Source for this file: `https://www.overleaf.com/read/jybmqrmffwrc#fd8b96`

Turn-based Game-playing Adaptation of Best-First (Greedy) Search with Multiple Heuristics

Authors:
Matthew Spagnuolo, ID: 261048256
Rambod Azimi, ID: 260911967

1. Motivation

The goal of the project is to design and implement an AI agent for COMP424's *Colosseum Survival!* game. This agent should be capable of employing the necessary logical reasoning required to play against its opponent with the aim of optimizing its win rate over several matches and against different adversaries. This includes a randomized-move agent, other classmates' agents, as well as average-skill humans. Despite having studied various methodologies for implementing a 2-player game agent this semester, such as Monte Carlo Tree Search (MCTS), Minimax Search and Alpha-Beta Pruning, we decided to go with Best-First Search as a base algorithm. After playing the game multiple times, we identified various factors influencing win rates, leading us to adopt a strategy consisting of multiple different heuristics for different game decisions. This approach enables our player to pick moves that consistently reflect a balance of both defensive and aggressive play styles, which, while admittedly leaving room for improvement, always beats a random-move opponent, and competes quite well against our human minds.

2. Agent Design

2.1 Overall plan

The agent implemented follows base design of Best-First Search, a common AI-applied searching method used to make prioritized step-like decisions over the observed set of reachable next states. This semester, we've seen this sort of algorithm as a simple informed-search over a static state space, but in the case of this game, that aspect had to be revised. In fact, considering a complete static tree representation of the possible moves at every turn, for both players, and in the given range of board sizes is simply not feasible under the tournament's constraints, neither is it an efficient approach for maximizing win rate as Best-First Search is not an optimal algorithm. Thus, we kept its use of heuristic-exploiting evaluation function(s) and it's greedy decision logic, but implemented it to search for best moves within the scope of one turn. Moreover, to maximize the potential gain from our heuristics, we decided to have the move decision be split in two steps. This means, rather than searching once for the best possible position and barrier direction combination, we use one evaluation function for the choosing of the position and another for the direction, both having different heuristic(s). This allowed us to pick and choose which logic was more impacting for each choice, which will be discussed further below.

2.2 Algorithm

The student_agent.py file features a Step() function implemented as follows:
Preliminary computations
Initially, the algorithm explores and stores all allowed board positions reachable to our agent. This is done using recursive DFS[1] which, in our case is complete as the search tree is finite, but could possibly be re-implemented to optimize run-time complexity.
Then, we perform an important edge-case check, which is the scenario where we have the opportunity to trap the opponent in its current 1 by 1 square using a single move. This measure ensures that the agent exploits the trivial blunder, often made by random-move agents, that is to, while the other player is in range, move to a position that already has 2 sides blocked by barriers, leaving only one direction open after having to place a third barrier. To check for this, our agent uses its previously computed list of allowed position, and sees if 1) the adversary is in reach from at least one side, and if 2) its position has 3 barriers.
Step 1: Find the best position to move to
In this first step, the agent uses Best-First Search methodology to find where to go. This consists of, as mentioned earlier, leveraging heuristics, custom-designed for choosing the best position to move to. The agent iterates over all possible moves, calculating the score of each position through our scoring function, and selecting the move with the highest score. Here, the considered heuristics are: 1) number of allowed directions, 2) number of allowed positions, and 3) Distance from the nearest corner[2]. The algorithm handles situations where multiple moves have the highest score during each round by selecting the first encountered move. Throughout this iterative process, precautions are taken to prevent self-trapping or cornering, unless no alternative positions are available. Notably, the heuristics used in step are more defensive
Step 2: Find the best direction to put up a barrier
Following the "reasoned to be" optimal position, the next step is to pick the best direction for placing a wall. Once again, we follow the greedy approach, meaning the algorithm first identifies all directions available to our agent from the chosen move. Then, it iterates over each potential direction and calculates a score for each and identifies the direction with the highest score. To do so, we use a second, different scoring function, which uses one single heuristic: the difference in the number of allowed positions between us and the opponent. In comparison to the position choice, this decision is more aggressive as we try to maximise getting an edge over the adversary.
Return our move
At the end of each turn, the algorithm returns what it deems to be the optimal move combination and prints the recorded execution time to ensure compliance with the 2-second time limit constraint.

1. Depth First Search (DFS).
2. See next subsection for further detail on our heuristics.

2.3 Heuristics

After a careful study of the game, we decided to implement the following heuristics to choose a next move:

1. *Number of allowed directions*:
   This heuristic comes from assessing a possible position of the player and counting the number of elements in the generated a list of allowed directions based on the game rules. For instance, if the considered position is enclosed by barriers on both its right and left sides, the function will identify the available directions as up and down, thus h1 = 2. We established that the player's advantage increases when it has more directions to choose from. Consequently, this defensive heuristic proves beneficial for our position-choosing strategy.

2. *Number of allowed positions*[3]:
   This heuristic is the count of all feasible positions available to the player from a certain coordinate. We simply compute a list containing these positions and use the list's length as the heuristic. The size of the list varies based on the board game dimensions, which ranges from 6 to 12. Typically, in larger board sizes and in early game stages, since more open space is available, said list can be quite large no matter where the agent is, which, apart from preventing cornering, helps us less. However, as the game progresses towards its conclusion, the total number decreases and becomes more important in order to make us harder to trap. In other words, the idea behind this heuristic is to have our agent be "evasive". This is a defensive heuristic.

3. *Distance from nearest corner*:
   To get this heuristic, we use a function that computes a position's distance to its nearest corners on the game board. It identifies the closest corner and calculates the Manhattan Distance, from the coordinates to that corner. We decided not to consider barriers during the distance calculation, focusing solely on determining the Manhattan distance. As a result, this heuristic pushes our agent to stay away from the game borders and closer to the center, minimizing the risk of being cornered by the opponent, and gaining greater control over the board in the early game.

4. *Difference in the number of allowed positions between us and the opponent*:
   This heuristic is used in Step 2 of our algorithm for identifying the direction that provides us with the best advantage over the opponent in terms of available positions. The agent gets this number by computing heuristic 2 for both players given their coordinates and calculating the difference between them: $h4(agent) = (h2(agent) - h2(adversary))$. This heuristic proves beneficial for our strategy as it prioritizes choosing barriers that hinder the adversary given the already chosen position, making for a more aggressive choice.

3. Note: This heuristic is not exact, as the computed list of possible move positions can be affected by our barrier choice or by the adversary at the next turn, but the computed result is still a good estimation for scoring.

	Heuristic 1	Heuristic 2	Heuristic 3	Heuristic 4
Position Selection	YES	YES	YES	NO
Barrier Selection	NO	NO	NO	YES
Weight	0.5	0.3	0.2	1.0

Table 1: Summary of Heuristics for Position and Barrier Selection, with their respective Weights

2.3.1 Table 1 interpretation

As shown above, heuristics 1, 2 and 3 were not used in our barrier selection evaluation function (step 2), that is because solely placing a barrier either does not have much of an effect[4], or has the same, unhelpfully consistent, effect[5]. Similarly, we did not use heuristic 4 for position selection (step 1) as the agent's position does not have much of an impact on the opponent's total number of available positions.

2.4 Scoring

To evaluate the overall score of a position, or of a barrier position given a chosen position, we used, as previously mentioned, two distinct scoring functions. Since our position evaluation involves multiple heuristics, we decided to use weights in it's implementation in order to prioritize specific considerations and fine-tune the decision-making process. Here's a rundown of both of these functions' design

2.4.1 Choosing the best position

To determine a favorable position, we use the subsequent score calculation formula:
Let P be a considered position,

- Score1 = Number of possible directions the player can move to from P
- Score2 = Total number of allowed positions the player can move to from P.
- Score3 = The distance from P to the nearest corner

Prior to applying weights to the aforementioned scores, we employed min-max normalization on each individual score. Min-max normalization or feature scaling is a technique which performs a linear transformation on the given data to make sure the scaled data is in the range (0, 1). The following is a formula for calculating the normalized value for X.

$$\text{Normalized Value} = \frac{X - \text{Min}}{\text{Max} - \text{Min}}$$

4. Relates to heuristic 2 and 3.
5. Relates to heuristic 1.

After applying the Min-max normalization on each individual score, we further refine the scoring process by assigning weights, specifically [0.5, 0.3, 0.2], and eventually calculate the final score.

$$\text{Final Score} = 0.5 * \text{Score1} + 0.3 * \text{Score2} + 0.2 * \text{Score3}$$

2.4.2 Choosing the best direction for barrier

To determine the best direction for the barrier, we use a similar approach as mentioned above, but with a single heuristic. As mentioned in the Heuristic's section, we calculate the direction score by maximizing the difference of allowed positions between our agent and the opponent. Since we only have one score, it is not necessary to apply min-max normalization, nor is it to choose a weight.

$$\text{Final Score} = \text{Difference in the number of allowed positions between us and the opponent}$$

3. Quantitative Performance

In this section, we conduct an analysis of our agent's performance across diverse scenarios, using various methodologies to assess its effectiveness under different conditions.

3.1 Depth

$$Depth = 1$$

As previously outlined, our algorithm makes a move decision solely based on the current position's set of next possible states (moves) and does not consider any future moves, nor does it try to estimate adversary retaliation. Thus, the depth is 1[6].

3.2 Breadth

$$MaxBreadth = 1 + 2K^2 + 2K$$

Where K represents the maximum step size as an integer. [7]
Since our algorithm considers all reachable positions in the board when choosing its next move, the maximum number of considered states for one level of play (one turn) equals to the quadratic equation above. That is, in the case where no barriers impede our agent's ability to reach it's full maximal step range on the board. Note that as a game matures, we end up considering less and less states as the number of allowed positions decreases due to an increase in barriers on the board. Also, the minimum breadth is 1, as we always consider at the very least the current position while the game is in play.

6. That is, for any board size.
7. Intuitively, the 1 comes from the current position and the rest of the equation stems from the player's max range increasing according to the max step K.

3.3 Scaling of depth and breadth with board size

Although our algorithm's depth stays the same for any board size or game stage[8], the maximum breadth of a level of computation does increase along with the board size.

In term of complexity, while we only look one depth ahead, we have to compute heuristics for *every position considered*. Now, because our most computationally complex heuristic is our second (Number of allowed positions), which, once again, computes every available position from the starting coordinate, this means that, in one turn, we might have to make up to $(MaxBreadth)^2 = (1 + 2K^2 + 2K)^2 = O(K^4)$ calculations. To give a proper big-Oh notation, we must first relate the max step K to the board size M, and since we have that $K = floor((M + 1)/2)$, we can approximate the following form as our big-Oh notation:

$$O(M^4)$$

(Note 1: if we consider $n = M^2$ being the number of squares on the board, we can express our big-Oh notation as $O(n^2)$.)
(Note 2: just as mentioned above, we do have to consider that game's natural progression tends to decrease our computation breadth as said game ages, meaning that further complexity analysis might reveal a more forgiving big-Oh notation.)
In all cases, this is admittedly not great in comparison to other algorithms, and, thus, we'll explore possible optimizations in Section 6.

3.4 Heuristic impact

As explained above, we have implemented a total of 4 heuristics: 3 heuristics for position selection and one heuristic for barrier selection. The computation of such heuristics aim to optimize the move picked and increase our agent's win rates, but do not offer any positive impact on our algorithm's run-time complexity or in achievable search depth/breadth. In fact, some of these heuristics greatly increase the amount of computation required to make a move[9], which somewhat restrains us from implementing a more extensive search. Although, this is something that was expected when choosing to adapt Best-First Search to this game.

3.5 Win-rate tests

We conducted evaluations of our agent throughout the project, testing games against other agents provided in the base code, as well as some of our own previous iterations. For each evaluation, we would use the simulator module to auto-play 2 sets of 1000 games, respectively alternating our agent between player A and B. Additionally, we assessed its performance against an average-skill human player. The following table provides the result of each of these test scenarios.

8. E.g: early game, middle game, late game.
9. See subsection 3.3.

	Games Played	Player A	Player B	Win Rate	Max Turn Time
Test Set 1	1000	Student	Random	1.0	0.27549
Test Set 2	1000	Random	Student	1.0	0.2858
Test Set 3	1000	Student	Old Agent	86.0	0.27178
Test Set 4	1000	Old Agent	Student	84.8	0.25178
Test Set 5	20	Student	Average Human	70.0	0.25

Table 2: Summary of Student Agent Win Rates Across Diverse Agents in the Testing Phase

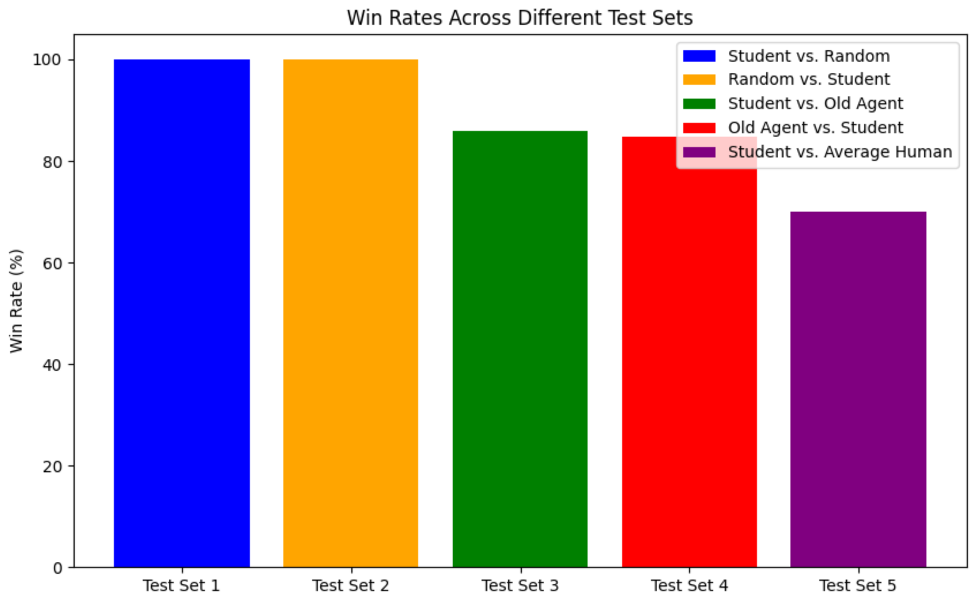

Figure 1: Student Win Rates Across Different Test Sets

3.6 Win-rate predictions

3.6.1 Random agent

As shown above, our agent consistently beats the random-move agent with a seemingly perfect win rate of 100 percent. Thus, we expect the same outcome from our agent's official evaluation.

3.6.2 Average human

We played against our developed agent 20 times and were only able to beat the agent 6 times, losing the game 14 times. Therefore, the agent achieved 70 percent win rate over the average human. We predict a similar win rate against the average human player.

3.6.3 Classmates' agents

Without having anything to truly base ourselves from, we can only make conjecture from what we know. For example, it is undeniable that our agent is on the simpler side in terms of its implementation, which, while providing for easier code, might put us at a disadvantage. Also, when comparing other studied algorithms such as MCTS[10] and Minimax with A-B pruning, we notice that our algorithm relies heavily on the use of its heuristics. This means that our tournament performance will also rely heavily on the logic of our heuristics. Nonetheless, we do trust our thought process for this project, thus we predict to have a win-rate of around 30-40 percent.

4. Approach Summary

4.0.1 Advantages

We have employed customized heuristics for both position and barrier decisions. By separating position selection from barrier selection, our agent independently assesses each aspect, which results in a balance of defensive and aggressive logic for each move. Through testing, we noticed that it offered for complex and interesting games when playing against the agent ourselves, making it hard to beat. In addition, as the evidence in Table 2 illustrates, our implemented agent's average maximum turn time is approximately 0.3 seconds, which is significantly less than the limit of 2 seconds. This performance shows our algorithm to have great promise in terms of the feasible range of potential enhancements, as it means there is much more room for the algorithm to be grow in computation. Moreover, our agent's added edge-case reinforcement is great as it allows for specific behavior in special scenarios where we deem our heuristics to be less accurate, such as ensuring that obvious trapping opportunities are always taken, but, never given to the adversary when other possibilities exist.

4.0.2 Disadvantages

The main disadvantage of our agent is its limited depth of search. While our agent uses heuristics which directly represent a move's potential for progression over the next turn to choose between currently available moves, it does not search through the further outcomes as an algorithm such as Minimax Search would do. This causes our agent to sometimes output worse results in cases where its valuation of moves is, as seen typically with greedy logic, too premature. To implement deeper depth searching, our algorithm would have to be anticipating adversary retaliation, which is not something we were able to achieve given the heuristics used and the current constraints. Another notable limitation of our agent lies in its reliance of heuristics. Since the algorithm is, as previously mentioned, limited in depth, we have that our agent's performance is closely related to its heuristics' accuracy in terms of appraising the true value of moves. Because of this, in the case of this 2-player game, where parameters such as the current total number of barriers and the number of completed turns cause the optimal strategy to constantly evolve, our agent consistently uses predetermined heuristics. This means that, although our approach of using separate scoring functions for

10. Monte-Carlo-Tree-Search

position and barrier selection helps mitigate this issue and balance out strategies, our agent still will sometimes valuate moves with less accuracy, thus choosing worse moves.

5. Other Attempts

Throughout the development of our agent, we kept previous versions, each being an other attempt at a different strategy for position and/or barrier selection. This allowed us to test games opposing our latest version to its predecessor, thus making it possible for us to evaluate our algorithm's amelioration and better the agent. To illustrate, one of our previous agents made use of a simpler, randomized approach for barrier selection. While this agent showed a reasonable win rate against a random agent (around 80 percent), it succumbed to the superior strategy of our final agent. Furthermore, we had another predecessor with different strategy for barrier selection, this time taking into consideration and attempting to maximise the number of resulting connecting barriers from choosing a side to block. The idea here was to promote a more aggressive move, by having the agent prioritize barriers that could enclose the enemy in a small space. Although, despite the initial expectation that this strategy might enhance overall performance, results revealed its weakness when playing against our submitted agent, as it would lose most often. This is possibly due to the more aggressive nature of the decision, making our agent more vulnerable to being enclosed itself. This iterative process of experimentation and comparison played a crucial role in honing our agent's capabilities, ultimately leading to our final agent being the one we speak about it this report.

6. Possible Improvement

While our agent seemingly demonstrates reasonable results, there is certainly room for further enhancements to improve its performance. Addressing the identified limitation of a static set of heuristics, a more dynamic approach could be implemented to allow our agent to adapt its strategy based on the game situation. This could be done by incorporating a mechanism that triggers a more offensive or defensive strategy, which would provide for increased flexibility in terms of decision logic. To do so, we could implement dynamic weights for our scoring function using parameters such as the number of turns executed or the number of barriers on the board, which would actively increase or decrease the influence of certain heuristics in different game stages. For example, it may be an interesting idea to slowly decrease the weight of our "distance from nearest corner" heuristic, as staying in the middle of the board becomes less and less important as it gets more crowded.

Another possible improvement would be to increase the depth of our search, as it would allow for decision logic that considers further moves ahead, thus most likely increasing our scoring accuracy and overall performance. The drawback from this enhancement, as we discovered while testing different practices in early development, is that it would dramatically increase this agent's maximum turn time (above 2 seconds) given the current heuristic functions. As mentioned prior, the algorithm's current run-time complexity could be improved significantly, and we would surely need to work on optimizing that aspect before increasing search depth (To do so, changing the function that computes allowed move positions is a start, as it is currently implemented using DFS, which has exponential time complexity.).

Lastly, inspiring ourselves from other searching algorithms we've seen in AI, it might be interesting to look at reducing the set of moves considered per depth, that is, implementing some pruning logic. For example, we could start by looking at all the moves in the player's step range, but keep track of moves with low scores. Then, for the following turns, use the data we have about low-score positions to distinctively ignore areas with a large amount of such poor positions from our calculations. On paper, this makes sense, as, given our current heuristics, position scores are unlikely to ever increase as the game progresses since more barriers are put up. This would allow for a more efficient algorithm in terms of time complexity, which we could then use, following the last paragraph, to increase our search depth and get better results.

7. Contact

matthew.spagnuolo@mail.mcgill.ca
rambod.azimi@mail.mcgill.ca

8. References

1. **Title:** Min-Max Normalization

   Source: O'Reilly Hands-On Machine Learning

   URL: `https://www.oreilly.com/library/view/hands-on-machine-learning/9781788393485/fd5b8a44-e9d3-4c19-bebb-c2fa5a5ebfee.xhtml`