COMP30024 Artificial Intelligence: Assignment 2

Due Monday 17 May at 11:00pm

Josh Brown, Lachlan Doig

1 Overview

Our final agent was an implementation of **negamax** that used alpha-beta pruning and transposition tables with Zobrist hashing to minimise the number of board evaluations required.

We ultimately combined two heuristics to evaluate a given state, each of which takes a player as input:

- the total number of captures the player, P has made (call this $c(P)$)
- (the sum of) the depths of the deepest 'branch' from each of a player's two goal-edges (call this $d(P)$, explained below)

The final evaluation from player 1's perspective would be

$$eval_{state}(P1) = (c(P1) + d(P1)) - (c(P2) + d(P2)$$

A vital approximation we made was to only consider tiles within a distance of 2 from an already-occupied tile. This reduced the search space enough to allow negamaxing with a depth of at least 3 for all relevant board sizes.

Note that this strategy was applied for larger board-sizes, however a board-size of 3 appeared to always be a win for blue with perfect play i.e. there are no fair openings.

2 Supporting Work

Before creating more complex agents, we began by creating a `TrackingBoard` class as a wrapper to the supplied `Board` class to keep track of useful information about the state of the game. This included which moves were available, which moves had previously been played, the total amount of time the agent had spent thinking and the number of captures made. We created two functions `get_first_move` and `time_to_steal`, which decided what move to begin with and whether or not to 'steal' in response. This simplified the task of tracking and selecting moves later on, as our move-set was reduced to a set of two-tuples, (p, q). A strategy-guide to the standard 'Hex' game suggested that playing in the obtuse-corner $(0, n-1)$ or $(n-1, 0)$ was the fairest opening, so we decided to set `get_first_move() := (0, n-1)` and `time_to_steal() := true`. Our logic for always stealing was that the opener is encouraged to play a fair move, so we have nothing to lose by always stealing, and even if this isn't always true, it would do to begin with.

We then began by creating an agent that could randomly select a valid move. We created a simple testing framework to simulate games between agents and to ensure that the random bot didn't attempt any invalid moves. This would then allow us to compare the skill-level of our more complex agents, and ensure all our agents could defeat the benchmark (random) agent.

The next step was to write an `undo_last_move` function that could perfectly return the board to its previous state, with all internal state information being consistent. This is necessary to perform an adversarial search such as minimax, without resorting to expensive deep copies of board states and move-sets.

3 Heuristics

The first non-trivial agents we created greedily selected a move based on how favourable the resulting board state would be. We trialled several different heuristics for evaluating a given board state. All our heuristics are designed to be non-negative and take a given player as input (e.g. $H_{state}(player)$), such that the value returned conveys how strong a player is currently performing with respect to this heuristic. To calculate a player's advantage, we calculate that player's heuristic value minus their opponent's heuristic value (e.g. $eval_{state}(P1) = H_{state}(P1) - H_{state}(P2)$), meaning that positive values indicate the player is winning and a value of 0 indicates no advantage. This is useful as it means we can sum up any set of $eval$ functions to give a new $eval$ function that is interpreted in the same way.

The first agent simply attempted to capture opponent tiles whenever possible (called `captures` in our code). This agent defeated the benchmark on 20/20 trial games (played on size 8 and 9 boards, alternating red and blue), just by having more pieces on the board. However it wasn't actively trying to build a path between its goal-edges, so we developed a second agent whose goal was to start from a goal-edge and build a path as far away from it as possible. To calculate it's utility, we begin with any of its tiles lying along one edge, then attempt a traversal from those tiles to any adjacent tiles also of its colour. Once this traversal is complete, the value returned is the distance between the edge and the furthest discovered tile. This traversal is completed from both goal-edges:

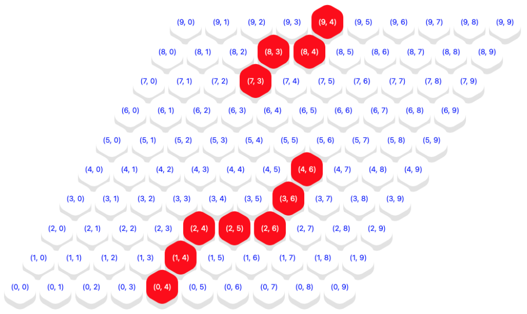

Figure 1: Example of branch with heuristic value 8, since it reaches the 5th row up the board, and 3rd row down the board (red is moving vertically)[1]

A greedy agent using this heuristic (called `edge_branch` was able to beat both the benchmark in 20/20 trials and the greedy `capture` agent on 15/20 trials. We suspected that this heuristic would underestimate states involving a long path that didn't begin at either edge. See figure 2 for an example of this:

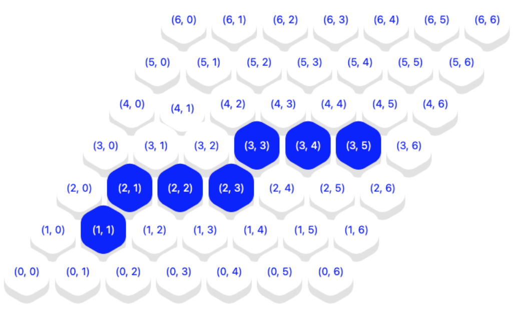

Figure 2: Example of branch with an inaccurate evaluation of 0, since it doesn't touch either of blue's goal-edges[1]

We therefore developed a similar heuristic (called *branch*) that traversed from any of a player's pieces, not just the pieces along one edge. The final score was the maximum length of any 'island' on the board joined by the player's tiles, in the direction that player is going. For example, the shape in Figure 2 would evaluate to 5, as the 'island' spans 5 columns and blue is moving left-right. This heuristic also defeated the benchmark in 20/20 games, and defeated `captures` in 13/20 games. Interestingly, `edge_branch` defeated `branch` in 12/20 games, indicating that building paths from the edge may be a better strategy than building from an arbitrary point. A potential explanation could be that it is much more difficult to capture edge pieces than central pieces. We then combined heuristics to produce `edge_branch_capture` and `branch_capture`, whose heuristics are the sum of their two namesakes' heuristics. These two heuristics performed similarly to each other and to their non-capture-focused versions.

We also implemented several other heuristics which attempted to favour certain regions of the board. We noticed that a sequence of pieces along an edge was less likely to be captured, so we developed a heuristic to encourage placing tiles at the edges. Conversely, we thought that moves on the centre of the board would be advantageous, as controlling the centre allows you to move in any direction. However, given that there are a very large number of combinations of heuristics, we decided not to test these fully before implementing adversarial search, as we suspected that the optimal greedy evaluation could differ from the optimal adversarial search evaluation.

4 Negamax

Cachex is a deterministic, perfect-information game consisting of alternating turns from players, which makes it suitable for minimax. Furthermore, the zero-sum nature of Cachex allows the simplification from minimax to negamax: the players' utility functions are identical and as such each player attempts to maximize the negation of the resulting state value. This is equivalent to minimax (but simpler to implement), as the negation of a player's utility function is the opponents utility function and vice-versa. Thus, our above *eval* structure is suitable since it fulfills this requirement:

$$-eval_{state}(P2) = -(H_{state}(P2) - H_{state}(P1)) = H_{state}(P1) - H_{state}(P2) = eval_{state}(P1)$$

For more, see https://en.wikipedia.org/wiki/Negamax.

The main challenge with negamax was the high branching factor present on a Cachex board. The initial branching factor is n^2 or 81 for a board of size 9. Ignoring the possibility of a 'steal', looking ahead 3 moves will produce approximately $(n^2)(n^2-1)(n^2-2) \approx n^6$ or roughly 500,000 board evaluations. Assuming the board is approximately half-full at the end of the game, this value will fall to $(n^2/2)^3 = n^6/8$ or roughly 64,000 evaluations. We quickly found this to be infeasible, and decided on a simplification: only consider tiles within a distance of 2 from an already occupied tile (Figure 3).

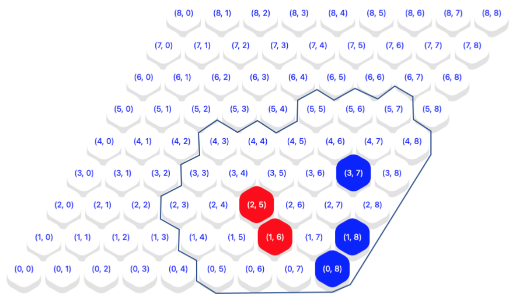

Figure 3: Tiles within a distance of 2 of already occupied tiles (roughly 1/3 of board)[1]

We justified this simplification by noticing that captures are only possible on nearby tiles, and that our above heuristics attempt to build a path of same-colour pieces across the board, which also makes it unlikely that far-away tiles will be useful. Additionally, if an opponent chooses to place a tile outside this boundary, that section of the board will then be included in our next turn, so there is no risk that we will ignore important parts of the board.

These tiles were found each turn by a breadth-first search from occupied tiles, up to depth 2. This was implemented with a standard queue and was extremely cheap relative to the actual negamaxing. This approximation drastically reduced the search space, especially on larger boards, which we would otherwise not be able to assess meaningfully in the allotted. This is because we were allowed n^2 seconds per game, but the complexity of negamax is much greater than this, and is roughly $O(n^{2d})$ for a search up to depth d. However reducing the search space in this way cut down the required time such that games were completed within n^2 even for boards as large as $n = 15$.

We combined this simplification with the standard alpha-beta pruning discussed in lectures. We ordered moves using a priority queue (binary-heap), with the priority of a move being the number of occupied neighbours a given tile has. This followed from the same logic, that captures are only possible with at least 2 neighbours and extending a branch or blocking an opponents branch also requires neighbours, so we further prioritise 'busy' sections of the board. This again reduced the number of evaluations significantly [1].

5 Transposition Tables and Zobrist Hashing

While the above techniques helped to reduce the runtime of our negamax search, they were still insufficient to maintain a decent depth (3 or more) for larger board sizes. One significant inefficiency in the standard negamax implementation is that it cannot recognise duplicate board states reached by different move sequences. In Cachex for example, the sequence of moves "red": (0, 0), "blue": (1, 0), "red": (2, 0) produces an identical board to the sequence "red": (2, 0), "blue": (1, 0), "red": (0, 0). However, in both cases the board will be evaluated as the two sequences are considered to produce distinct leaf nodes. Thus, at a depth of 3 this will lead to roughly double the required evaluations as red's moves can almost always be swapped to produce the same board state. These duplicate boards are called 'transpositions'. A solution to this problem comes through the use of transposition tables. This is a form of memoization that records previously visited board states in a hash-map to further prune evaluations. Each time negamax is called, the current board state is queried in the hash-map to check if the state has been encountered before. If so, we retrieve the value of the previous evaluation to help prune further and save evaluations.

The most significant obstacle to memoization here is efficiently generating short hash-keys from a given board state. Since we perform millions of lookups, we don't want to iterate through each tile on the board or to recalculate the key from scratch every-time a move is made. Zobrist hashing[2] is a highly efficient method that solves both these issues, which is commonly applied in chess, checkers and go, but is equally relevant in cachex. The method involves generating a random bitstring (commonly of length 64) for each piece/tile combination. That is, for a set of piece-types P, and a set of tiles $T = \{0, ..., n-1\} \times \{0, ..., n-1\}$ where n is the board size, we generate a bitstring for each element of $T \times P$, where $\times$ is the Cartesian product. In our case, we have $P = \{red, blue\}$, meaning we must generate $2n^2$ bitstrings. This occurs once at the beginning of the game and gives us a 3D array, $zTable$, indexed by row, column and piece type. Our initial hash is simply $H = 0$, and each time a piece of type $p \in P$ is added or removed from a tile $(r, c) \in T$, this hash is updated to $H' = H \oplus zTable[r][c][p]$, where $\oplus$ represents bitwise 'XOR'.

[1] note that table 1 only shows a slight improvement, but this is due to the small board size of $n = 4$

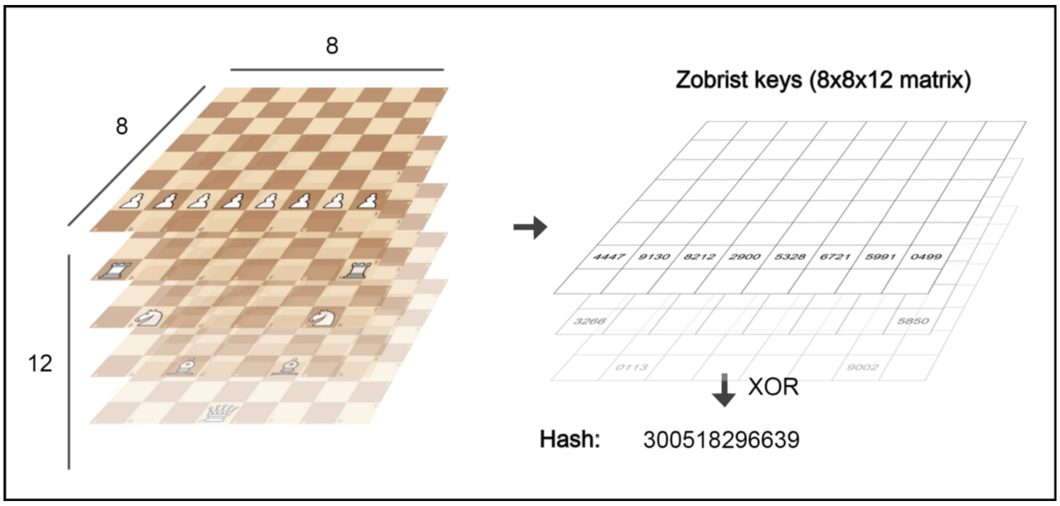

Figure 4: Zobrist hashing in Chess. 12 different pieces on an 8×8 board requires $12 \times 8 \times 8$ bitstrings

This has the vital property that adding and then removing a given piece from a given tile will leave the hash unchanged, since $H \oplus zTable[r][c][p] \oplus zTable[r][c][p] = H$ (this is because $x \oplus x = 0$ for any bitstring x, and $H \oplus 0 = H$). Thus, any sequence of actions that results in the same board state necessarily will produce the same Zobrist hash.

Importantly, the hash never needs to be recalculated from scratch, it is only updated with bitstrings corresponding to the piece that was placed that turn (and in some cases pieces that were captured that turn). This makes it very easy to update the hash as we modify the board during negamax traversal, meaning the hash will always be valid and ready-to-use when we need it. Furthermore, bitwise operations are especially cheap. There is therefore only a small overhead to implementing transposition tables, but a significant decrease in evaluations (see table 1). For more, see https://en.wikipedia.org/wiki/Zobrist_hashing.

It's worth noting that there is a nonzero but very small chance that two boards will map to the **same** hash. This is because there are 2^{64} hashes but there are many more valid board states on larger boards. However, since only a tiny proportion of possible boards are encountered each negamax traversal and the hashes are uniformly distributed, it is unlikely to happen even in a lifetime of negamaxing.

Zobrist hashing has the added benefit of allowing us to check for draws during our negamax traversal. To do this we keep a seperate hash-map that takes a board as a key and returns the number of times a board has occurred, which is kept up-to-date throughout the game. We can therefore modify our evaluation function to return 0 if the board state occurs 7 times, and therefore we can avoid a draw if we are winning (as our utility will be positive) and preference a draw if losing (our utility will be negative so a draw is preferable).

6 Timing

We also took steps to ensure that no agent would ever overstep the timelimit. For the negamax-based agents, we decided that once 75% of the allocated time had elapsed, the agent should reduce its search depth by 1. If that still wasn't enough to see out the game, once the agent had exhausted 97% of its allotted time, it would simply make a greedy move to avoid timing-out [2]. We observed that pruning,

[2] We applied slightly different rules for smaller boards for which we could handle greater depth

Model	evaluations	total time (s)
negamax	43040	12.4
$\alpha - \beta$ **pruning**	8477	3.0
reduced search space	7946	2.7
transposition tables	1902	0.5

Table 1: Resources required to beat the benchmark at size 4 playing as blue with search depth 4. Note: each agent uses all the improvements of previous agents. Reduced search space is more effective with larger sizes at which negamax is unfeasible even with pruning

reducing the search space and transposition tables led to a much stronger agent even with the same evaluation function, since the slower bots would time-out and default to greedy moves. Therefore, cheaper evaluation functions were preferable, and so we used `edge_branch_capture`, the cheapest of the stronger evaluation functions, at initial depth 3. Other heuristics that encouraged play along the shorter axis, along the edges or in the centre did not give any noticeable improvement, and so we discarded them in favour of a cheaper heuristic.

7 Perfect Play for Size 3

Our initial assumption that $(0, n-1)$ was a fair opening didn't appear to hold for n = 3. In fact, no opening move seemed to be 'fair' for size n = 3, in that blue can always win by stealing advantageous openings, namely $(0,2), (0,1), (2,0), (2,1)$, and ignoring the other 4 openings. By employing this rule for the opening moves and setting the negamax depth to 10 (enough to find any possible win-nodes), we created an agent that appears unbeatable as blue. For other board sizes we still decided to always steal and begin on $(0, n-1)$

8 Conclusion

For future refinements, we would consider studying opening moves in-depth to analytically determine the fairest opening. We would also like to trial more complex evaluation functions, however this will come at the expense of time, as well as more sophisticated procedures for allocating time to a move, for example based on who is winning or the average time taken so far. Our final agent stayed comfortably within the time and space constraints and was able to consistently defeat greedy opponents. By playing against this agent ourselves, we felt it had a good 'understanding' of how to play the game and was able to defeat us through consistency and fairly good foresight.

References

[1] Surya Venkatesh *Cachex Visualiser*, https://surya-ven.github.io/cachex/

[2] Albert Lindsey Zobrist, *A New Hashing Method with Application for Game Playing*, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, (1969).