode GetHashCode() const

void Print(ostream& out) const

void SwitchToPlay()
```

Please see SgGridBasedGame Interface under Details and Extensions for a list of accessor methods that subclasses can use.

Implementing Your Own Search

To implement a new search algorithm in Fuegito, you can either write a new search class by inheriting from SgSearch directly, or you can modify and extend one of the existing search classes. Currently, only SimpleMonteCarloSearch can be extended from. After the search class is created, you can create or modify a search factory for your search that will allow it to be used with the SearchPlayer.

Classes to Inherit From

SgSearch

The SgSearch class provides an interface for implementing search algorithms that operate on games where players make discrete moves that alter the game state. It also provides support for using a hash table and move timer when conducting the search.

SimpleMonteCarloSearch

Inheriting from SimpleMonteCarloSearch allows a custom variation of the Monte Carlo algorithm to be implemented. See Extensions and Details and How to do Improvements for more details.

Methods to Override

Methods in SgSearch

```
SgMove GenerateMove()
```

This is the only method that needs to be overridden.

Accessor Methods in SgSearch

SgSearch provides accessor methods that allow subclasses to access some of the class resources such as the hash table and move timer.

```
SgGame& State()
```

Returns a reference to the current board 'state' (really just a reference to a copy of the player's board instance).

```
SgSearchHashTable* Hash()
```

Returns a pointer to the hash table if there is one, else returns a null pointer.

```
const SgMoveTimer* Timer()
```

Returns a pointer to the search timer if there is one, else returns a null pointer.

```
SgTimeVal GetTimeLeft()
```

Returns the time left in the search timer in seconds. If no search timer was given, returns 0.

```
SgBlackWhite OwnSide()
```

Returns the side of the search (the side that is being searched for) as SG_BLACK or SG_WHITE.

Methods in SimpleMonteCarloSearch

SimpleMonteCarloSearch provides default functionality for all its methods. Thus, derived classes can override as many or as few as needed in order to effect the desired behavior. See How to do Improvements for more details on implementing your own Monte Carlo variation.

Creating a Search Factory

To use your search with the SearchPlayer, you need to write a factory that will allow SearchPlayer to make an instance of your search. This is also done through inheritance. In general, every search algorithm needs a corresponding search factory that defines how to create an instance of that algorithm.

Inheritance

To make a new search factory, you must inherit from the SearchFactory class. For extending AlphaBeta and SimpleMonteCarloSearch, you can inherit from AlphaBetaFactory or SimpleMonteCarloSearchFactory instead. SearchFactory declares a pure virtual factory method called MakeSearch() that subclasses can override with the details of how to make their particular search type.

Override MakeSearch() method

The *MakeSearch()* method must be overridden in the derived class. Its signature is

```
SgSearch* MakeSearch(SgGame& game, SgBlackWhite color,
                     SgSearchHashTable* hash,
                     const SgMoveTimer* timer) const
```

The method takes a reference to a game object, a pointer to a hash table object, and a pointer to a move timer object. The subclass should define how to return its matching search instance created with *new*.

The hash pointer and move timer pointer can be ignored if your search does not use them.

Examples

Please see SearchFactory.h and .cpp to see how the provided search factories are done.

SimpleSearchFactory

Additional search types can be added to SimpleSearchFactory by adding if statements to the MakeSearch() method like the following:

```
  if (m_searchType == "my_search")
    return new MySearch(game, color, hash, timer);
```

Use the appropriate arguments for your search.

Implementing Your Own Player

There are a couple of different options to choose from when creating your own player with Fuegito. One way is to inherit from SgPlayer and have full control over how your player behaves. The second way is to inherit from SearchPlayer and override the GetMoveTimer() method in order to modify how it does time management.

Methods to Override

SgPlayer

The only method that needs to be overridden in SgPlayer is DerivedGenerateMove().

```
SgMove DerivedGenerateMove()
```

This method is called to get a move when GenerateMove() is called in the SgPlayer base class. All this method has to do is return a valid legal move for the game being played, or return SG_NULLMOVE if no valid moves remain. Use the base-class accessor methods to access the current game, hash table, player color, time left, and game timer (see SgPlayer.h). The SgPlayer base-class handles these resources for the derived classes automatically, including copying of the external game so that the player's state is always consistent.

SearchPlayer

SearchPlayer uses the GetMoveTimer() method to get a timer for its search method. Override this method to control how the timer is set with regard to the external-game timer, which will affect how long the SearchPlayer has to generate a move.

```
SgMoveTimer* GetMoveTimer() const
```

This method returns an SgMoveTimer pointer that SearchPlayer passes to the search factory to make a search. This method can return a null pointer to signify that the search timer should not be used. The timer should be set with the maximum time that the search should take to generate a move. Note that not all searches use timers and may choose to ignore the timer.

Accessor Methods in SgPlayer

```
SgGame& Board()
```

Returns a reference to the player's copy of the game.

```
SgBlackWhite Color() const
```

Returns the player's color.

```
SgTimeVal GetTimeLeft() const
```

Returns the time left in seconds for the player's game timer. If no timer was given, returns 0.

```
const SgMoveTimer* GameTimer() const
```

Returns a const pointer to the player's game timer.

```
SgSearchHashTable* Hash() const
```

Returns a pointer to the player's internal hash table.

Registering Your Class with the User Interfaces

The user interfaces in Fuegito rely on the SgGridBasedGameRecord class to setup a game with players and searches. Therefore, to use a new class with the existing UIs, you need to register it with SgGridBasedGameRecord. This process has been streamlined with Fuegito's dynamic class registration system.

What is Involved

Two things must done to register a class in Fuegito: a declaration macro needs to be placed in the class' private section, and the method for creating an instance of your class needs to be placed in the .cpp file with the definition macro.

Declaration Macros

The appropriate macro needs to be placed in the class' private section depending on what base class your class inherits from.

```
SG_REGISTERED_GAME
```

```
SG_REGISTERED_PLAYER

SG_REGISTERED_SEARCHFACTORY
```

Definition Macros

The appropriate definition macro needs to be placed in your class' .cpp file along with a function body that returns an instance of your class via *new*. All macros take two arguments: the name of your class as a string, and the exact typename of your class.

```
SG_REGISTER_GAME("game_name", GameType) {
  return new GameType(rows, cols);
}
```

The game macro provides two arguments, *rows* and *cols,* to use when instantiating your game.

```
SG_REGISTER_PLAYER("player_name", PlayerType) {
  return new PlayerType(*game, color, factory, timer);
}
```

The player macro provides arguments *game, color, factory,* and *timer* to use.

```
SG_REGISTER_SEARCHFACTORY("search_name", SearchType) {
  return new SearchType();
}
```

The search factory macro does not provide any arguments, so be sure to provide an appropriate constructor.

In all cases any unneeded arguments may be ignored.

Example using Average Player

In AveragePlayer.h, AveragePlayer inherits from SgPlayer and defines its constructor to take a const SgGame reference and a color. It calls the SgPlayer constructor with the game reference, color, and its name. It also declares itself for registration with the SG_REGISTERED_PLAYER macro.

```
class AveragePlayer : public SgPlayer {

  SG_REGISTERED_PLAYER;

public:
  AveragePlayer(const SgGame& game, SgBlackWhite color)
    : SgPlayer(game, color, "AveragePlayer") {}

/* ... */
};
```

In AveragePlayer.cpp, AveragePlayer overrides DerivedGenerateMove() and uses GreedySearch, SafetySearch, and RandomSearch in order to try and generate a good move for itself. If none of the searches find a valid move, the method returns SG_NULLMOVE.

```
  SgMove DerivedGenerateMove() {
    GreedySearch gSearch(Board(), Color());
    SgMove move = gSearch.GenerateMove();
    if (move == SG_NULLMOVE) {
      SafetySearch sSearch(Board(), Color());
      move = sSearch.GenerateMove();
    }
    if (move == SG_NULLMOVE) {
      RandomSearch rSearch(Board(), Color());
```

```
      move = rSearch.GenerateMove();
    }
    return move;
  }
```

Next, the SG_REGISTER_PLAYER macro is used to define how to instantiate an AveragePlayer class:

```
SG_REGISTER_PLAYER("average", AveragePlayer) {
  return new AveragePlayer(*game, color);
}
```

How to do Improvements

Fuegito allows you to effect various improvements in the way the search algorithms play. Two examples that are described here are writing your own variation of the Monte Carlo algorithm, and writing an evaluation function for Alpha Beta tree search.

Write your own variation of Monte Carlo

There are a number of variations of the basic Monte Carlo algorithm. With this in mind, the basic algorithm was written following the template method. Implementing a specific variation is supported through inheritance and overriding of virtual methods. In this way, derived classes of SimpleMonteCarloSearch can provide their own specific functionality for various steps of the basic algorithm.

The Algorithm Template

The steps of the algorithm template are listed here along with the method names. Almost every step can be overridden in a derived class.

1) Get a move list from the current position. *State().Generate()*
2) Create data tables that keeps a record for each move of the number of times the move has won when played, and the number of times in total the move has been played. *Constructor()*
3) Run simulations on the move list: *PlayGames()*
    a) While there is time left or have not reached minimum number of simulations:
        i) Select and play a move from the move list according to some selection strategy. *SelectMove()*
        ii) Do a playout on the resulting position to the end of the game according to a simulation policy (i.e. random, biased). m_simPolicy->*PlayGame()*
            (a) Moves in the playout are generated by *GenerateRandomMove()*
        iii) Get the result of the playout (win, loss, or draw)
        iv) Update the move statistics based on the result of the playout. *UpdateData()*
            (a) Default behavior is to update the mean data. *UpdateMeanData()*
4) Select a best move according to some selection strategy (e.g. by picking the move with the highest win rate from the move data table). *GetBestMove()*
    a) Value for the move is determined by an evaluation policy. *GetValue()*
5) If for some reason the algorithm fails to pick a move, a null move is returned and must be handled. *HandleNullMove()*
    a) Default behavior is to return a random move.

Please see AmafMonteCarloSearch.h and UcbMonteCarloSearch.h for examples of how these steps in the algorithm can be overridden.

Examples

AmafMonteCarloSearch and UcbMonteCarloSearch will be examined in more detail for examples of how derived classes can alter the behavior of the base algorithm. Please see Details and Extensions for details of how these algorithms work.

AmafMonteCarloSearch

AmafMonteCarloSearch both provides new data and overrides methods in defining its behavior.

Data Members

The class keeps an AMAF table and an alpha parameter as data for use in its algorithm.

m_amafTable

This is a second move data table that stores results from moves played anywhere in the playout.

m_alpha

This is the alpha parameter—the fraction of the AMAF value to use in the final move value.

Overridden Functions

The AmafMonteCarloSearch class overrides the following two functions with its implementation.

UpdateData()

Now updates the AMAF table as well as also calling UpdateMeanData() to perform the standard update. In this way it keeps the two data sets separate.

GetValue()

Returns a combined value from both the standard mean table and the AMAF table. The ratio between the two values can be set with the alpha parameter.

UcbMonteCarloSearch

UcbMonteCarloSearch defines new constants, data, overridden functions, and new functions for its implementation.

Constants

UCB_CRITICAL_VALUE

Default critical value to use in calculating the upper confidence bound.

Data Members

m_criticalValue

Critical value corresponding to a desired confidence interval to use when computing the upper confidence bound.

Overridden Functions

SelectMove()
Returns the move with the highest upper confidence bound to use for running the simulation. Calls *GetBound()*.

GetBestMove()
Selects the best move to play by finding the one that has been played most often. This should correspond to a high move value and a tight upper confidence bound.

New Functions

GetBound()
Computes the upper confidence bound of a given move value. Called by *SelectMove()*.

Simulation Policy
You can write your own simulation policy to use with the Monte Carlo search classes. Simply inherit from the SgMonteCarloSimulationPolicy class and override the PlayGame() and GenerateMove() methods.

See the SimpleMonteCarloSimulationPolicy class for an example.

Write an Evaluation Function for Alpha Beta Search
When Alpha Beta search runs out of depth before it reaches the end of the game, it uses an evaluation function to estimate the value of the position. The accuracy of this function can make or break the effectiveness of alpha beta. Writing a good evaluation function for a particular game is essential for alpha beta search to perform well on it.

Override *Evaluate()* Method in SgGame Class
The evaluation function is a member of the SgGame class because it is inherently game dependent. This ensures that game-dependent code is kept with the game.

```
virtual double Evaluate() const
```
Evaluates the current game position using some fast method and returns the value. Positive values being good for the current player and negative values being bad.

Building and Running Fuegito
There are two methods to communicate with Fuegito. They are the GTP interface, and FuegitoGUI graphical user interface (coming soon). Please see the separate documentation for more information about FuegitoGUI (to be provided). To use the GTP interface, the FuegitoMain application must be compiled with the desired extensions.

Building Fuegito from Sourceforge
Using Subversion[2], type the following into the terminal to checkout the latest version of the Fuegito source code from the SourceForge repository:

[2] http://svnbook.red-bean.com/

```
svn checkout svn://svn.code.sf.net/p/fuegito/code/trunk fuegito-code
```

Navigate to the *fuegito-code* directory in the terminal and type *make* to build the FuegitoMain application with all the provided extensions. Ensure the makefile completes the build without errors. There are no other steps to building the application.

To compile FuegitoMain with extensions other than those provided, the makefile needs to be edited manually.

GTP

Fuegito supports the GTP protocol. GTP stands for Go Text Protocol and sets a standard for communicating with game programs using text-based commands. The list of supported commands is provided here, as well as how to configure Fuegito with command-line arguments.

Starting Fuegito from the Terminal

Fuegito can be started from the terminal with the following command:

fuegito –g <game_name> (where game_name is the name of the game to play)

Optional command-line arguments that can be passed:

-w <white_player_name> <optional_white_search_method>

-b <black_player_name> <optional_black_search_method>

For example, to start Fuegito playing clobber with average player as white and search player as black using alpha beta search:

fuegito –g clobber –w average –b search alphabeta

Once Fuegito is running, it is ready to accept commands.

Standard Commands

The following are standard GTP commands that Fuegito supports by default.

- name
- boardsize
- clear_board
- play
- genmove
- reg_genmove
- undo
- showboard
- main_time
- time_left
- list_commands
- version

For an explanation of these commands please see the official GTP specification[2].

Fuegito Custom GTP Commands

Fuegito also implements some non-standard commands for custom functionality.

- list_games; list_players; list_searches
  - Returns the list of currently registered games, players, or search factories.
- setgame <game_name>
  - Sets the game to that given by *game_name*.
- setplayer <color> <player_name>
  - Sets the player of the specified *color* to that given by *player_name*.
- setplayer <color> <player_name> <search_factory>
  - If *player_name* accepts a search factory then additionally sets the search factory of the specified *color* to that given by *search_factory*.
- setsearch <color> <search_factory>
  - If the current player of the specified *color* is a search player, sets the search factory of the specified *color* to that given by *search_factory.*
- settings
  - Lists currently loaded game, players, and search factories.
- redo
  - Replay a move that has been taken back.
- getwinner
  - Returns the winner of the game as "black", "white", or "none".
- endofgame
  - Returns true or false depending on if the game is over or not
- gettoplay
  - Returns the current side to play as "black" or "white".
- getwhite
  - Returns the name of the white player or "none" if no white player set.
- getblack
  - Returns the name of the black player or "none" if no black player set.

Adding Extra GTP Commands

#include "FuegitoEngine.h"

Custom functionality implemented by Fuegito extensions may require additional GTP commands not already supported. Derived classes of SgGame, SgPlayer, and SearchFactory can register their own GTP commands by first providing command handling methods and then overriding the RegisterCommands() method in their base classes.

Command Handlers

Command handlers are simply member functions of any class that take a GtpCommand reference and are registered with the GtpEngine. For example, in AlphaBetaSearchFactory, there is a command handler to get and set the search depth:

```
void CmdDepth(GtpCommand& cmd);
```

Inside the command handler lies the logic behind handling the command. The command reference *cmd* can be manipulated: it can be queried for arguments, and responses can be sent back via the << operator. Please see the Fuego documentation[1] and the code for GtpEngine and FuegitoEngine for more information and examples of how to write a command handler.

RegisterCommands() Method

There are two versions of the RegisterCommands() method depending on the base class.

In SgGame

```
void RegisterCommands(FuegitoEngine& e) const
```

FuegitoEngine calls this method automatically on games when they are loaded. Commands should be registered inside this method by using the FuegitoEngine reference to call the RegisterExtraCommand() method for each command. For example, suppose you wanted to register a new command for your chess game class to see if a player was in check:

```
void ChessGame::
RegisterCommands(FuegitoEngine& e) const {
  e.RegisterExtraCommand("check", &ChessGame::CmdCheck, this);
  /* Any additional commands follow. */
}
```

ChessGame registers the command with the FuegitoEngine reference. It passes the name of the command as a string, a reference to the command handler, and a self reference. Any number of commands can be registered here.

In SgPlayer and SearchFactory

```
void RegisterCommands(const string& color, FuegitoEngine& e) const
```

The RegisterCommands() method for SgPlayer and SearchFactory classes comes with the additional *color* parameter to ensure the command is registered for the right color. Thus, a player or search factory such as AlphaBetaSearchFactory would register commands in this fashion:

```
void AlphaBetaSearchFactory::
RegisterCommands(const string& color, FuegitoEngine& e) const {
  e.RegisterExtraCommand(color, "depth", &AlphaBetaSearchFactory::CmdDepth, this);
  /* Additional commands follow. */
}
```

GetSet() Method

In the *GtpCmdUtil* namespace, the *GetSet()* method is a generic implementation of a getter/setter GTP command.

```
template<typename T>
void GetSet(T& param, GtpCommand& cmd)
```

The first argument is a reference to the class parameter that needs to be accessed, and the second is the GtpCommand reference. When the command is given in a GTP session with no arguments, the value of the parameter is returned. When the command is given with an argument, the parameter is set to the value of that argument.

Using the Commands in GTP

Once extra commands are added and registered, the game, player, and/or search factory that registered the commands needs to be loaded into the current GTP session for the commands to be available. They will then appear when the user types *list_commands*. If another extension is loaded in the previous one's place, then the currently registered commands will be updated to reflect this automatically.

References

[1] Müller, Martin. *Fuego.* http://fuego.sourceforge.net/ (accessed August 20, 2012).

[2] Farnebäck, Gunnar. *GTP - Go Text Protocol.* http://www.lysator.liu.se/~gunnar/gtp/ (accessed August 20, 2012).

[3] Wolfe, David. *Clobber.* http://homepages.gac.edu/~wolfe/games/clobber/ (accessed August 8, 2012).

[4] Russell, Stuart J, and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 3rd. Upper Saddle River, New Jersey: Pearson Education, Inc., 2010.

[5] Hsu, Feng-hsiung. "IBM's Deep Blue Chess Grandmaster Chips." *IEEE Micro*, March-April 1999: 70-81.

[6] Helmbold, David P., and Aleatha Parker-Wood. "All-Moves-As-First Heuristics in Monte-Carlo Go." Edited by de la Fuente Arabnia and Olivas. *Proceedings of the 2009 International Conference on Artificial Intelligence.* Los Vegas, 2009. 605-610.

[7] Zobrist, Albert L. "A New Hashing Method with Application for Game Playing." Technical Report #88, Computer Sciences Department, The University of Wisconsin, Madison, 1970.

Planned Additions to the Manual

Sample GTP Session

Contributing to Fuegito

From Fuegito to Fuego

- *Correspondence Between Main Classes in Both Programs*
- *Main Differences*
    - **Game Class in Fuegito**
    - **Extra Functionality in Fuego**
        - MCTS
        - Multithreading
        - Implements Go
        - DF-PN Solver

List of Constants

Further Reading