COMP 424 Final Project Game: *Colosseum Survival!*

Ruari McEwan 260972007
School of Computer Science
McGill University

Vlad Zimmerl 261113967
School of Computer Science
McGill University

1. Motivation

The goal of this project was to use the material about game-playing algorithms learned in class to build an AI algorithm to play the game *Colosseum Survivor*. In order to understand this game better, we played multiple games against each other. Our algorithms and heuristics are based on these initial experiments. We saw the strongest moves were generally in the middle of the board, close to the opponent, or far from our own position. We also noticed that the best moves were the ones that put us in a bigger area than our opponent. Finally, finding a game winning move is a crucial part of any strategy. We ended up using these three different heuristics as a means of limiting the branching factor in a MiniMax algorithm. We used a pruned MiniMax tree because we realised that although we had some decent heuristics, they were not strong enough for a full on alpha-beta style algorithm. This allowed us to benefit from the breadth of search of MiniMax, while cutting away weaker branches.

In terms of results, our very first iteration, using a depth limited to three levels, and only using two heuristics, was already able to beat the random agent every time. Our latest version, incorporating all heuristics and searching to an arbitrary depth can also beat the random agent every time, but it also beats our first iteration almost every time, and beats all other iterations consistently. It is undefeated against human opponents.

We had to adapt our algorithm to many different factors. The time limit forced us to think about efficiency and come up with algorithms that delivered results but not at the cost of computation time, leading to interesting decision making in the design process. We viewed this project as fun honestly; it was really cool to put into words what we saw behind a winning strategy. Overall, this project was invaluable in understanding the practical applications of course material.

2. Detailed Description of Agent Design

Algorithms from course material:
BFS "L2, slide 27", MiniMax "L6, slide 18"
Note 1: Some functions using BFS to traverse the board have a small helper method used to check the validity of exploring a new square. These all have similar implementations (check for out of bounds, check for wall, check for visited squares), with slight differences which are seen in the algorithms using these methods.
Note 2: Our code and logic is divided into multiple sections (0, A, B, C, D, TEST), the

documentation will follow these sections.
Note 3: Our code is structured such that methods will never call a method defined below it. The documentation follows this layout.
Note 4: Functions are described without their prefixes and parameters. A description our coding conventions can be found at the start of the StudenAgent class definition.
Note 5: This part of the paper is longer than 2 pages (which was the recommended legth), but we used very loose spacing (better for reading), and our code is now almost 1000 lines long, and we could not make this description shorter without sacrificing the accuracy.

2.1 Section 0: Constants and Timeout function

`timeout():`
True if we have used more than 1.9 seconds. This function is used to wrap all expensive processes in time-checks so that they are not run if we are about to exceed our time, and so we just return nothing instead.

2.2 Section A: Getting the top 3 best positions to move to

The first step is getting some promising squares to move to. We found that moving as close to the opponent as we could, moving close to the middle of the board, and moving far away from our own position were generally the best moves. We then narrow those down to just 3, by using our main heuristic, which is: how many squares do we reach before the opponent minus the number of squares our opponent reaches first.
Note: only the last method in this section is used outside of this section.

`get_distance():`
Get the Manhattan/walking distance between two squares

`get_promising_squares():`
Get promising positions to move to based on our heuristics
We will return up to 9 promising squares (from all our possible moves), these are:
-the positions closest to the opponent (one on each of the 4 sides)
-the positions furthest from our own position (one in each of the 4 directions)
-the position closest to the middle
1. initialise each of these with our own position
2. run BFS for *max_step* number of iterations from our position (these are all the squares we can move to)
3. for every square replace one of the most promising squares if it fits the criteria better
4. return these squares as a list

`get_best_squares():`
Get the 3 best positions by narrowing down promising squares based on our heuristic which is defined by the number of squares we reach before the opponent minus the number of squares the opponent reaches before us
1. get a list of promising positions/squares (use method above)
2. we will establish a heuristic value for each position using the following process:

3. run simultaneous BFSs starting from every promising position as well as from the opponent's position
4. for every BFS, if it reaches a square before the opponent's BFS does, add +1 to the count of the corresponding square
5. if it reaches the position at the same time as the opponent, add +1 to a count of "neutral" squares for the corresponding promising position
6. once we are done we can do the following math for every promising square and its corresponding values found:
7. $my\_squares = square\_reached\_before\_opponent$
8. $opponent\_squares = total\_squares\_explored - my\_squares - neutral\_squares$
9. $heuristic = my\_squares - opponent\_squares$
10. return up to three squares with the highest heuristic, along with their heuristics

2.3 Section B: Efficiently find a game winning move

We are able to do this ultra-efficiently by checking if a position on the board connects to two (not directly connected) walls which are both connected to the perimeter of the game board. Although this misses the case when a structure is not connected to the outside of the board and we can seal the opponent in this area, this almost never happens (especially against a competent opponent). Instead we often seem to win by splitting the board and being in the bigger area. This allows us to make algorithmic assumptions which greatly reduce the time complexity of finding a winning move.
Note: only the last method in this section is used outside of this section.

`get_opposite_barrier():`
Get the opposite representation of the barrier

`does_wall_exist():`
Checks if the wall exits on the board

`get_wall_neighbors():`
Get each of the 6 walls that could be connected to this wall if they exist

`get_wall_set():`
Get the set of walls which connect to the outside barriers on the border of the game board.
1. add the walls surrounding the board to the *border_set*
2. for every wall on the board:
3. find its neighbors (use method above):
4. merge all the neighbor walls (and the sets of walls they are a part of) and the current wall (and its opposite) to the same set.
5. either this is the *border_set* and all these walls are connected to the perimeter of the board, or this is a set on its own and may be connected later (we keep all such sets in a list, and use it in step 4.)
6. return the final *border_set*

`is_square_game_ending()`
Check if moving to this square could end the game.
1. take a *wall_set* as input which contains all walls connected to the perimeter of the board
2. this is a game ending square if it at least two corners of the square have walls in *wall_set* and those corners are not directly connected by a barrier.

`get_game_ending_squares()`
Get a list of game ending squares we could move to
1. use BFS in *max_step* iterations
2. for every square, add it to the returned list if it is a game ending square (use method above)

`get_possible_game_winning_moves()`
Narrow down the list of game ending squares by eliminating the ones which cannot be reached without going through another game ending square (i.e. get rid of the square if the area it secures is a subset of another game ending move's area)
1. get a list of game ending squares (use method above)
2. run BFS from the opponent's position
3. if the opponent reaches a *game_ending_square*, don't go past it, and add it to the squares we will return
4. return the game ending squares, as well as their corresponding game ending moves (use the direction in which the opponent found the square), as well as the squares reached by the opponent

`get_winning_move():`
1. get a list of *possible_game_winning_squares* (use method above). We also get the corresponding game winning moves, and a set of squares the opponent has reached.
2. for every *possible_square*
3. continue the opponent's BFS from every other square in *possible_squares* (i.e. where the search was stopped), and continue the opponent's square count.
4. run BFS from the *possible_square* (but don't go into the opponent's territory). Keep track of the number of squares we reach
5. if the number of squares reached from the *possible_square* is more than the opponent reaches, immediately return the *possible_game_winning_move*

2.4 Section C: Define some small helper methods for the next section

`get_possible_walls():`
Get the possible walls (i.e. those that do not already exist on the board) for a certain square

`get_possible_moves():`
Get the possible moves we can make given a list of squares do (i.e. get the associated possible walls (use method above))

`set_barrier():`

Set a barrier in a given chess board and also set the opposite representation of the barrier

`new_board()`:
Given a certain move and board, return a deep copy of the board along with the new barrier added to the new board (use method above)

2.5 Section D: Put it all together to make a pruned MiniMax algorithm

`leaf_max()`:
Given a list of "leaf" moves from a parent move, expand those moves and calculate the best heuristic for the parent move.
1. for every move:
1.2. get all the sub-moves (responses) of this move (use method above) based on the best squares for of this move (stored in the tree as part of this method), and add them to the tree
1.3. for each of those sub-moves:
1.3.1. see if we can find a game winning move (method from section B)
1.3.2. otherwise get the best squares to respond to it (method from section A), and add them to the tree
1.3.3 the heuristic for the sub-move is the max of the response square heuristics
1.4. now we have many sub-moves and their heuristics. keep the 4 best ones (with the lowest heuristic, since the high heuristics are good for the responses to sub-moves), prune the rest.
1.5. the heuristic of each move is the lowest heuristic from the sub-moves (strongest response to the move)
2. the heuristic of the parent-move is the maximum of the all the move heuristics

`recursive()`:
Get the heuristic for a certain move at a certain depth.
1. if the move leads to the game being over, return 1000 or -1000 or 0 based on the outcome and based on the parity of the depth (if it is odd it would be inverted based on leaf node evaluations)
2. if we are at depth=0, we are at the parent of leaf nodes. Just return the heuristic result of `leaf_max()`
3. otherwise, based on the parity of the depth, return the min or the max of the heuristics of the sub-moves of this move using a recursive call

`get_best()`:
Set the best move within 2s
1. We need use some dummy values for depths 1 and 2 to setup our data. First find the initial best squares (method in section A) and give them some dummy parent-moves so we can run `recursive()`
2. Do the following until our time is up:
3. increase the depth counter by 1
4. we use a recursive process (use method above) to get the heuristic value for each of the

moves at the top of the tree (i.e. the ones we can make next turn)
5. depending on the parity of the depth, either set the best move as the one with the highest or lowest heuristic (if the squares we are expanding are the opponent's moves, then the heuristic is calculated as high if it is good for the opponent, and vice versa).

2.6 Section TEST:

Some testing methods to print info about the board state (e.g. print all existing walls)

3. Quantitative Performance

3.1 Depth

For the size 12 board we manage to achieve a depth of 5 or 6 on average. As more walls get put down, and there are fewer options to consider, our depth increases. Our depth is the same for all branches (unless of course a move in the tree ends the game, then that branch can not be continued).

3.2 Breadth

The breadth achieved is 12 sub-moves which get pruned down to 4 moves in the next level of the tree. The number of moves considered is at any depth is 4^{depth}. This is the same for both players.

3.3 Scaling

Breadth remains the same, we will always consider the same number of moves based on our heuristic. So the breadth scaling is $O(1)$ as a function of the board size. Depth on the other hand, decreases in $O(log(n))$ as a function of the board size. This is because the time to run evaluation functions increases by $O(n)$ as a function of board size, but the number of evaluation functions we have to run at any depth increases by $O(4^n)$ as a function of depth.

3.4 Heuristic Impact

Heuristics allowed us to narrow down the breadth in order to increase the depth. The time to compute heuristics also affected the eventual depth our algorithm could reach.

3.5 Predictions (win-rate)

i. vs Random Agent: 100%. In our tests our agent always beat the random agent.
ii. vs Average Human: 90-100%. In our tests we never beat the strongest version of our algorithm, but we managed to win a couple games against a similar version.
iii. vs Other Students: 80%. Our agent is very consistent against previous versions and beats human opponents with ease, we think it will have a very good chance against others.

4. Advantages and Disadvantages

Our agent has multiple advantages. It can look far ahead in terms of move calculations, further than a human being. It can find a game winning move in most situations if there is one. It is not overly reliant on a single heuristic. We have tested our agent over many games, not once did it make a move over 2s, and not once did it throw an error.

However there is also another side to the coin. Although we are going quite deep in our search, we are only examining a limited number of moves on each level. We are using multiple heuristics, but we are heavily relying on their combined result to reduce the breadth of our search. This means we will sometimes miss moves which initially seem very sub-optimal, but would later be effective. By looking very deep our agent also sometimes notices that the opponent can force a win. But this means that we stop distinguishing between moves that lose immediately and moves that lead to a loss later, sometimes resulting in a move that kills our agent when we could have survived a bit longer (perhaps in hopes our opponent makes a mistake).

5. Previous Approaches

We have built multiple iterations of this agent over the last few weeks. These versions all improved on the previous models.
The first version of the program could go three levels deep and used simpler versions of our heuristics to select potential moves and evaluate them. This version could already beat the random agent, but it was not hard for a human to beat it. It also took a lot of time computing moves on large board sizes.
The next version slightly improved on the 2 heuristics, using the ones in our current algorithm. This version also improved some algorithm runtimes, most significantly, reducing the amount of BFS instances that were used in evaluating moves (which is done very often). We were able to reduce the computation time by a factor of 10x.
The third version used MiniMax search to go to depths beyond three levels, often achieving depths in the double digits. This version was almost unbeatable by a human. It also beat the previous versions about 80% of the time.
The fourth version implemented our new algorithm for finding a winning move at any board position. This version consistently beat the previous one. In our search algorithms, we replaced the use of the deque class with lists (since we only used append() and pop() anyway), to make sure no imported libraries were used.
The final version expanded the breadth of search to 12 moves pruned down to 4 moves (sacrificing some depth). This version outperformed its predecessors, winning up to 85% of its games and being unbeatable by human players.

6. Future Improvements

In the future we would try to improve the speed of our code, perhaps using numpy arrays or simplify some algorithms. Another thing that would be immediately beneficial would be to write a strong base of testing code for our agent to help development. We would also like to try some different approaches and potentially combine them with our current

one. One interesting approach would be using the Monte Carlo Tree Search algorithm to evaluate a bigger breadth of moves. We think this could be done very efficiently, especially by using a disjoint sets data structure to check for end of game state. We can also improve our algorithm which searches for a game winning move by considering some rare cases. Another improvement could be to exclude moves that lead to an immediate loss from the search, even if all moves eventually lead to a loss. A quantitative testing approach seems very promising as a means to find agent performance improvements. Perhaps, the best asset for improving our agent would be time (and many powerful servers) to run a large number of tests and find which approaches are truly stronger than others.