COMP30024 ARTIFICIAL INTELLIGENCE – EXPENDIBOTS

Project Part B: Playing the Game

Sothea-Roth Bak 910126, Andrew Naughton 910691

1. Describe the approach your game playing program uses for deciding on which actions to take throughout the game. Comment on your choice of search algorithm and any modifications you have made, and why. Explain your evaluation function and its features, including their strategic motivations.

Approach

Our game playing program bases which action it should take according to a variant of the alpha-beta search algorithm taught in class. Alpha-beta search is part of the minimax[1] family of algorithms, which suit two-player zero-sum games[2], like Expendibots.

Note that, throughout the development process, we likened Expendibots to Chess since they share the following properties (this is not an exhaustive list):

1. Turn-based, zero-sum, and adversarial
2. Deterministic and perfect information
3. They play on an 8x8 board
4. They have a branching factor that varies throughout the game
5. They have a high average branching factor[3]
6. They have an implicit goal, as there may be multiple possible goal states

Algorithm and modifications choices

In choosing which algorithm to base our program on, the following two details held much weight:

- Minimax-based algorithms, including Negamax implementations like ours, are complete and optimal when the search space is finite, and the opponent is a utility-maximiser. Regarding the former, Expendibots has rules to ensure the search space is finite – namely the 250th turn and 4-repeated state policies. In terms of the latter, the assumption that opponents will aim to maximise their utility appeared wholly reasonable considering the opponents described in the 'performance' section of the assignment specification.
- A consequence that high average branching factor games like Expendibots and Chess have is that the search for a solution path is intractable due to the immense size of the search space. The challenge therefore becomes finding an algorithm that can search the deepest in the least amount of time. In our research, we learnt that the (almost) top ranked chess engine belongs to the minimax algorithms family and is an enhanced variant of alpha-beta search. Stockfish's[4] success in the chess programming realm is attributed to its ability to search to unmatched search-tree depths thanks to its optimisations and modifications, some of which are distinct to Chess.

Thus, our team implemented an alpha-beta search algorithm with optimisations and modifications befitting an Expendibots context. While some of our optimisations and modifications were pioneered in the Chess programming realm, we have confirmed through experiments that these techniques transcend Chess to make tree-search more efficient. Our program features the following optimisations and modifications, all of which were chosen for their ability to appreciably improve the efficiency of the tree-search:

1. Alpha-beta pruning
2. MTD-f (zero-window search)
3. Transposition table with least recently used (LRU) algorithm
4. Move prioritisation by sorting next moves by evaluation score
5. Iterative deepening with time-based and PV-Nodes game-over cut-off
6. Hardcoded strategy for opening four-moves of game

[1] https://en.wikipedia.org/wiki/Minimax

[2] Zero-sum games describe games where each player's gain or loss of utility is balanced by the losses or gains of the utility of the other player(s)

[3] Chess' average branching factor is said to be 35 to 38 nodes and varies throughout the game. Initially its branching factor is 20. Expendibots, on the other hand, has an initial branching factor of 50, suggesting that its average branching factor is likely to be the same, if not higher.

[4] https://en.wikipedia.org/wiki/Stockfish_(chess)

Evaluation function and features, including strategic motivations

Our evaluation function has the following features (and inputs to features). Note that a cluster is a group of pieces that would be exploded should any of them decide to boom at that moment.

Feature	*Description*
Friendly pieces	How many pieces on the board belong to our team
Enemy pieces	How many pieces on the board belong to the opponent's team
At-risk ratio differential	The friendly pieces in mixed (friendly + enemy) clusters as a percentage of the total number of friendly pieces, less the enemy pieces in mixed clusters as a percentage of the total number of enemy pieces.
Friendly-only clusters	Clusters involving only friendly pieces
Enemy-only clusters	Clusters involving only enemy pieces
Friendlies safe	Total number of friendly pieces in friendly-only clusters
Enemies free	Total number of enemy pieces in enemy-only clusters
Optimistic total distance to swarm each enemy-only cluster	The sum of the minimum Manhattan distance calculated to every enemy-only cluster. Note that we account for the permissible step size of friendly pieces, meaning that this feature is always optimistic/admissible.

The function comprises mid-game (default) [1], near-endgame [2] and endgame [3] strategies as well as a utility function for terminal states [4].

Given a board:

1. The mid-game strategy is the default and is as follows:

   > 10 * *friendly pieces* – 10 * *enemy pieces* – 20 * *at risk ratio differential* + *friendly-only clusters* – *enemies free* – *optimistic total distance to swarm each enemy-only cluster*

   The heuristic places most importance on the number of friendly pieces and enemy pieces, and the at-risk ratio difference. The former is important because it ensures that the good boom opportunities are not passed up, and vice versa. The latter is a forward prediction of the promise or threat of boom zones – for example, it will assume that the pieces in mixed clusters are in imminent danger of being exploded, and tells us who stands to benefit – a positive at-risk ratio differential is harmful for our team, and vice versa. The friendly-only clusters, enemies free and distance metrics are equally weighted. However, the enemies free and distance measures are correlated – so, the friendly-only clusters feature really carries the least weight. The intuition behind this is that our team's pieces should only disperse if we cannot further minimize the distance and enemies free as a last resort. The distance and enemies free measures help to ensure we close on the enemy clusters efficiently.

2. The near-endgame strategy is enacted when we have at least <u>two</u> friendlies safely away from the enemy, and the enemy has exactly <u>one</u> piece safe. It is described as follows:

   > 20 * *friendly pieces* – 20 * *enemy pieces* – 20 * *at risk ratio differential* + 0 * *friendly-only clusters* – *enemies free* – *optimistic total distance to swarm each enemy-only cluster*

   When it is nearly the end of the game and we are down to chasing just one enemy (enemies safe == one) with at least a couple of friendly pieces at our disposal, we tweak a couple of coefficients to ensure we finish the game

quickly. Thus, we further emphasise the value of the number of the friendly and enemy pieces and remove the friendly-only clusters feature. The friendly-only clusters measure encourages the spreading out of our pieces, and this is unwanted behaviour at this stage of the game since we are likely the favourite to win.

3. The endgame strategy is invoked when we have at least one friendly safely away from the enemy, and the enemy has exactly zero pieces safe. It is described as follows:

100 * (12 – *enemy pieces*)

When it is the end of the game with every enemy in our reach, we reduce our evaluation so that the only feature is the number of enemy pieces. This will ensure we finish the game without trying to maximise the number of friendly pieces -- we note that the winner gets no extra points for winning with 2 pieces left compared to 1.

4. The utility function for terminal states is simply:
    - If a win is detected, we return a score of 10,000
    - If a draw is detected, we return a score of 0
    - If a loss is detected, we return -10,000

Thus, the program looks most favourably on wins, then draws, and finally losses, in that order.

2. Comment on the overall effectiveness of your game-playing program. If you have created multiple game-playing programs using different techniques, compare their relative effectiveness, and explain how you chose which program to submit for performance assessment.

Our program has undergone several changes and reworkings to become the best version of itself. On the battleground, our agent has performed well, having won most of the games it has played. Furthermore, it has played against previous iterations of itself and it has safely earnt its spot for selection. The litmus test for whether a change persisted was to play each iteration against its previous iteration and verify that it either wins or, if it draws, is more efficient in arriving at the same conclusions. This testing regimen was how we chose which program to submit.

Our first iteration was for educational purposes to understand the mechanics and idiosyncrasies of an Expendibots game-playing algorithm within the prescribed framework (referee, init, action, update). Thus, we started with a random game-playing agent that chooses its available actions randomly. The iterations after this, in order, were:

1. Negamax algorithm[5] with a static depth cut-off
    - This algorithm performed better than the random agent because of the Negamax (minimax variant) framework and our evaluation function. Its weakness was that the search space was often so vast that we could only evaluate 2-ply game trees within a reasonable amount of time (with some variation depending on the state of the board)[6]. Even so, the static depth cut-off meant that we could only evaluate 2-ply game trees for the entirety of the game. Note that Negamax is simply a programmer's shortened implementation of minimax, contingent on the property max(a,b) = -min(-a,-b) holding true.
2. Alpha-beta pruning with move ordering and static depth cut-off
    - This algorithm was a substantial improvement on the previous iteration as it prunes uninteresting nodes that would never be considered due to the nature of the minimax framework. The move ordering[7] went hand in hand with the alpha-beta pruning, as we sorted the 'next moves' to be considered at each ply by their evaluation score. The intuition behind this is that promising nodes at one ply are likely to lead to more promising nodes at deeper plies. This aids the pruning process by narrowing the alpha beta window earlier, translating to more cut-offs and the ability to search to greater depths in the same time.
3. Alpha-beta pruning with move ordering and a depth cut-off tied to the initial branching factor from the root

[5] https://www.chessprogramming.org/Negamax
[6] It is our intuition that the higher the branching factor of the root board state, the higher the average branching factor for that game-tree, meaning the time complexity b^m will also be greater for search.
[7] https://www.chessprogramming.org/Move_Ordering

- After having some difficulty getting a time management system working, we came up with a work-around solution to manage which depths we could realistically search to. It was based on our intuition that the higher the branching factor of the root state, the higher the average branching factor for that search tree, meaning the search time complexity b^m would also be higher. We therefore tied the depth cut-off to the initial branching factor of the state in question. The depth cut-off for states that have an initial branching factor between 40 and 60 would have a depth cut-off of 3 (3 plies), and those between 20 and 40 would be 4 (4 plies). The starting state of Expendibots has a branching factor of 50, so it would search to a depth of 3 in deciding its action, for instance.
- The performance of this iteration was pleasing and an improvement on previous iterations.

4. Alpha-beta pruning with move ordering and an iterative deepening framework tied to time and game-over states in returned best-move sequence
   - We were finally able to get the iterative deepening framework to work, which facilitates a time management strategy that stipulates how long the program can explore for the best action to take. This is another configurable setting that we have set to 0.5 seconds. How it works is that we iteratively search to increasing depths, but, before doing so, we check that we have not exceeded the allowed time for exploring. We cut off our search when either we have taken more than the allocated time or latest returned best move sequence ends with a terminal state, insinuating that we do not need to look any deeper because the end of the game is already in sight.
   - The iteration also improved our performance as we could search to greater depths than before.
5. Alpha-beta pruning with move ordering and an iterative deepening framework tied to time and game-over states in returned best-move sequence, and caching states with their depth, score, move and bound within a transposition table
   - A transposition table was a great addition as it saved us a lot of time that would ordinarily be spent doing recalculations, such as recalculating the minimax value and best move of a state we had seen previously for a given depth. As a result, we could significantly reduce the number of nodes to expand for a given depth, trackable by our 'nodes' count variable in the program, and search to greater depths in the same time, enabling better informed decisions. Another variable that helped confirm to us that the transposition table was valuable is the 'hits' count variable.
   - We saw improved performance with this iteration as well.
6. MTD-f algorithm to narrow the window of alpha beta to zero, with iterative deepening tied to time and game-over in PV-Nodes and caching states with their depth, score, move and bound within a transposition table
   - The MTD-f algorithm[8] relies on the principle of convergence. By keeping a window size of zero for each call to alpha-beta, we zero in on the bounds of the minimax value until we finally have a lower bound that meets or surpasses the upper bound. While we do make several more calls to alpha-beta, we enjoy many more cut-offs which is how we achieve greater efficiency overall. Before this iteration, our standard implementation of alpha beta search used the widest possible window being negative to positive infinity.
   - We experienced great benefits to our performance and efficiency from adding this change.
7. A fixed opening sequence of four moves to be taken, where possible, and the MTD-f algorithm to narrow the window of alpha beta to zero, with iterative deepening tied to time and game-over in PV-Nodes and caching states with their depth, score, move and bound within a transposition table
   - The first four moves were highly predictable and so we figured we would give the program direct rules for how to start the game. This culminated in a lightning quick start and time-savings for more worthwhile time-consuming decisions in the middle and end stages of the game.
   - An opening strategy such as this was time efficient and, in terms of performance, the program benefitted greatly from its addition.

Throughout development, we attempted to tune the evaluation features' coefficients manually. We did this through observing the actions it would take in games against previous iterations, the same algorithm with different combinations in coefficients, and various opponents on the battleground. We tried to avoid overfitting the data to the sample data by testing against several different opponents. The tuning process proved to be most exhausting

[8] https://people.csail.mit.edu/plaat/mtdf.html

and, in retrospect, would have been an excellent time to try our hand at unsupervised reinforcement learning for automated tuning[9] of evaluation parameters and weights.

3. Include a discussion of any other important creative or technical aspects of your work, such as: algorithmic optimisations, specialised data structures, any other significant efficiency optimisations, alternative or enhanced algorithms beyond those discussed in class, other significant ideas you have incorporated from your independent research, and any supporting work you have completed to assist in the process of developing an *Expendibots*-playing program.

We made the following choices with a view to increasing efficiency. Note that this algorithm is likely to be improved with a smarter heuristic, trying PV-Moves first at leftmost PV-Nodes for move ordering[10], bitboard state representation[11], quiescence search[12] amongst other things. However, for one reason or another, we have not implemented these likely optimisations.

- Board Representation in a Python String

  We use a 64-character square-centric[13] string to represent game states for two reasons.

  1. Search, which is a heavily used operation in our program, can be done in amortized constant time since the string's indices stably map to board coordinates. For example, searching for whether the square at (5,5) is vacant is done like so, board[21].isspace(). Note that a set of coordinates, like (5,5), is mapped to an index location via the following lambda formula:

```
lambda c: c[0] + (7 - c[1]) * 8

where c is a tuple of coordinates, e.g. (5,5) -> 21
```

  2. A transposition table[14] borrows from the idea of memoization, as we cache previously visited states with some relevant information, such as depth, best move, and minimax value. Our transposition table takes the form of a (ordered) dictionary where the key value pairs are states and relevant information to describe the state, respectively. To be a key in a dictionary, the object must be hash-able. To be hash-able, the object must be immutable. The issue with using dictionaries and lists for state representation is that these data structures are mutable objects in Python, and, hence, cannot be keys in our dictionary. Thus, a string is suitable.

- Move ordering

  Move ordering is an optimisation that best complements an alpha beta search, since it is a way to foster more cut-offs/pruning. Before we explore our next moves, we sort them by their evaluation scores, and prioritise searching moves with higher evaluation scores first. The assumption we are making here is that nodes with higher evaluation scores are more likely to be where we find the best move sequence and, if that happens, there will be more cut offs/pruning. More cut offs result in a reduction in the size of the search-tree.

- Transposition table

  The idea for a transposition table, described above, arose from the fact that the number of times we re-evaluated previously seen states and subtrees was not insignificant, and that much time was being spent recalculating critical details pertaining to these states and subtrees. Thus, we saw a great opportunity for time-savings. We verified the efficiencies gained via a 'hit' count, which identifies how often the transposition table is used to save

[9] https://www.chessprogramming.org/Automated_Tuning
[10] https://www.chessprogramming.org/PV-Move
[11] https://en.wikipedia.org/wiki/Bitboard
[12] https://en.wikipedia.org/wiki/Quiescence_search
[13] https://www.chessprogramming.org/8x8_Board
[14] https://en.wikipedia.org/wiki/Transposition_table

us the trouble of recalculating critical information associated with a state, such as its minimax value, depth and bound. To ensure the resource constraints are respected and consistency is maintained between depth-iterations, we wipe the memory of the table from the previous depth-iteration. In an iteration, the maximum size for the transposition table is a configurable setting we have (somewhat arbitrarily) set at 10 million items. If we reach the maximum, we prioritise which states should remain, and which should be removed according to the least recently used (LRU) algorithm[15]. This algorithm assumes that the least recently used state is the least likely to be called upon in the future, and so is the best candidate for removal.

We also note that the transposition table look up operation can be performed in amortised constant time thanks to the (ordered) dictionary data structure employed.

- Iterative Deepening

The merits of an iterative deepening framework[16] in games like Chess are well-documented. In our experience, Expendibots is no exception as it proves to be an asset to our program. The major advantage is that it facilitates the use of a time management strategy, allowing the program to spend a consistent amount of time to ponder moves across turns. The time allowed is yet another configurable setting that we have set to 0.5 seconds per turn based on the experiments and manual tuning we have carried out. There is a school of thought that states that there is merit to increasing the time allowed to, say greater than 1 second per turn, because, that way, the algorithm is allowed to look deeper down the tree and make better informed decisions. However, we argue that, given the time permitted for our program to finish the game is capped at 60 seconds, we should not to spend too long to decide an action because the benefit from looking deeper down the tree is not guaranteed and bears the risk of wasting precious time. For this reason, we believe the cost associated with raising the 'time allowed' constant outweighs the potential benefit of finding a better next action.

- MTD-f

The MTD-f algorithm[17] is founded on the premise that the narrower the window passed to alpha-beta's initial call, the more cut-offs that will occur. Where cut-offs occur, efficiencies are gained. This algorithm exploits this fact by invoking numerous zero-window searches to converge on the minimax value. The 'f' in MTD-f represents the 'first guess', which is always the first bound to be passed in the zero-window search to alpha-beta. It is widely acknowledged how important the first guess is in terms of efficiency, as the algorithm performs fewer calls to alpha-beta the better the guess, 'f'. To this end, we make our first guess to be the best score returned from the previous iteration in our iterative deepening framework. This is intuitive because the best score at a depth of x is likely to be close to the best score at depth x + 1, making it a good candidate for a first estimate.

- Opening Strategy

With the benefit of having watched hundreds of games play out, it dawned on us that our algorithm's opening four moves are highly predictable. However, precious time was still being allocated to calculating the next best move analysing many possibilities for predictable output. We therefore conceived an opening strategy that dictates the opening four moves where applicable, giving the game-playing agent more time for more worthwhile, time-consuming decisions further on in the game.

[15] https://en.wikipedia.org/wiki/Cache_replacement_policies
[16] https://www.chessprogramming.org/Iterative_Deepening
[17] https://en.wikipedia.org/wiki/MTD-f