Adversarial Attackers Part B

By: William Lord and Luke Hawkins

I Minimax with alpha-beta pruning

A common approach to applying AI in games is minimax with alpha-beta pruning. This is the approach our submitted bot has followed.

A Minimax

Minimax is the approach where we simulate taking a moving and then our opponent until we reach the terminal state, win/draw/lose, Max's goal is to maximise the score it returns, thus it will use the move that returned the highest value, and Min's goal is to minimise the score it returns, thus it will use the move that returned the lowest score. This effectively searches the whole tree and produces the optimal move. However the ability to search the whole tree is usually impractical as the number of nodes typically grows exponentially. Instead the search is cut off earlier and a static evaluation function is used to try and approximate the 'goodness' of the state at the cutoff.

B Alpha-beta pruning

Alpha-beta pruning is the technique applied to a minimax tree to prune the number of nodes searched by not exploring certain nodes when their result will have no effect on the outcome. This occurs when max has found a move worth α and min has then found move worth β, at this point if $\alpha \geq \beta$ then min will never give max anything higher, thus this move will not ever be selected the search can return early. Similarly for if $\alpha \geq \beta$ subsequent moves don't need to be explored as max will never give anything lower for min. The effectiveness of this can be seen in Table 1, reducing the time from 86.5 seconds to 2.18 seconds.

C Forward Pruning

Forward pruning is where certain actions are immediately removed from consideration without evaluating their minimax score. This greatly reduces the number of nodes searched as it prunes entire subtrees, however for this efficiency gain we did not want to accidentally prune "good" moves. Hence we decided to restrict our forward pruning only to "suicide booms".

In most classical games such as chess, it is not possible for a player to force the opponent to win by committing suicide. Consider chess as an example; it is almost impossible to force the enemy to checkmate you (they can be determined not to by moving pieces back and forth) in chess. Expendibots is different, as a player can execute boom actions which only destroy their own pieces which we call "suicide booms", thereby forcing the enemy to win no matter what the

opponent does (short of the opponent having first move and also executing the same strategy). It is highly unlikely that any suicide boom would yield a future state that is beneficial to us, hence we forward prune it from consideration. If we have *k* positions occupied on the board, this can reduce up to *k* subtrees from evaluation, a simple but large speed improvement.

D Evaluation function

The evaluation function is a mapping from the board state to a real number. The real number denotes the “goodness” of the board for the player to move, the higher the better. Due to the branching factor and depth of the game, it is not practical to run minimax to termination, hence an evaluation function is required to evaluate these intermediate board states. There is a trade-off between information and speed. Arguably the more features we use in our evaluation function, the better we can approximate the “goodness” of the board, however by using more features we increase the processing time. Since the evaluation function is run at each node in the tree (that has not been pruned), it needs to be fast enough to be run thousands of times a move, for possibly hundreds of moves (capped at 250) per game. As a result we chose a simple evaluation function:

$$eval(boardState) = \frac{\#friendly\ pieces + 1}{\#enemy\ pieces + 1}$$

We add 1 to prevent zero division errors. We chose this ratio as it takes into account how many extra pieces we have available to us when a boom is chosen. The evaluation function is not perfect as it does not take into account potentially sacrificing our pieces in order to gain a positional advantage, but more importantly, the function will only return a different number after a boom has been performed (since there is no other way to reduce the number of pieces on the board). However since our evaluation function is so simple, we are able to perform our search to an extra depth of 5; a more complex evaluation function would require us to reduce the depth.

In the middle of the game when the pieces are beginning to develop to the middle of the board, there is likely to be a short sequence of moves to reach a boom, and this is where our bot will perform well. However in the beginning of the game, there may not be any short sequence that results in a boom; although our move-ordering heuristic can help select a good move in this situation, this is a weakness in our algorithm. Aside from the ratio between friendly pieces and enemy pieces, we mention some other features we tried to incorporate as well, but removed for efficiency reasons.

1 Ratio of Available moves

We reasoned that the player that could make more moves given a board state would be in a better position — there is also a correlation between this ratio and ratio between the number of pieces. However adding this as a second feature we took into consideration increased the time it took to complete an average game by over 15x.

2 Ratio of number of connected-components

A connected-component is defined in II.G.2. We reasoned that the more connected-components your pieces were spread in, the more resilient you were to booms destroying your pieces. The cost to identify how many connected-components each player has however was too expensive however.

E Move-ordering heuristics

We found that we could double the depth of the search when ordering the moves prior to alpha-beta pruning. This drastically improves the skill of our bot. Our ordering attempts to predict what is the most optimal action based on the current board state. However since this ordering needs to be done at every node it needs to be efficient. The final heuristic we used prioritises boom actions first, as that affects the core of our evaluation function; as a side benefit booms will also reduce the branching factor in subtrees as there are less tokens on the board. After the booms we ordered the movement moves with the most aggressive moves first.

To come up with this movement heuristic we decided on ordering moves by using an area of effect technique. We would compute the clusters of the enemy tokens, and then set the number of tokens in each cluster to the value of that position in the boom map. The resultant matrix is what we call a boom map; you want to boom where the numbers are high, destroying more enemy tokens. We also noticed this did not change in a player's turn so need only be computed once per turn. An example boom map can be seen in Figure 2.

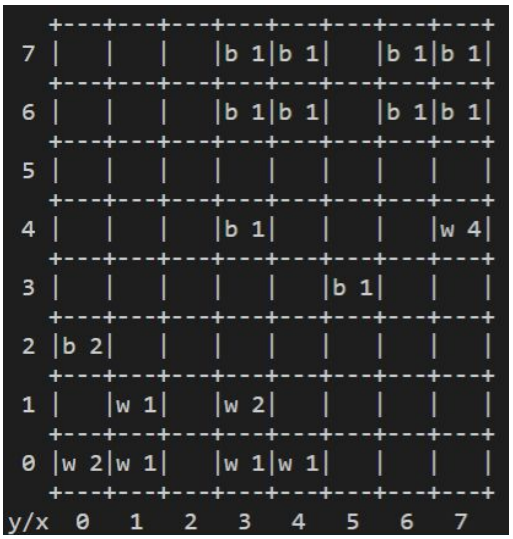

Figure 1: Board State

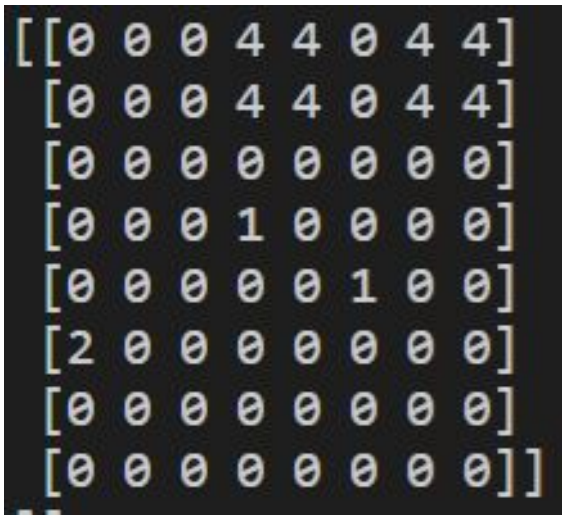

Figure 2: Boom Map for Black

Then for each move action, its movement map is generated. This is created by setting the location where the stack/piece moved to by the number of pieces now at that position Figure 3, then a gaussian filter was used to spread these values out, with a sigma of number of

pieces(the higher the sigma the more blurry), to represent it moving in future turns. This result is shown in Figure 4 of a token at (0,0) with a stack size of 2. So the lower the value, the longer it takes to reach that tile, however the higher the stack size the further these numbers spread out. The gaussian blur is quite expensive, however the movement map only took in a position and number of tokens, thus the possible combination, were relatively small and it could be cached. This sped up the computation by about 4 times.

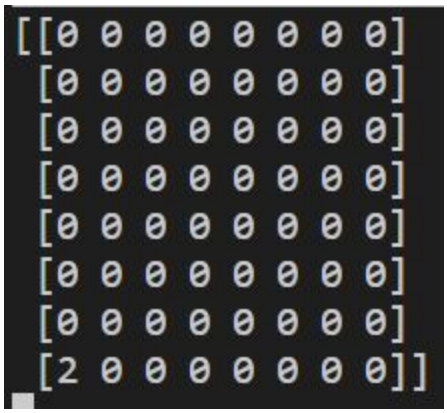

Figure 3: Movement map for (0,0) white

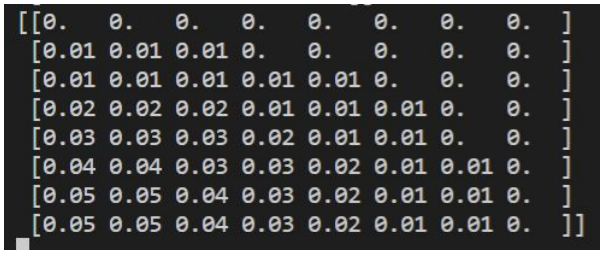

Figure 4: Movement map for (0,0) white

With these two maps we took the grand sum[1] of their hadamard product[2], ordering by the resultant sum in descending order. The idea was to identify moves that move towards enemy stacks as well as stacking our pieces to increase their mobility (and value in the movement map). A downside of this tactic is that it prioritises offense and does not actively consider defensive moves (although arguably stacking pieces also allows us to evade attacks on that stack more easily).

Table 1: Comparison of $\alpha - \beta$ pruning and ordering of moves		
Approach	3 turns	4 turns
Minimax no $\alpha - \beta$ pruning	443016 visited, 86.5 seconds	N/A
Minimax $\alpha - \beta$ pruning, Booms first then moves	41376 visited, 2.18 seconds	74678 visited, 8.17 seconds
Minimax $\alpha - \beta$ pruning ordered Booms then aggressive moves	5684 visited, 0.37 seconds	16148 visited, 3.78
Improvements with better ordering	86% less nodes visited, 589% speed increase	78% less nodes visited, 216% speed increase

[1] The grand sum of a matrix is the sum of all elements in the matrix
[2] The hadamard product of two matrices is the element-wise multiplication of the matrices.

Finally we benchmarked the ordering to see its improvement. From Table 1 it clearly shows we were successful, speeding up our agent by about 2-5 times and visiting around 80% less nodes.

F Quiescence Search

Ordinarily when our depth has reached the cut-off in minimax we return the value of the evaluation function at that state. If the state is “unstable” or “volatile” however, the evaluation function is more prone to inaccuracy. Quiescence search will extend the minimax search from these from these unstable/volatile nodes, deferring evaluation until a later (preferably more stable) state.

Determining what constitutes an unstable or volatile state parallels is similar to trying to create an evaluation function: it is not easy. We identified 2 sources of instability:

1 A good boom move is available

If the cut-off state contains at least one good boom move available then this is an unstable state as, per our evaluation function, only boom moves can cause a change in the ratio. Hence, we want to defer evaluation for a few moves until we see those good boom moves play out and hopefully arrive at a more stable state; this was included in our implementation as it was a great aid to the evaluation function. In extending the search, instead of continuing to search move actions, we forward pruned so that only boom actions were available (that were next to an opponent’s token) as continuing to search move actions significantly increased the runtime.

2 Existence of pieces with a stack size of greater than *k*

If a piece with a stack size of say 4 is on the board, then there is a high potential for this piece to slingshot one of the pieces in the stack 4 moves in any the cardinal directions, potentially next to an opponent's piece. Hence, we want to defer evaluation for a few moves to see whether this slingshot does occur, and potentially evaluate the subsequent boom that it results in.

Although positions with high stack sizes are reasonable sources of instability, practically, a lot of states at the cut-off triggered quiescence search for this reason. Even when only extending search by 2-3 more moves, a large bulk of our processing time was spent on quiescence search due to high stack sizes, meaning our algorithm could not perform within the time constraints. Accordingly, we did implement the boom quiescence search in our submitted bot, but did not include quiescence search if the instability was due to a high stack size.

II Greedy Approach

In order to test the effectiveness of our bot, we also developed, in parallel, a bot which selects moves greedily. This is better than merely testing our bot against a bot which selects a move at random from the set of all legal moves, as the latter bot is very easy to beat. Our greedy

approach attempts t o select locally optimal actions at each game state for the player (while potentially looking ahead one move). The actions are selected on the basis of heuristic identified on the basis of domain knowledge about the game itself. We have identified 6 classes of heuristics and will discuss their rationale briefly.

G Classes of heuristics moves

1 Sensible booms heuristic

The sensible booms heuristic returns the set of booms for the player where the number of enemy pieces destroyed in a boom move is greater than or equal to the number of allied pieces destroyed.

2 Moves out of enemy connected-components

To use the language of graph theory, if we view each position on the board (occupied by a piece) as a vertex in the graph, and an edge existing between vertices if and only if the chebyshev distance between the vertices is 1, then we will have a graph with k connected-components. For example, in the beginning of the game, $k=6$ another example, if $k=1$, then any boom move will cause the game to draw.

This heuristic returns the set of moves where the piece is initially in a connected-component that contains an enemy piece, and after making the move, the piece is no longer in any connected-component which contains an enemy piece. In essence, this heuristic moves our pieces out of danger.

3 Slingshot moves

This heuristic returns the set of moves where making the move from position x to position y would cause there to exist a sensible boom move (heuristic 1) from position y. In essence, this heuristic moves our pieces to attack the enemy.

4 Increase ally connected-components

An ally-connected component is a connected component where all pieces in the component are our own pieces. This heuristic returns the set of moves that would increase the number of our connected-components. In essence, this heuristic moves our pieces so recursive booms are less damaging to our pieces.

5 Stack pieces

This heuristic returns the set of moves that would increase the stack size of one of the positions our pieces occupy. A pieces with high stack sizes can move greater distances allowing more slingshot moves to exist (heuristic 3) and consequently more sensible booms (heuristic 1)

6 Distance reduction

The final heuristic is a catch-all in the event the other heuristics do not apply. Rather than merely returning a random move in that case, this heuristic will select the move that, in effect, decreases the distance between all our pieces and all of the opponent's pieces. In essence, the heuristic prioritises aggression if all else fails.

H Greedy Approach Conclusion

Our greedy algorithm selects the available actions from the above heuristics prioritising the heuristic actions in order of our presentation here (eg if a sensible boom exists and a slingshot move exists it will always choose the sensible boom). In general the greedy bot will perform decently if it can choose from the first 5 heuristics, however it has weaknesses in the early game against an aggressive enemy since it prioritises increasing the number of its connected-components, but in doing so neglects the enemy stacking pieces and slingshotting them into one of its clusters of 4 that it neglected. Nevertheless, the bot served as a good benchmark against other bots utilising more principled methods to play against. Since it was not deterministic it was useful to run multiple times to evaluate new scenarios quickly allowing a more broader understanding of our bot without changing its opponent.

III Monte Carlo Tree Search (*MCTS*)

As an alternative to the typical minimax with alpha-beta pruning approach we considered a bot based on a vanilla implementation of MCTS. MCTS is composed of 4 stages: selection, expansion, simulation and backpropagation.

MCTS involves storing a game tree where each vertex is a board state, and an edge from parent to child is the application of an action on the parent board state resulting in the child board state. Each vertex also contains statistical information about the simulations which is used to decide which nodes the algorithm will investigate and ultimately which move will be returned.

As MCTS is not an algorithm discussed in class, we will explain both how it operates and the important implementation details we tried at each stage of the algorithm.

I Selection

The selection phase in MCTS involves starting from the root of the game tree (the current board state) and traversing the tree until we reach a leaf node. No simulations have been performed at the leaf node yet.

At any given node, there may be more than one child. Each child (assuming they are not leaves) have statistics regarding the number of simulations run from that child node, as well as the number of wins, losses and draws. A key problem is balancing exploitation and exploration, we may initially think selecting path whose child nodes have the highest win rate is desirable, but this is prioritising exploitation heavily at the expense of exploration: what if other child nodes with lower win rates may actually yield a higher win rate if we just ran more simulations?

We used the UCT (Upper Confidence Bound 1 applied to trees) algorithm to determine which child node should be traversed until we reach a leaf node.

J Expansion

The expansion phase in MCTS is applied when we reach a leaf node. For a given leaf node, suppose there are *k* legal moves available at this node. How many nodes do we add to this leaf before the simulation step? If we only add a subset of the legal moves, which ones do we add?

The default value we set for expansion was 6, so 6 nodes would be expanded at the leaf. Furthermore, the 6 nodes to be expanded are determined based on the evaluation function we described above. A large deficiency with our approach is we are effectively pruning *min(0, k - 6)* nodes and if *k* is large, we could be pruning potentially 80 percent of the actions, and since our evaluation function only changes in values when booms are played, the expansion phase has no good way of determining what constitutes a good move action or not.

If we increased the number of nodes expanded however, we would be storing a larger game tree. For example, If we ran selection *d* times, we may have approximately 6*d* nodes stored in memory. In general if we altered the branching factor of the expansion from 6 to *b*, we have *bd* nodes stored.

K Simulation

The simulation phase simulates a game from these expanded new leaf nodes by selecting random actions from the set of all legal moves to play until termination. This code relating to simulations, specifically which moves to select is important as if we run *s* simulations which have an average depth to termination of say *d*, then we would be selecting an action at each depth *sd* times. Assuming the time taken to select a move on average is *t* this would take *tsd* time, and so the total time taken is heavily sensitive to run all the simulations is heavily sensitive on the time taken to select a move. This is why we opted for selecting random actions.

Unfortunately, due to the nature of suicide booms being present in this game, this requires an extraordinarily large number of selections (and consequently simulations) for the law of large numbers to converge to some meaningful proxy evaluation of which move has the highest probability of winning (removing suicide booms from the random actions takes too much time for reasons mentioned in the preceding paragraph).

L Backpropagation

Once the outcome of the simulation has been determined, the results (win, loss or draw) are back propagated up from the leaf node to its children until we reach the root node.

M Monte Carlo Tree Search Conclusion

Although MCTS as a technique has had extraordinary results when combined with other approaches in cutting-edge AI such as AlphaGo, our very basic implementation performed very poorly even against a primitive greedy algorithm. The number of simulations required is too high to achieve any meaningful action given our evaluation function that cannot distinguish between actions that are merely moves, and alterations to improve how actions are selected during the simulation phase from random to something less than random increase the time spent processing by an unacceptable amount. As a result, we did not explore this approach further.

IV Conclusion

Ultimately we submitted a bot which applied minimax with alpha-beta pruning with ordering of moves. Compared to our other types of bots (Greedy and MCTS), it performed the best categorically. In terms of the modifications to a vanilla minimax alpha-beta pruning bot, we have discussed these modifications in section I.