COMP 424 Final Project Report

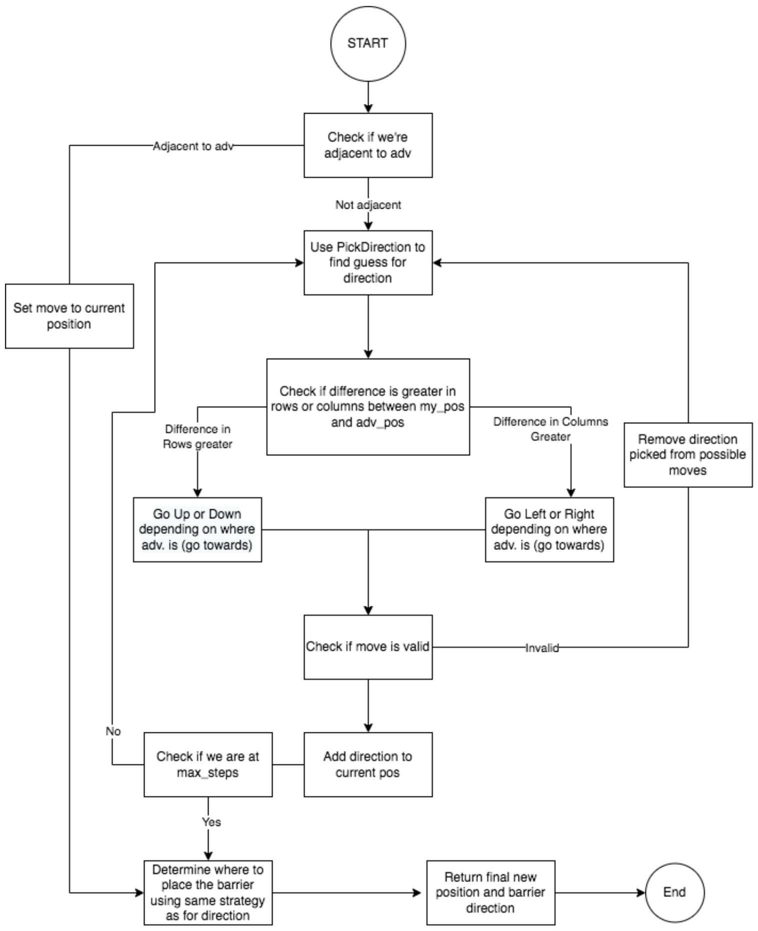
Vitoria Lara Soria
Adam Hochschild

Contents

1. Initial Technical Approach

When we first approached the assignment, we looked at the context of the problem at hand and decided to build a solution based off a simple heuristic. After some contemplation, we concluded that having our agent get as close of possible to the adversary and then placing a wall adjacent (went possible) would lead to the most wins. From this very rudimentary heuristic, we implemented improvements such as checking when the agent was adjacent, checking if the agent was boxed it, etc. This approach took a step-by-step angle to determining the best move, and separated the process of deciding where to move and where to place the barrier.

Below is a basic flowchart to outline the logic of our first approach


START
Check if we're adjacent to adv
Adjacent to adv
Not adjacent
Set move to current position
Use PickDirection to find guess for direction
Check if difference is greater in rows or columns between my_pos and adv_pos
Difference in Rows greater
Difference in Columns Greater
Remove direction picked from possible moves
Go Up or Down depending on where adv. is (go towards)
Go Left or Right depending on where adv. is (go towards)
Check if move is valid
Invalid
No
Check if we are at max_steps
Add direction to current pos
Yes
Determine where to place the barrier using same strategy as for direction
Return final new position and barrier direction
End

2. New Approach - General Idea

After implementing our initial approach we realized that we were only averaging around 80% wins when playing 1000 matches against the random agent. Therefore, we decided to implement a new, perhaps more complete algorithm. Because we were not able to find an obvious heuristic to maximize victories in the game, we chose to first code a "brute force" algorithm that tried almost all possible moves, with hope of getting more insight about the game.

The main idea of the algorithm is to make a big arraylist of all possible moves and choose the best move by classifying and dividing the master list into sublists. To decrease the time complexity of this approach, we evaluated moves in a simplified way, that could compromise correctness, but was very important in order to keep our solution within the time limit.

2.1 'Worst Move'

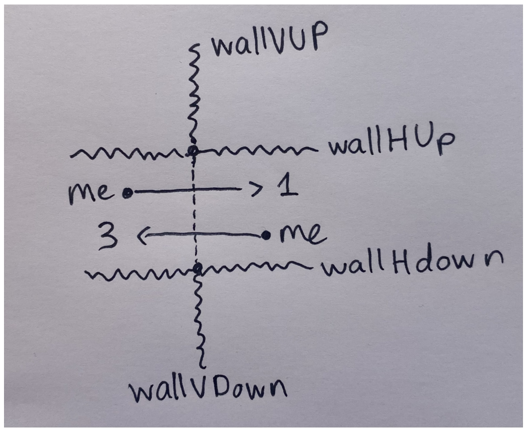
This algorithm keeps track of a variable called 'worst move', which is only used in case the agent does not find another possible move or to avoid the risk of running out of time and being forced to take a random step. This variable is first initialized with the agent staying at the same square and placing a wall in any possible direction (in order: up, right, down and left). The 'worst move' is later updated if we find a move which would tie the game.

2.2 Possible Moves Array

To create the array, we iterated through all possible combinations of steps and only added to the array valid moves, which was verified using the check_valid_step() function given in the world.py file. After filling the list with all possible moves, we try to choose the best one by categorizing the moves into 2 different categories: 'game-ending' and 'non-game-ending' moves, which then later is further divided into 'notGood', 'better' and 'adjBest'

2.2.1 Game-Ending Moves

The game can end one of three ways: a win, a tie, or a loss. The best possible move we can make at any time during the game would be one which wins us the game. Although we're provided with a function check_endgame(), this function is very computationally heavy. Therefore, in order to save a significant portion of time, one of the rudimentary simplifications we chose to implement was designing a checkPossEnd() function which takes a move and the board as parameters. The function returns True if there is a chance that the barrier placed, as a result of the chosen move, could cause the game to end. An illustration of the two possible cases is shown below



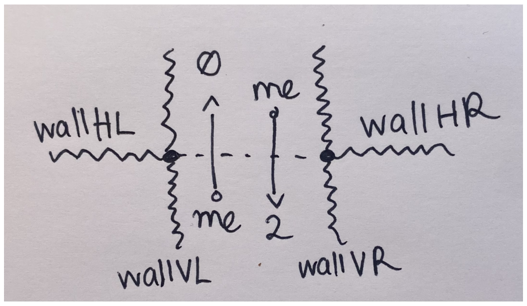


Figure 1: Showcasing all possible directions

There are two possible cases, either a wall is being placed left or right (left figure) or a wall is being placed up or down (right figure). The function will return True if the wall being placed blocks off a corresponding part of the board. In the diagram, the squiggly lines represent possible barriers, such that one has to be present on each side of the dashed line in the middle representing the barrier being placed by the move. If one of the squiggly barriers is present on each side of the dashed line, the move is potentially game-ending.

If this function returns true, we can go on to compute whether the move would really end the game using check_endgame(). This will tell us if the move is a win (take it immediately), a tie (replace worst case move with this and add to array), or a loss (disregard, move on to next possible move).

2.2.2 Non-Game-Ending Moves

If our checkPossEnd() function returns False, this is an indication that the move will not end the game. Often, this is the majority of the moves and we therefore classify them into 3 distinct sections: adjBest, better and notGood.

- **notGood** First, we categorize any move which would result in our agent being surrounded by 3 barriers or more (essentially boxed in) into a notGood array. These moves are the least desirable.
- **adjBest** At the top of our list is moves which would place us adjacent to the adversary, as well as placing a barrier between us and them. There are therefore four possible positions in relation to the adversary we could be in which would qualify for a move to be added to adjBest. This is illustrated in the code segment below

  ```
  def checkIfWallAdj(self, my_pos, adv_pos):
      mX, mY, dir = my_pos
      aX, aY = adv_pos

      if((mX == aX and mY == aY-1)): # We're on the left
          if(dir == 1):
              return True
      if((mX == aX and mY == aY+1)): # We're on the right
          if(dir == 3):
              return True
      if((mY == aY and mX == aX-1)): # We're on top
          if(dir == 2):
              return True
      if((mY == aY and mX == aX+1)): # We're on the bottom
          if(dir == 0):
              return True

      return False
  ```

  According to our approach, these moves are the most likely to box the opponent in and lead to a win. Therefore, we consider the moves in this list the most valuable ones.

- **better** Moves qualify for the better category if they do not fall into adjBest. For this category, we calculate the euclidean distance between the position of the move and the center of the board. We then sort these moves from the lowest to highest distance to the center by using a dictionary. Our approach considers the move with the lowest distance to the center the most favorable. This is because it is advantageous to be further from the borders of the board, as it decreases our chances of being boxed in and losing We calculate the euclidean distance using:

```
def dist(self, my_pos, adv_pos):
    mR, mC = my_pos
    aR, aC = adv_pos

    return ((mR - aR)**2 + (mC - aC)**2)**(1/2)
```

Initially, we actually approached this case by dividing the board into 4 quadrants and seeing if our agent was in the same quadrant as the opponent. However, the distance to center was simpler, cleaner, more versatile, and most importantly won more games. Before coming up with the solution we currently have, we experimented with reusing the heuristic from our initial technical approach of prioritizing moves which minimize the euclidean distance from us to the adversary. We found in practice though, that a major drawback to this approach was the fact that by approaching our opponent we were opening ourselves up to the possibility of losing as well.

2.3 Combining Results

After classifying the possible moves, we construct a masterList by combining the non-game-endings sublists and ties in such a way that more favorable moves are placed at the beginning. It's important to note at this point that although very unlikely, it is possible that this list could be empty. If that is the case, the 'worst move' is the one chosen to be played. Otherwise, due to the ordering of the masterList, we finish the algorithm by choosing to play the first move (element) in the list. Regarding the order, tie moves come last simply in favor of keeping the game going.

2.4 Managing Time Complexity

Throughout the code, in order to ensure we are staying within the two second time limit, we've implemented a timer which resets every turn and gets compared to time.time() to see if the difference is greater than 1.95s. If this is the case, return the worstCase move.

3. Evaluating Our Approach

3.1 Advantages

First, our agent has a considerably high winning rate against the random agent, averaging 95%+ wins when playing 1000 matches. On top of this, the agent also performs very favourably against an agent with our initial technical approach implementation. Secondly, if a winning move is available, our agent will always find and choose such. Furthermore, our approach is fairly intuitive and simple to understand. Lastly, we designed the algorithm in such a way that our agent will never be forced to make a random move, since if unable to find the best move under 2s, the agent will use the move stored under 'worst case'.

3.2 Disadvantages

The first and perhaps biggest drawback from our approach is that the branching factor of our algorithm (the possible moves array) is really big, which leads to a significant increase in the time complexity (possible fixes to this are later discussed in the improvements section). Moreover, we recognize when classifying non-end-game moves, we have rather strict criteria for a move to be considered ‘notGood’, which may result in obvious non-good moves being forced to the ‘better’ category and perhaps even being played.

4. Motivation

When approaching the assignment instruction, we analyzed the context of the task at hand, and the environment the agent was operating in. The deterministic, observable, discrete, and dynamic board in which the agents were moving around and placing barriers within led us to immediately think of the high branching factor at each state. Every state change came with a new environment, and would expand our knowledge base. The turn based nature of the Colosseum survival game did not allow us to implement a learning based solution. We therefore were constrained by using the information available to our agent from the given state. The observable environment and given functions allowed us to implement efficient heuristics to get a rudimentary idea of how to strive for optimality.

By defining a series of pre-conditions to classify each move, the agent makes decisions based on the quality of the move. We organize our list of possible moves in such a way that the least desirable are de-prioritized by being placed at the end of the list. The computationally heavy operations are only executed if the move passes a ”filter” which allows the agent to reserve computation time for moves with a higher probability of success.

5. Possible Improvements

The approach we chose has a lot of room for improvement. For example, when classifying the possible moves masterList multiple rudimentary assumptions were made as an attempt to diminish the time complexity. First, when classifying non-game-ending moves, the category of ‘adjBest’ is always considered most favourable, when that is not necessarily always the case. Moreover, in the ‘better’ sublist, lower euclidean distances from the center should not always be preferred over higher distances. However, despite our knowledge about these issues, we have not yet found more precise heuristics to consider these edge cases and improve our winning rate. We have also noticed that our agent sometimes chooses moves that are easily seen as not good but yet, due to strict conditions imposed, that move is not classified into the ‘notGood’ sublist. Therefore we conclude that if we acquire a little more knowledge about the game we could implement important heuristics that would benefit the correctness, optimality and time complexity of our algorithm. Another possible avenue for improvements could be implementing a learning-based approach wherein the agent would make adjustments based on past mistakes.