Documentation for Tzaar robot

Pavel Veselý

August 9, 2012

Our robot consists of the library tzaarlib for searching for best moves (described in Section 0.3), the program tzaarmain that loads the board from a file and starts the search (Section 0.2) and the Python web client for communication with Boiteajeux.net (Section 0.1). For simple building tzaarlib and tzaarmain, a Makefile is provided.

Official Tzaar rules can be found on gipf.com.[1] They were implemented with the exception of the tournament version in which the game starts with a placement phase, because the tournament version is not implemented on Boiteajeux and Boardspace too.

This document requires at least a little knowledge of Tzaar rules. We will use this a word "move" for one part of player's turn (one move with a stone). So each turn of player consists of two moves: the first move has to be always a capture, the second can be another capture, a stack move or a pass (no move).

0.1 Python Web Client for Boiteajeux.net

Web client for communicating with the website Boiteajeux.net is written in Python. It downloads a page from BAJ and looks for a game in which the robot is on turn. When such a game is found the client downloads the page with board and calls the function `PlayGame`.

Function `PlayGame` has a parameter page containing HTML code of the page with board. From the HTML code it parses a position, converts it to the board representation (described in Section 0.2.1) and saves it to a file. Then it calls the program tzaarmain and waits until the computation is done. Finally it loads best moves from a file and executes them on BAJ (via HTTP POST requests).

The client also looks for an invitation for playing – one can start a game with the robot by creating a new game and adding the robot's username in the field Guests. The client sends email to a given recipient when an exception occurs, and every day after midnight it sends statistics of searches done during the day.

Note that before using the client for BAJ, we have to configure it. That means, inside the Python code, fill the robot's username and password, an email where to send error messages and daily outputs, and directories where is the Tzaar robot located and where to save positions and outputs.

0.2 Program tzaarmain

The program tzaarmain written in C is quite simple. Given a file with a position in Tzaar it initializes arrays and variables with a board representation that are in the library. Then it calls function `GetBestMove` in tzaarlib for searching for best moves and saves returned moves to a file.

The files and other options are given via command line arguments:

- `-a N` or `--ai N` – set the algorithm number `N` (see tzaarlib.h). The default AI number is in constant `MAINAI` in tzaarlib.h (it is AI used by the expert robot).

[1] http://www.gipf.com/tzaar/index.html

- `-b FILE` or `--bestmove=FILE` – search for best moves in a position stored in `FILE` and then save best moves into this file. An input file format is described in Section 0.2.1 and output file format in Section 0.2.2. This is a required option.
- `-e FILE` or `--execute=FILE` – execute best moves after searching for them and then save the position to `FILE`. Default is not to save any position after the search.
- `-t SECONDS` or `--timelimit=SECONDS` – set the time limit of the search to `SECONDS`. The default value is in the constant `AITIMELIMIT` (30 seconds).
- `-h` or `--help` – print usage.

0.2.1 Board Representation and Input Files

Now we describe a board representation and then the format of input and output files.

The Tzaar board is a hexagon with 5 fields on each side, but we cannot simply store it in the memory as a hexagon. Thus it is sloped to be fit in the quadratic array 9×9. In the library, one linear array of size 81 is used instead of a quadratic array.

Example of the array containing fixed starting position (array members are separated by spaces):

```
-1   1   1   1   1  100 100 100 100
-1  -2   2   2   2  -1  100 100 100
-1  -2  -3   3   3  -2  -1  100 100
-1  -2  -3  -1   1  -3  -2  -1  100
 1   2   3   1  100 -1  -3  -2  -1
100  1   2   3  -1   1   3   2   1
100 100  1   2  -3  -3   3   2   1
100 100 100  1  -2  -2  -2   2   1
100 100 100 100 -1  -1  -1  -1   1
```

The number 100 stands for a field outside the board and also for the field in the middle of the board. Empty fields with no stone have number 0. Other numbers stand for different types of stones:

- 1 is a white Tott,
- 2 is a white Tzarra,
- 3 is a white Tzaar,
- -1 is a black Tott,
- -2 is a black Tzarra,
- -3 is a black Tzaar.

In this array, a player can move in six directions: horizontally left, or right, vertically up, or down, and diagonally right and down, or left and up. The other diagonal directions (right and up, left and down) are not possible.

The heights of stacks are in another array of the same size. Fields outside the board and empty fields have stack height zero. Example for the fixed starting position:

```
1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1 0
1 1 1 1 0 1 1 1 1
0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

In the tzaarlib, there is also an array with names of fields (`"-"` is a field outside the board):

```
"A1", "B1", "C1", "D1", "E1", "-" , "-" , "-" , "-" ,
"A2", "B2", "C2", "D2", "E2", "F1", "-" , "-" , "-" ,
"A3", "B3", "C3", "D3", "E3", "F2", "G1", "-" , "-" ,
"A4", "B4", "C4", "D4", "E4", "F3", "G2", "H1", "-" ,
"A5", "B5", "C5", "D5", "-" , "F4", "G3", "H2", "I1",
"-" , "B6", "C6", "D6", "E5", "F5", "G4", "H3", "I2",
"-" , "-" , "C7", "D7", "E6", "F6", "G5", "H4", "I3",
"-" , "-" , "-" , "D8", "E7", "F7", "G6", "H5", "I4",
"-" , "-" , "-" , "-" , "E8", "F8", "G7", "H6", "I5"
```

The input file for the program is a sequence of numbers separated by some whitespace characters:

- first a player on turn is specified, i.e., robot's color. Number 1 stands for white, -1 for black,
- then there is the array with the board representation – 81 integers with types of stones, or 0 as an empty field, or 100 as a field outside the board,
- the last is the array with stack heights (81 nonnegative integers).

Example with comments after '%' (they should be deleted before using the file):

```
1 % our robot is white player

% stones on the board
 1  -1  -1  -1  -1  100 100 100 100
 1   2  -2  -2   2   3  100 100 100
 1   0  -3   0  -3   0   1  100 100
 0   0   0   0   0   0   2   1  100
```

```
 0  -3   0   0  100  0   3   0   1
100 -1   0  -3   0  -1  -3   2  -1
100 100 -2   0   0   1  -3  -2  -1
100 100 100  0   2   2   2  -2  -1
100 100 100 100  1   1   1   1  -1

% stack heights
1 1 1 1 1 0 0 0 0
1 1 1 1 1 3 0 0 0
1 0 1 0 1 0 1 0 0
0 0 0 0 0 0 1 1 0
0 3 0 0 0 0 1 0 1
0 1 0 1 0 1 1 1 1
0 0 3 0 0 2 1 1 1
0 0 0 0 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

Note that in the format it does not matter on whitespace characters (spaces, new lines and tabs). The file with numbers separated by a single space and no new lines will is loaded successfully.

0.2.2 Output File with Best Moves

Now we describe how best moves are saved in the output file. On the first line, there is the first move of a turn. It is saved as the name of the field from which a player moves a stack, and the name of the field where the player captured an opponent's stack (the names are separated by a space).

On the second line, there is an integer specifying what is the second move. Number -2 stands for no move (in the case of the first turn of white player, or win after the first move), -1 stands for a pass move, 0 for a stacking move, and 1 for a capture. In the last two cases, names of two fields follow, the first is the field from which a stack is moved, and the second is the field to which the stack is moved.

On the third line, there is some information about the search, namely the search duration in seconds (with three decimal places) and the returned value.

Example of a capture move and a stacking move:

```
G4 C6
0 F2 F1
33.303 33319
```

Example of a capture and a pass move (the returned value 2 000 000 000 means that the robot is going to win):

```
H2 D1
-1
2.742 2000000000
```

0.2.3 Module main

The module main contains function `main` which parses command line arguments using the getopt library and calls `ProcessPosition` which loads position. Then it calls `GetBestMove` in tzaarlib and finally it saves the result.

0.2.4 Module tzaarSaveLoad

The module tzaarSaveLoad contains functions for loading and saving positions in the format described in Section 0.2.1 and a function for saving best moves in the format described in Section 0.2.2. Function `SaveWholePosition` save all information about the position, including counts of stones according to a type, the zone of control, but this function is not used by the robot.

0.3 Library tzaarlib

The library tzaarlib contains the computation part of the program and it is also written in C (standard C99). For searching for best moves the algorithms Alpha-beta and Depth-first Proof-number Search (DFPNS) are implemented. There are also auxiliary functions for generating, executing and reverting moves. It is possible to choose between different versions of the algorithms, for example there are versions for beginners and intermediate players.

0.3.1 Arrays and Fields for Position Properties

For the current position representation there are arrays `board` and `stackHeights` of size 81 with the same format as described in Section 0.2.1. The variable `player` is 1 when white is on turn and -1 when black is on turn. Since the turn of a player consists of two moves, the variable `moveNumber` determines which phase of a turn is: 1 is the first phase (a capture move) and 2 is the second (capture, stack, or pass).

In the variable `turnNumber`, there is the number of turns from the root position of the search (starting 1) – this is different from the number of turns in the whole game which is not known to the robot. Moves are stored in the array `history`.

There are some variables and arrays for storing information about the current position that can be counted directly from arrays `board` and `stackHeights`, but that would be very slow. The array `counts` contains the number of stacks alive (visible) for each stone type, the variable `stoneSum` is the sum of stacks on the board and it is used by the Iterative Deepening.

The variable `value` is the value of the current position determined by the evaluation function or by a search. The variable `materialValue` is counted by the part of the evaluation function that is counted incrementally when executing or reverting moves. The variable `hash` contains the value of the Zobrist hash function for the current position and it is also counted incrementally.

The array `highestStack` of the size equal to the number of stone types contains the height of the highest stack for each type. For incrementally maintaining this array library uses the quadratic array `countsByHeight` that contains the number of stones for each type and height.

The array `zoneOfControl` contains for each stone type how many stones of that type can be captured with one move. This is used in the evaluation function and for determining whether a player has any possible captures when he is on turn and it is his first move. The array is updated also incrementally using the array `threatenByCounts` of the size 81 which contains for each field how many stacks can capture a stack on this field.

0.3.2 Module tzaarlib

The module tzaarlib is the main module of the library. It contains definitions of structures, types, macros, global arrays and variables used throughout the library (some of them are described in the previous section).

The function `GetBestMove` starts the search according to the chosen algorithm and does the time estimation via the Iterative Deepening for the Alpha-beta based algorithms and for DFPNS via the estimation of the maximal number of nodes that can be searched.

In the header file, types and basic macros are defined first. Constants for properties of the game, types of AI (algorithms), and AI settings follows. The structure for a move, global variables, and arrays are defined at the end.

0.3.3 Module tzaarmoves

The module tzaarmoves contains functions for generating, executing and reverting moves. There are also helping functions for deallocating memory (free a single move or a linked list of moves), determining whether someone won in the current position, updating the zone of control after executing or reverting a move and converting between a field index and a field name (for example the field on index 3 has name D1).

Most of functions in this module are optimized to be as fast as possible, because they are called many times during the search. Note that the functions for generating moves are used for the first and the second move of a turn separately.

In the header file there are constants for the Move Ordering and arrays for possible directions that are used in the functions for generating moves.

0.3.4 Module tzaarinit

The module tzaarinit has functions for initializing arrays and variables with information about the current position. The counting of the hash value, the material value (the part of the value of a position that is counted incrementally during the search) and the zone of control is implemented here. Function `InitBoard` prepares starting position according to the parameter `setup` (random, or fixed) and calls other initialization functions, but it is not used by our robot.

Functions in this module are not optimized to be fast, because they are not called during the search, only before it. The values of the parameter `setup` and the fixed starting position are defined in the header file.

0.3.5 Module alphaBeta

The module alphaBeta contains functions that implement the Alpha-beta pruning algorithm with its enhancements, functions for working with the Transposition Table (TT) and static evaluation functions.

There are different Alpha-beta functions with different enhancements used:

- `AlphaBeta` – simple Alpha-beta with storing positions to TT,
- `AlphaBetaPV` – Alpha-beta with TT and the Principal Variation Move (PV),
- `AlphaBetaPVMO` – Alpha-beta with TT, PV and the heuristic Move Ordering (MO),
- `AlphaBetaMO` – Alpha-beta only with the Move Ordering (without TT),
- `AlphaBetaPVMORandom` – random Alpha-beta with TT, PV and MO. It should be called only on the root of the search tree.
- `AlphaBetaPVMONegascout` – Alpha-beta with TT, PV, MO and Negascout,
- `AlphaBetaPVMOHistory` – Alpha-beta with TT, PV, MO and the History Heuristic,
- `AlphaBetaPVMOHistoryNegascout` – Alpha-beta with TT, PV, MO, the History Heuristic and the Negascout,
- `AlphaBetaPVMOBeginner` – Alpha-beta with TT, PV, MO and the beginner static evaluation function,
- `AlphaBetaPVMORandomBeginner` – random Alpha-beta with TT, PV, MO and the beginner static evaluation function. It should be called only on the root of the search tree.

The Alpha-beta enhancement Iterative Deepening is implemented in the module tzaarlib. The Transposition Table use the replacement scheme Two Big.

0.3.6 Module pns

The module pns contains the implementation of the Depth-first Proof-number Search (DFPNS) with some enhancements and the Transposition Table (TT) used by DFPNS (it differ from TT used by Alpha-beta).

There are different DFPNS functions with different enhancements used:

- `dfpns` – simple DFPNS without enhancements,
- `dfpnsEpsTrick` – DFPNS with the $1 + \epsilon$ Trick,
- `weakpns` – Heuristic Weak DFPNS (one can modify it to Weak DFPNS easily),
- `dfpnsEvalBased` – Evaluation Function Based DFPNS,
- `dfpnsWeakEpsEval` – Heuristic Weak DFPNS with the $1 + \epsilon$ Trick and the Evaluation Function Based enhancement,

- `dfpnsDynWideningEpsEval` – DFPNS with the Dynamic Widening, the $1+\epsilon$ Trick and Evaluation Function Based enhancement,

Constants for the enhancements are in the header file. The Transposition Table for DFPNS also use the scheme Two Big.

0.3.7 File hashedpositions.h

This header file contains data for the Zobrist hashing. There is a three dimensional array `HashedPositions` of unsigned 64bit integers (type thash) that contains random numbers for each combination of a field, a stone (or an empty field) and a stack height that can occur on the board. Impossible combinations have value zero.

The array is indexed in this way: `HashedPosition[field][stoneType][stackHeight]` where `field` is an index in the array `board` and it is in range from 0 to 80, `stoneType` is the value from the array `board` plus three (the value ranges from -3 to 3) and `stackHeight` is the height of a stack if there is any, or zero otherwise. Note that for fields outside the board, the values are not used.