COMP3130 – Group Project in Computer Science
Warm-up Project – 4×4×4 TicTacToe Agent

Andrew Haigh – u4667010;
Timothy Cosgrove – u4843619;
Joshua Nelson – u4850020

March 24, 2012

1. *Abstract*

The purpose of this project was to implement an intelligent agent to play 3 dimensional 4 by 4 by 4 Tic Tac Toe. We chose to implement this in C, with a pipe interface to python. This allowed us to use python's 3D libraries for visualisation while utilising the speed of a compiled C program.

2. *Solution Overview*

The core of the agent is implemented with a combination of minimax and α-β pruning.

Board states are stored simply as a 3 dimensional array of chars; but this data is passed through the program as a struct including extra information such as a heuristic evaluation of the state, a move number indicating how many positions have been filled, and a short list of the array coordinates that are occupied (for fast iteration over the array). Once the minimax program reaches a fixed cut-off depth we select a state based on this heuristic. For testing purposes all code is compiled with the -g flag. This allows us to use the program `gprof` to analyse running time and total number of function calls (this is used to optimise and evaluate the effectiveness of α-β pruning).

2.1. *Overview of modules*

`visualisation.py`:

- The python module which takes user input (to play against the computer) and displays the game board using VPython
- This is the main program, it creates a sub-process (`worker.c`) which returns board states to display
- Contains several visualisation options such as labels on each square and even red-cyan stereoscopic 3D. See `README` for details.

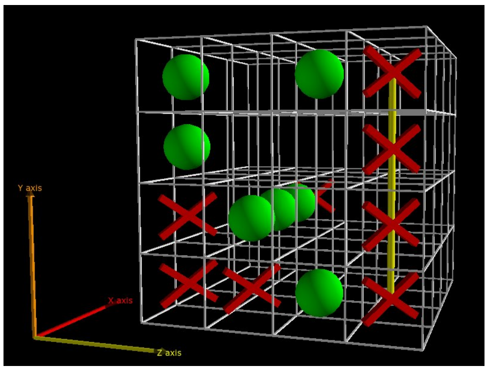

Figure 1: Example of a game visualisation

`worker.c`:

- The link module which handles communication between python and C
- Calls code from `state_functions.c` to start minimax and pick the next move

`state_functions.c`:

- Contains all core functionality of the AI agent including minimax functions and state evaluation functions
- Includes a simple victory function which checks if a player has won with a line nearby to the most recent move. (comprehensive victory checks are made in `visualisation.py`)

3. *State Space for 4×4×4*

In the original scope of this project we had intended to examine the entire state space before making a move. However, the state space is in fact exceedingly large. Considering all possible states we have 3^{64}, but many of these (far more than half) are illegal or unreachable. According to [Pat80], after 18 moves (under perfect play by player 1) the game will be over or in a state where every move is forced. With this in mind we can consider a more reasonable upper bound on the states we must examine:

$$\binom{64}{18} = 3601688791018080$$

If we assume we can examine 1 state per clock cycle (a completely and utterly unreasonable assumption) it would still take over 250 hours to search this reduced state space on a 4GHz CPU. Minimax with α-β pruning and other clever tricks could significantly reduce this figure; however it remains obvious that searching to depth 18 for every move is unachievable. Thus we set a depth cut-off where we stop searching and perform a heuristic state evaluation.

Due to this inevitable cut-off it also became impossible to determine if either player has a winning strategy. It is however interesting to note that (in the paper mentioned above) Patashnik determined that the first player can always win with perfect play.

4. *Heuristics and Cut-offs*

By default we have set the depth cut-off to 6. This value was decided mostly through experimentation; based of how long it takes the agent to make each move. At depth 7 it can take over a minute for the agent to play, making the game a bit unplayable. Furthermore, at a depth of 6 the agent is able to look ahead 3 of it's own moves, allowing it to examine states where it has completed a victory line where it currently has 1 piece. Note that the user is able to set the agent minimax depth (see `README`)

The agent uses two kinds of heuristic evaluation:

A state evaluation function; used to estimate the value of a state at the search cut-off depth

Heuristic ordering; a simpler value estimation to decide which states should be selected first during α-β pruning

The state evaluator gives a state a score between -304 and 304, based off the positions held by each player. Each square on the board has either 4 or 7 victory lines through it; the score of a state is simply the sum of scores held by the computer player (minus the sum of scores held by the human player) [Bog10]. This encourages the agent to take corner and centre squares throughout the start of the game, opening more opportunities to find a victory state (given a score of 305) later.
Heuristic ordering places a simple restriction to first select squares worth 7 when performing α-β pruning, and consider opponent moves where they might score 7 first. A cutoff depth for the sorting procedure is specified, and after this, sorting by heuristic value is no longer performed on the states. This choice was made because the sorting is most effective early in the search; it has the potential to enable α-β pruning to truncate large branches of the game tree. Later sorting has a decreased impact on the α-β pruning. Later sorting also costs more, as there are more nodes to evaluate, so there is a tradeoff involved when deciding the cutoff value for the heuristic ordering.
The agent was profiled during a depth 5 search, with varying depth cutoffs for sorting. The times, and the amount of leaf nodes evaluated, were recorded. The results show that a depth 2 heuristic cutoff is the most effective in terms of time, but the amount of leaf nodes evaluated is consistently reduced as we increase the heuristic cutoff.

Table 1: Heuristic cutoff performace

Heuristic Cutoff	Time (s)	Leaf nodes evaluated
0	1.05	622,961
1	0.94	622,961
2	0.86	618,553
3	1.01	582,091
4	1.25	517,952

5. *Pure minimax vs. α-β pruning*

Some tests were run using the Unix tool `gprof` to determine the effectiveness of heuristics and α-β pruning. By counting the number of calls to our state-evaluation function we get the total number of examined leaf nodes. Furthermore `gprof` returns a summary of how much time (in seconds) is spent running each function and the whole program.

The tables below summarise `gprof` results when the agent is making it's first move; run at various cut-off depths:

Table 2: Total leaf nodes examined

Depth	Minimax	α-β	α-β + Heuristics
3	249,984	8,026	7,964
4	15,249,024	256,936	256,866
5	914,941,440	622,961	618,553
6	5.398×10^{10}	16,771,844	16,770,400
7	3.131×10^{12}	62,306,364	62,003,670

Table 3: Total time taken (seconds)

Depth	Minimax	α-β	α-β + Heuristics
3	0.27	0.01	0.01
4	15.30	0.16	0.14
5	941.99	0.97	0.90
6	2000+	9.49	9.31
7	hours	83.44	79.97

The benefits of α-β pruning are obvious and indisputable, often cutting the number of nodes and total time by a factor of over 1500. However, the benefits of our heuristic function are less obvious. We notice that a great deal less leaf nodes are examined; but the improvement in running time is negligible. This is due to the fact that we must constantly evaluate states for their potential value; taking precious CPU time.

6. *Reflection*

The purpose of this project was to prepare ourselves for building an AI agent for a much more complex game; Othello. As such we used this project to evaluate various technologies and prepare for the challenges we will face in the larger project.
The code hosting service github was invaluable throughout this warm-up project. Now that we have spent the time getting familiar with git we can make better use of github's analysis tools (such as the issue tracker and network graphs) in the upcoming Othello project.
This project was also useful to evaluate our choice of language; the combination of C and Python. While this choice was effective for tic tac toe (due to the need for simple 3D visualisation) we have encountered many drawbacks that could prove overwhelming in the main project:

- No fully featured IDE
- Lack of local libraries (in particular for hash tables)
- No high-level data structures
- Difficult to debug and test

Furthermore the game Othello has no need for 3D visualisation. Because of this we will probably switch to a higher level language (e.g. Java) for the main project.

References

[Bog10] Bradley Bogenschutz. Artificial Intelligence: Implementing 3D Tic-Tac-Toe in C++. *Summation: Mathematics and Computer Science*, 3:5–8, 2010.

[Pat80] Oren Patashnik. Qubic: 4x4x4 Tic-Tac-Toe. *Mathematics Magazine*, 53(4):202–216, 1980.