The University of Melbourne
COMP30024 Artificial Intelligence

Project Part B:
Playing the Game

Boomers: Tuan Khoi Nguyen - 1025294, Nicholas Wong - 926736

May 12, 2020

Overview

A warm welcome! This is the report for Project B, which goal is to create an invincible agent to be run on the game of *Expendibots*. In this report, we, the *Boomers*, will briefly demonstrate the process of creating the agent and, find the path to destroy all opponent's stacks.

1 Search Formulation

Expendibots is a deterministic game of perfect information, where all the state representation are provided. It can be represented as a search problem:

1.1 Initial state

Initial state is given by the standard beginning of *Expendibots* for each side: 12 single token stacks, divided into 3 groups of 4 stacks. For the implementation, these stacks will be read into two collections of stacks in the from of `Counter[(x, y)] = n` - a stack of `n` tokens on `(x,y)`. Each list will represent either black or white stacks on a board, and therefore provides full information for each state representation of the game.

1.2 Actions

The actions involve:

- Moving a number of token in 4 directions: `left, right, up, down` within the range, given by the stack's number of tokens. Moving into a space occupied with black tokens is not allowed.
- Blowing up the stack: `boom`, triggering adjacent stacks to blow up and creating a chain reaction.

Applying one of the available moves will update the state into resulting state.

1.3 Terminal test

The terminal test for *Expendibots* is the same as chess, having 3 possible outcomes: Winning, Drawing and Losing.

1.4 Utility function

With the terminal test above, *Expendibots*'s function is a zero-sum game, and therefore the utility function will have these outcomes:

- Winning: +1
- Drawing: +0
- Losing: -1

2 Choice of strategy

For a standard one-on-one game of *Expendibots*, each token has a maximum of 5 moves, and for a board of 12 tokens, up to 60 moves are available for each turn. With the constraints in resource usage - an allowance of 60 seconds and 100MB each turn, development needs to come up with a strategy that can be effective within the expansion limits.

2.1 Strategy planning `<boomers.player.get_action()>`

With the limited allowance of expanded nodes, if full evaluation is made for up to 60 available moves, it would only able to reach around a 2 to 3-ply level. Therefore, for different cases of moves, different approaching methods should be implemented, to provide effectiveness on the agent. By researching on a variety of websites on chess programming, many methods are considered and applied to the strategy forming of *Boomers*.

A common rule in chess programming is that the game is divided into 3 different stages and applying different strategies to each one: The Opening, the Middlegame and the Endgame [2]. As a similar deterministic zero-sum game of perfect information like chess, it is true that each stage of *Expendibots* should require different approaches.

2.1.1 Opening

The opening has many available moves, and implementing advanced search on the beginning might be too excessive and unnecessary for development. In chess, there is already an open book for the opening moves, and chess programming is implemented with a system of database to fast search through the open book [3, 11]. As the time for development is limited, a safe opening strategy for the agent is evaluated, and then hard-coded to save up time while maintaining good development for the first steps, then maintain search at a low depth until middle game.

In *Expendibots*, opening ends when both sides have less than 9 tokens, and moves to the middlegame stage.

2.1.2 Middlegame

The middle game is where the actions are well evaluated in chess, such as the perfect time for castling or detecting pawn rams [4]. Similar to chess, *Expendibots* has a lot of potential decisions to consider for defense and attack. With the use of minimax search to a larger depth, a small number of impact factors of game states have more influence to the evaluation of *Boomers*, such as:

- Potential splits that can be connected and destroy consecutive opponent's stacks
- Avoiding dangerous spaces that may result in friendly-fire
- Encourage center control and avoid corners

In *Expendibots*, middlegame starts after a game opening, and ends when both sides have less than 3 tokens. Then the final stage, endgame is reached.

2.1.3 Endgame

The endgame in chess games requires a lot of changes in strategy and implementation to reach a checkmate [5]. In *Expendibots*, the strategy is different from chess. *Expendibots* is only limited in the given tokens, while chess can have sudden situation changes such as queening a pawn. But regarding of this, it still needs to reach to a high ply search to be able to win the game.

While an evaluation function changes a lot in chess going from middlegame to endgame [2], *Expendibots* requires fewer changes to be able to bound the necessary factors, and therefore an underlying temporal difference learning is suitable to update these changes. At this stage, transposition table becomes very useful, the same way it is applied in chess: to skip evaluating repeated nodes [12].

3 Choices and algorithms

With self-playing, self-evaluation and further research, a number of methods and algorithms are found to be suitable to improve the agent's usage and performance:

3.1 Representation data structure `<boomers.player>`

The agent implementation has two main classes to operate:

3.1.1 State representation `<boomers.player.Player>`

`Player` represents the board state's information. The mutable attributes of the class consists of 2 `collections.Counter` to store the number of tokens of each color in a square.

Initially these values were defined as 2 lists of stacks in the form of `[n_tokens, x_coordinate, y_coordinate]`. However, noting that to access each location will have to take at most O(n) time to search through the list, the implementation is changed to `collections.Counter` to reduce the access time to O(1).

3.1.2 Search tree node `<boomers.player.Node>`

`Node` is used as the element of the minimax search tree. The mutable attributes of the class consists of pointers to the children nodes and parent node, a `Player` to represent the desired state, and a shared transposition table of type `collections.Counter` with its parents and children to look for repeated nodes.

3.2 Opening moves `<boomers.player.Player.action()>`

To save up time on evaluating nodes, chess engines use the open book system to look for initial moves. Being a newly customized game with limited time, *Expendibots* is not yet to be developed in the means of strategy, and therefore such database for the opening moves of *Expendibots* is not available yet. To save up time, a hard-coded strategy is implemented for the initial moves.

The moves include combining 2 stacks from the upper middle cluster into 1 and do fast jumps to approach the opponent's middle stacks. Most counter cases will result in a 1-on-1 trade, and a successful route can destroy up to 4 stack of the opponent using only 1 token. Step-by step includes:

1. Combining 2 upper middle stacks into 1
2. Jump the combined stack up 2 tiles
3. Jump 1 token from the combined stack up to 2 tiles if applicable, otherwise start the search using other algorithms
4. Start the search using other algorithms

3.3 Evaluation function `<boomers.player.Player.evaluate()>`

The evaluation applies the philosophy from modern chess engines: Keep the evaluation simple to make the system bug-free and more maintainable [6]. With that in mind, let n be the number of token stacks on board, the evaluation function includes:

- The total number of tokens of each side to determine which side the advantage is on. It takes O(n) iterating through each stack. To let the agent be more aggressive, award for destruction of an opponent's token is set to be larger than the penalty for a friendly fire.
  - Returns an infinite value if win/loss state is detected from this.
- Small bonus reward for stacks taking control of the board center ($x \in [1, 6], y \in [3, 4]$) and penalties for stacks lingering in the 4 corner tiles of 4 directions. Taking this factor in will make the agent more aggressive and lean forward. Calculating these will iterate through $6 \times 2 + 4 \times 4 = 28$ tiles to look for stacks.

- Closest distance between 2 stacks of different colors of the form $\frac{1}{Euclidean\_distance}$ square units. To avoid the tokens from lingering around, this function is minimally used with a small weight to encourage stacks to move closer to opponent. Takes $O(n^2)$ for 2 iteration loops to compare each pair of stacks on 2 sides.
- The token number of each color on a shared cluster, and of player's color on its one-sided clusters. This is to encourage stacks to approach to connecting positions that can trigger explosions of many opponent's stacks, and discourage the player tokens to stand next to each other, which can result in creating friendly-fire stacks. The time complexities are:
  - Clustering stacks: $O(n)$ in total for consecutive recursive calls of each clustered stack.
  - Detecting all different clusters: $O(n^2)$ for 2 loops iterating through each stack and cluster

Therefore, in total, each evaluation will take $O(n^2)$ time.

3.4 Minimax `<boomers.player.Player.expand_minimax_tree()>`

As from the Week 4 lecture, the minimax algorithm is a good choice for deterministic games with perfect information, in which *Expendibots* is classified in. It focuses on minimizing the opponent's desirability and maximizing the player's outcome, and so all possible moves are examined using an evaluation function. Step-by-step and recursively, it propagates up the tree level and returns the best move possible. The deeper the depth, the more efficient minimax works.

Time guarding is also implemented to handle search cases that take too long, such as quiescence's worst cases, or bad move ordering. If time limit is exceeded, the current best leaf node so far will be returned.

However, the branching factor for minimax in *Expendibots* is very large. With a maximum branching of up to 5 moves per token, a standard *Expendibots* can be available of up to 60 moves, and therefore more likely to average the branching factor to around 25 to 45 moves. Assuming the average branching factor to be 30, both space and time complexity can go up to $O(b^m)$, that is $O(30^{max\_depth})$. With the usage constraints, this branching factor will only be able to expand to less than 3-ply using the standard minimax. This is where $\alpha - \beta$ pruning comes to save the day.

3.5 $\alpha - \beta$ pruning `<boomers.player.Player.minimax_alpha_beta()>`

$\alpha - \beta$ pruning is the improvement to the traditional minimax algorithm. It works by setting bound values α, β: the upper and lower bound. If a node is found to be definitely worse for the player by comparing its bound to the current bound, it will not be examined, reducing the resource usage for the search.

On average the $\alpha-\beta$ pruning can reduce the time and space complexity of up to $O(b^{\frac{3}{4}m})$. With a good ordering that makes the best moves to be expanded first, the complexities can even be reduced to $O(b^{\frac{m}{2}})$. Therefore, using $\alpha - \beta$ pruning, the search can go as deep as 4-ply to 6-ply, playing a decently good game in comparison to a standard player.

3.6 Boomers' Inverted Quiescence Search `<boomers.player.Player.quiescence()>`

In game playing search exists a problem called Horizon Effect: A move might be a good choice in the cutoff depth, but will lead to a bad result that is not detected because of the search's depth limit [8]. This effect can be found right in the greedy search's downside: It only looks for an instant good move without knowing if it is a trap. Quiescence search solves this problem, by detecting moves that may cause sudden swings and expand them, and only terminate when all expanded nodes have reached a "quiet position" [1]. For example in chess, the "king in check" positions are further searched to avoid future harming.

Unlike chess that involves with many potential traps, *Expendibots*'s board is more direct: as long as a stack keep its distance, it will be safe from being boomed. Furthermore, cluster checking is already implemented in the evaluation function, making detecting traps no longer necessary. Therefore, the idea of a modified variation of quiescence is came up, and implemented to empower *Boomers*' strategy:

- While the original quiescence searches on the danger moves to detect potential traps [1], the modified quiescence will pick "quiet nodes" to search on instead - leaf nodes that does not make much difference to the original state. This will make these child nodes continue to develop their strategy fully like its "unquiet" siblings, to find a better solution, such as approaching to opponent's cluster.

- The terminal condition of the original quiescence is when the child's value reaches close to the "quiet state" [1]. For the modified version, it is the other way around: the node's terminal condition is when it detects a sudden change, such as a big boom or being gang-boomed.
- The method to detect if a node is quiet or not, depends on the implementation of each individual. For *Boomers*, the node is considered an unquiet node if the difference between its state evaluation and the original state evaluation exceeds the `QUIET_THRESHOLD`, a limit parameter. For *Boomers*'s *Expendibots* agent, this value is 0.25 by default.
- To save up time and usage, a simplified version of this method can be used. That is, from the quiet node, start a new minimax search tree to a limited depth and return its newfound value resulted on that tree. This version is also implemented in *Boomers*'s *Expendibots* agent.

The worst case for the quiescence search depends on how deep the nodes are expanded. For an expansion limit of n, in the worse case it can make the tree complexity expand up to $O(b^{m+n})$, which happens if all leaf nodes satisfy the quiet condition and get expanded. This usually occurs when same states are repeated and the search algorithm denotes the state as a quiet state. Therefore, the best companion to go with this method, is a transposition table to eliminate the repeating states.

3.7 Transposition table and Zobrist hashing `<boomers.player.Player.to_hash()>,<boomers.player.Node.table>`

As mentioned in the game ending, transposition table is a vital factor to cut down the state repetition, and can dramatically double the reachable depth of search in the same usage limit [1]. The table keeps track of the expanded moves in the tree, or player's action history [5, 12]. However, the process of storing the table and retrieve a state's hash key is also a considerable factor, especially in a game with many squares [12]. *Expendibots* has the same dimension as chess, and similar states can be occurred very frequently, especially observed in the effect of detonating any one of the stacks in a cluster will always clean out the same cluster.

Zobrist hashing creates a pseudorandom 64-bit number for each space and each type of token stacks on the board, and uses XOR operation to retrieve the hash key [13]. For a game of *Expendibots*, the number of array element is $n_{color} \times n_{max\_stack} \times height \times width$, that is $2 \times 12 \times 8 \times 8 = 1536$ values. To test its efficiency in *Expendibots*, the method is brought into comparison with using tuple of sorted coordinate tuples as the hash key. The former takes half of the time the tuple method needed to finish the search (`2.475s/4.533s`). Being the faster option, Zobrist hashing is chosen as the key retrieval method for the transposition table of *Boomers*.

The transposition in *Boomers* is implemented as a `collections.Counter` to make the hash retrieval process takes constant O(1) time. To find the key for each state, all board squares will be iterated through, making each key retrieval takes constant $8 \times 8 = 64$ operations.

3.8 TDLeaf(λ) `<boomers.player.Player.update_TDLeaf()>`

As introduced in Week 5, TDLeaf(λ) is a temporal difference machine learning implementation for game playing, by updating the weight after each turn. The algorithm is chosen to operate concurrently with the state update for its simplicity, using less computation and external connection, given that the set up is already done by the environment itself [1]. Furthermore, the evaluation uses simple functions with a seperate weight vector, hence more compatible for setting up $TDLeaf(\lambda)$.

To better evaluate the state representation, numpy is used to operate calculations during training. For *Boomers*, λ is set to 0.1 to make the evaluation function more realistic, while the Learning Rate η is set to 0.1 as a safe choice. Step-by-step includes:

Applying chain rule:

$$\frac{\partial r}{\partial w_j} = \frac{\partial eval}{\partial w_j} \times \frac{\partial eval}{\partial r} = [1 - tanh^2(eval(s_i^l, w))]\frac{\partial eval}{\partial w_j} = (1 - r_i^2) f_j(s)$$

And converting the coefficient to matrices:

$$\sum_{m=i}^{N-1} \lambda^{m-i} d_m = \begin{bmatrix} \lambda^0 & \lambda^1 & \dots & \lambda^{N-1-i} \end{bmatrix} \begin{bmatrix} d_i \\ d_{i+1} \\ \vdots \\ d_{N-1} \end{bmatrix}$$

The weight update function becomes:

$$w_j \leftarrow w_j + \eta \times f_j(s) \times \sum_{i=1}^{N-1} \begin{bmatrix} \lambda^0 & \lambda^1 & \dots & \lambda^{N-1-i} \end{bmatrix} \begin{bmatrix} d_i \\ d_{i+1} \\ \vdots \\ d_{N-1} \end{bmatrix} \times (1 - r_i^2)$$

The rest will then be implemented using Python's built-in loops and numpy's matrix functions.

4 Potential extensions and past ideas

Here lies the ideas that have been evaluated, or implemented, but have been put down or removed due to limited development time, or no longer being efficient to *Boomers*.

4.1 Null-Move Heuristics `<heuristics.search.Node.null_move_search()>`

Being an estimator of how desirable a node may expand, the Null-Move will assume that the player returns a null move, or in other words, passing the turn to the opponent without making any move. The minimax search will run on this case with a limited depth and keep track of the nodes that produce cut-offs. Going back to the standard minimax search, the nodes with the concurrent actions to the saved nodes will be expanded first [9].

With a null-move search to 2-ply, it would take up $O(b^2)$ for space and time. Using this effectively makes the minimax search complexity reduced to $O(b^{\frac{m}{2}})$. The total complexity will then be reduced to $O(b^{\frac{m}{2}})$ for search trees with a maximum depth greater than 2.

This method was coded and put in the old implementation. However, with the use of quiescence search, simple implementation of null-move search is unnecessary, as the agent's maximum depth of search are now mostly relatively small (2 for opening), making it no longer useful for time reduction, and might end up taking the same time to sort the child nodes.

4.2 Tapered Eval

Tapered Eval is an evaluation function that is used to avoid sharp differences between phases [14]. It evaluates using interpolation on the different phases.

The use of Tapered Eval can help simplify the evaluation function, making the trade between simplicity and time complexity more efficient [6]. However, it requires that both evaluation for different phases be concurrently calculated, and require a lot of small elements retrieval to be able to make up the efficient interpolation, observed from Pettersson's framework [14]. With limited time for development, the method may not work perfectly if little time is spent on, and therefore will be looked up to again when there is more space for improvement.

4.3 History transposition table `<boomers.player.history>`

Based on the idea of transposition table for the search tree, a history table using Zobrist Hashing was also created, with the purpose of avoiding movement cycles, and further reduce the excessive node expansion. However, noting that most states in *Expendibots* can never go back (such as booming), storing these would not make much difference, hence ineffective. A proposal idea to this is to develop a checking method that detects moves that can hardly go back (definitely appending all boom moves in), and not appending it to the history table to save up usage. More research is needed to make the most out of this table.

5 Testing and results

5.1 Testing agents `<random_agent>`,`<greedy_agent>`

To test the efficiency the agent, 2 player modules are created: a `random_agent` that picks a random available move and a `greedy_agent` that takes on the move that gives the best immediate outcome based on the number of boomed tokens and closest Euclidean Distance:

- *Boomers* is able to defeat `random_agent` from 5 moves to around 15 moves.
- *Boomers* is able to defeat `greedy_agent` at around 30 moves. The reason for taking so long is because the greedy agent lingers around as well, taking the *Boomers* a long time to approach as the distance weight function only takes a small part in the evaluation function.
- When self-played against each other, from observation, *Boomers* tend to make symmetrical moves and lead to a draw. This behavior is likely caused by the static attributes shared by the `player.py` file, such as Hash tables of evaluation weight vector.

5.2 Test Scripts `<test_minimax.py>`,`<test_search.py>`,`<test_basics.py>`

Test scripts are also used to check if the functions run smoothly, with measurement packages such as `time` or `guppy`. From the test scripts, the results indicates that:

- For a *Boomers* agent with the minimax cutoff depth of 2, it takes around 2 to 5 seconds to return an action, using around or less than 10 Megabytes of memory.
- For a *Boomers* agent with the minimax cutoff depth of 3, it takes around 35 to 50 seconds to return an action, using around 105 Megabytes of memory.

Overall

At the end of the day, over 1500 lines of code in total are spanned for this project. Despite the effort, there is definitely more that still can be improved and *Boomers* is not at its best, but we are proud that this is our original work and research. With more space for development, the *Boomers* will definitely flourish its best in the future.

References

[1] Russell, Stuart J. Norvig, Peter.
Artificial Intelligence: A Modern Approach (3rd. edition).
Upper Saddle River, New Jersey: Prentice Hall, 2009.

[2] Phases of a Chess Game,
https://www.mark-weeks.com/aboutcom/ble24phs.htm.

[3] Opening - Chessprogramming wiki,
https://www.chessprogramming.org/Opening.

[4] Middlegame - Chessprogramming wiki,
https://www.chessprogramming.org/Middlegame.

[5] Endgame - Chessprogramming wiki,
https://www.chessprogramming.org/Endgame.

[6] Evaluation Philosophy - Chessprogramming wiki,
https://www.chessprogramming.org/Evaluation_Philosophy.

[7] Quiscience Search - Chessprogramming wiki,
https://www.chessprogramming.org/Quiescence_Search.

[8] Horizon Effect - Oxford Reference,
https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095944934.

[9] Question On Null Move Pruning,
http://www.talkchess.com/forum3/viewtopic.php?t=65024.

[10] Null Move Heuristic - Wikipedia,
en.wikipedia.org/wiki/Null-move_heuristic.

[11] Chess Openings and Book Moves - chess.com,
https://www.chess.com/openings.

[12] Transposition Table and Zobrist Hashing - Adam Berent,
https://adamberent.com/2019/03/02/transposition-table-and-zobrist-hashing/.

[13] Minimax Algorithm in Game Theory — Set 5 (Zobrist Hashing) - GeeksforGeeks,
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/.

[14] Mediocre Chess: [Guide] Tapered Eval,
http://mediocrechess.blogspot.com/2011/10/guide-tapered-eval.html.