Castro Documentation

Timo Ewalds

March 2, 2012

1 Overview

Castro is a Havannah player written in C++. It is composed of several parts. It includes an MCTS player and several solvers. It parses GTP commands, and passes these to the players and solvers as necessary. The player and each of the solvers includes its own copy of the root board, and its own tree or transposition table to represent the possible futures. Some are single threaded and some multi-threaded. The multi-threaded ones have their own thread pools. The GTP controller has an instance of each of the players and solvers, and dispatches commands as needed.

The main() function lives in castro.cpp, and mainly does command line parsing, and sets up HavannahGTP.

HavannahGTP (havannahgtp.h) is subclassed from the GTPClient library (gtp.h), and includes all the non-trivial commands. It sets up callbacks to the commands, which are split into 3 files: gtpgeneral.cpp, gtpplayer.cpp and gtpsolver.cpp. It also includes an instance of the game history (Game class, defined in game.h) and an instance of each of the players and solvers.

The Board class (board.h) maintains a state of the game, and includes an implementation of the rules. The Board class is copyable, and supports making moves, but cannot undo moves without breaking win detection. All moves are stored and made using the Move class (move.h), and are defined in terms of x,y coordinates (stored y,x). The state of the board is represented as a one dimensional vector of Board::Cell, which each represent a cell on the board. The x,y coordinates must be translated into this single dimension. The cells include a union find structure that represent the groups on the board. Each cell can be a group leader, and includes bits for which edges

and corners it is connected to, as well as the group size. Each cell also includes some information about its local neighbourhood.

The MCTS player is defined in the Player class (player.h). It is split into the Player class and the PlayerThread class. The Player class does the high level things, like storing the parameters, controlling the threads, and final move selection, most of which are defined in player.cpp. The PlayerThread class implements the actual MCTS algorithm and mainly lives in playeruct.cpp. It implements descending the tree, expanding children, playing a rollout and backing up the values. The tree is a n-ary tree of Player::Node elements, which are stored in a CompactTree. Each node includes some experience and rave experience, both stored in Player::Exp elements and taken from their parent's perspective. They also include the proven value and best move, and a knowledge value.

The player uses a compacting tree, which is defined in compacttree.h. It is thread-safe, and gives strong bounds on the memory limits. It allocates large chunks of memory from the operating system, and then dishes it out as needed. After garbage collection, the empty areas can be compacted, leaving a contiguous chunk of used memory, followed by a contiguous chunk of empty memory. This is important for fast allocation, and memory efficiency, as well as to avoid fragmentation.

All of the solvers use the same interface: Solver (solver.h), but use different algorithms. The Solver class also includes some basic 2-ply solving routines. SolverAB implements an alpha-beta search with iterative deepening and an optional transposition table. SolverPNS is a single threaded implementation of proof number search, that uses the compacting tree. SolverPNS2 is a multi-threaded PNS, also using the compacting tree. SolverPNSTT is single threaded, but uses a transposition table with a simple replacement strategy. All of the proof number search solvers implement the depth-first feature, and the epsilon trick, and a 2-ply lookahead.

In general, players are represented as 1 and 2. In the context of the outcome, 0 is a draw, -3 is unknown, and -1 and -2 mean that one or the other players can no longer win. In the context of cells on the board, 0 means the cell is empty. All values in the trees of the players and solvers are stored in negamax formulation, meaning they show the value from the perspective of their parent. Outcomes are represented as an absolute outcome, so win for player 1 or 2, not as a win or loss for the root player or for a fixed player.

2 Castro Usage

Castro can be compiled with `make` using `g++`. It has no dependencies other than pthread, and at a minimum runs on linux and OSX.

When started, it will drop into a GTP prompt. It supports many of the standard commands, like `list_commands`, `version`, `showboard`, `boardsize`, `clear_board`, `play` and `genmove`. The `help` command is similar to `list_commands`, but gives a description of what each command does. Many of the commands output a description of their usage when parameters are expected but no parameters are given.

A few shorter commands are available, such as `print` which is an alias for `show_board`, and `clear` which is an alias for `clear_board`. `history` returns a list of the moves made so far, while `playgame` plays a list of moves and plays them in succession. A few havannah specific commands are available, such as `havannah_winner` which returns the outcome of the game once it has ended.

Two coordinate systems are supported, the default is the one used by HavannahGui, but the little golem coordinate system can be used with the command `gridcoords`. It's possible some coordinates aren't output in grid-coords though, as this is a poorly tested feature, but if these are found, please fix or report it.

It accepts a few command line parameters. When castro detects that it is connected to a tty, the board will output in colour, but this detection isn't always correct, so colour can be disabled with `-n` (useful when connecting to HavannahGui). During testing it is often handy to pass a single gtp command, such as to set a parameter, which can be done with `-c`. Sending it a file with gtp commands can be done with `-f`.

3 Move

Moves are stored in the Move class, or one of its descendants. They are stored in x,y coordinates in the same coordinate system as HavannahGui. Moves can be added or substracted, and compared. A few move values are special, namely M_SWAP, M_RESIGN, M_NONE and M_UNKNOWN. M_UNKNOWN is the default value, and is widely used.

4 Board

The Board class (board.h) represents a state of the game, and implements all the operations that can be done to it. It includes some basic values, such as the board size (and some pre-computed values like diameter of the board, number of cells, vector size, etc), the final outcome of the board, and a vector of cells. The cells are represented with the Board::Cell class, which includes the colour of the piece, the union find data structure, the group size, the edges and corners it is connected to, and a few other things.

Moves are made using Board::move(). If you only want to know whether making a move would win, that can be tested with Board::test_win(). Other properties of making a move, such as how many edges/corners it would be connected to, and the size of the resulting group, can be tested with Board::test_cell().

Win detection is done independently for bridges/forks and rings. Bridges and forks are detected by checking the number of edge/corner bits set for the group of the most recently placed stones. Ring detection is implemented in two ways, which are described more detail in Timo Ewalds' masters thesis. The depth-first search method is implemented in Board::checkring_df, and is used when making a move. It has the advantage that it can be skipped most of the time, and can compute properties of the ring, such as the size of the ring, and the number of permanent stones (stones placed before the rollout). The O(1) ring detection is used in Board::test_win() because it is quicker, and can often be skipped when the board isn't being modified.

Board::MoveIterator is an iterator that returns a list of available moves. It works fairly similarly to a standard c++ iterator, and is created with Board::moveit().

Finding neighbours of a move can be done either by adding a value in the `neighbours` array, or by calling Board::nb_begin(), which returns a pointer to a list of neighbours for a position.

Positions created by adding neighbour or asking for neighbours can be off the board, or already taking by a player. Whether it is on the board can be checked with Board::onboard(). The value of a cell can be asked for by Board::get().

Since moves can't be undone, the outcome of a move can be asked for without modifying the board state by calling Board::test_cell().

5 GTP

6 Compacting tree

The CompactTree class (compacttree.h) is a tree data structure that is thread safe and that manages its own memory. It gives hard upper bounds on the allocated memory and avoids fragmentation. `CompactTree` is the overarching memory pool, though you can allocate several independent ones if needed. Castro currently allocates three of them: one for the player, one for SolverPNS and one for SolverPNS2. Being a tree data structure, it is templated on a Node class that must have an element of type `CompactTree<Node>::Children` named `children`, where `Node` is your specific type of Node. The root of the tree is defined as just a simple Node element.

Nodes must have an element of type `CompactTree<Node>::Children`, which let you know how many children it has, allocate/deallocate the children, and index into them as an array or using an iterator. To allocate or deallocate children of a node, you must pass in a reference to your `CompactTree` object so it knows where to allocate the memory. In a single threaded environment you can simply allocate the children and fill them in. In a multithreaded environment you can lock a node. If it fails, some other thread is creating those children, so you should fall back and do something else. If it succeeds, then create a temporary node, allocate the children there, fill them in, and then swap the children into the tree. This avoids the wasted time and memory and potential memory leaks from having multiple threads create the same children.

The CompactTree gives you a way to ask for the memory usage, then a way to compact it. It is your responsibility to deallocate nodes that are no longer needed. The compact function is single threaded, so make sure to stop all the threads before calling it.

7 MCTS

The MCTS player is defined in the `Player` class in player.h. Several classes are defined under Player.

`Player::ExpPair` holds a wins/sims or num/sum pair. The ratio between the two is a winning rate. It allows atomic increments and computes the ratio each time.

`Player::Node` holds a node in the MCTS tree. It stores the real and rave experience (both instances of ExpPair), along with a knowledge value, the proven outcome, the move made from the parent's position, and the best move to make from this position, and of course the children (an instance of CompactTree¡Node¿::Children). Player::Node::value() returns the MCTS value of the node, so it calculates the experience+rave+knowledge value that MCTS maximizes over.

`Player::MoveList` holds the moves made during an iteration of MCTS, including all the different rollout paths. It is used to decide how much experience and rave experience to give to each node in the back-propagation phase.

`Player::PlayerThread` and its descendant Player::PlayerUCT include the actual MCTS algorithm. PlayerThread is intended to be the thread pool, with the MCTS/UCT specific bits being in PlayerUCT. If other algorithms would use the same Tree/Node structure, they'd inherit from PlayerThread, but this hasn't been done yet. The actually code for this bit is implemented in playeruct.cpp.

The Player object itself mainly keeps the player config variables, stores the tree root, manages the thread pool, and does the high level functions like making a move and generating a move. These bits are implemented in player.cpp.

The thread pool is a state machine whose states are defined in Player::ThreadState:

```
enum ThreadState {
    Thread_Cancelled,  //threads should exit
    Thread_Wait_Start, //threads are waiting to start
    Thread_Wait_Start_Cancelled, //once done waiting, cancel
    Thread_Running,    //threads are running
    Thread_GC,         //one thread is gc, the rest wait to run
    Thread_GC_End,     //once done gc, go to wait_end
    Thread_Wait_End,   //threads are waiting to end
};
```

The progress of the states is governed by the state transitions and a couple thread barriers. The actual state machine is mainly in player.cpp in void Player::PlayerThread::run().

..... thread state diagram ....., description

Garbage collection just walks the tree and removes any node with less experience than some threshold, and any solved nodes with a higher fixed threshold. The lower threshold is raised and lowered to try to garbage collect

about half of the tree each iteration.

8 PNS