Comp424 Final Project

Abstract

We combined the two AI algorithms to make decision for our student agent, namely Constraint Satisfaction search and Best First Search. In this paper, we will discuss the mechanism, pros and cons of our algorithm. In addition, we compared our algorithm with Monte-Carlo trees, Minimax, and it greatly outperformed the other two algorithms in both winning rate and efficiency, having a 99.7% winning rate in 1000 runs compared to 86% for the Monte-Carlo tree and 92% for the Minimax.

1. CSP and BFS

1.1 Motivation

We use an algorithm which combines the Constraint Satisfaction Search and Best First Search to make decision for student agent.

The goal of a CSP solution is to assign values to variables in a way that ensures all the constraints are met.[1] Colosseum Survival involves two types of constraints: the first is the hard rules of the game, such as the fact that players are not allowed to cross a barrier and can only take a limited amount of steps. The second constraint is implicit, such as to avoid being trapped in a closed zone, which will result in losing the game. Therefore, there are a lot possible next moves for each step, by applying the constraint on them and narrow down each move's domain, we hope it can greatly increase our algorithm's efficiency and accuracy.

After the filtering by CSP, there still remain a great number of moves, which does not violate the game rules but may result in losing eventually. A good heuristic could result in a very optimal solution. And in this game, heuristic function can be easily determined: we prefer the step which can put barriers around the adversary and keep our player in open area. With this technique, we came up with a good heuristic function which can correctly rate how good each step is and make the best move. Plus, due to the time limit, we cannot simply simulate every possible outcomes for each step, so we decided to use a BFS-like algorithm which make decisions greedily so as to reduce the time complexity.

1.2 Program explanation

We create a State class to record the "state" of the game. A state of the game is defined by following attributes: the "chess board" variable which records the positions of all barriers; "my-pos" and "adv-pos" which has the position of each player respectively; "playerNo", represents whose turn it is for this specific state; "score" holds the value of score for this play; "parent" attribute records the state of play before it and "leaf" attribute records all the possible states after making this move. Therefore, by having a "State" object, we can obtain all required information to set up constraints, and determine the heuristic value for each child state. More importantly, all States will have interconnection thus forms a doubly linked list, which we can use it for backtracking later.

1. Lecture Slide 5, page 16

To begin with, we will initiate a State object using the input of the step function. Since it is "my turn" so set the playerNo to be 1, and every other parameters fills correspondingly. Then we will use the "expand" function to expand the state which we just initiate, so as to get the all possible states from the current state. The expand function can expand the state by determining all the next legal move for a player, using the function "check-valid-step". After we obtain all legal moves from current state, we would create a new State object for each new state we obtain. We will repeat the previous process every time the step function is called.

Thereafter, we can obtain the heuristic value of each child node of current state by using "evaluation" function, and decide the best move. Specifically, it will first utilize "check-end-game" function to get rid of those states in which our player loses, and gives those states a 0 score for tie result, which means we will only choose to make the tie move if we don't have other options left. For the remaining outcomes, we have a formula to give each a score, depending on the number of barriers around my-player, number of barriers around adversary player, and the total distance from my-player to the each barrier around it. From this setup, we expect to see my-player choose the next state which maximizes the number of barriers around adversary, the distance from itself to adjacent barriers, and minimizes the number of barriers around itself.

Finally, we choose the child node with largest heuristic value to be our best next step, and determine what move it made and pass it to the output.

2. Theoretical Basis

For CSP problem, we first determined its variables, domains, and constraints. The variables are all the State objects we created. Domain is all possible states of game. Constraints are expressed in two ways, the first is the hard constraint: it is forbidden to cross or place barriers in areas already marked with barriers or where adversary locates, the step player takes should not exceed the max-step, etc.; the second is the soft constraint: unless there's no other options, our player will not choose the next move which make him lose or tie, and we assign these states with a high cost. The constraint satisfaction search is delivered during the expansion of the state when we are looking for all legal next moves from the current state. It would filter all the move which violates the constraints and assign a cost of negative score to those moves which make our player lose.

For remaining child States in the domain of current State, we use BFS to decide which node is the best. The method evaluate(State) is used to return the heuristic value of each child State. When determining a heuristic value, 3 components will be taken into account: the number of barriers around my-player, number of barriers around adversary player, and the total distance from my-player to the each barrier around it. And we will choose the next move with the largest heuristic value greedily.

3. Advantages

- **Low Time Complexity**. The time complexity of our algorithm is extremely low because of the combination of CSP and BFS, we first limit the domain for each state by setting up the constraints, and use best first search to greedily search the child

state with the optimal heuristic value. The average running time for a single game is approximately 0.8s on average.

- **High Accuracy according to the good heuristic**. The heuristic function of a single step in this game is easy to determine. By observation, we found the key to winning is to stay away from the opponent, going to the block which is wide open, and trap the opponent with barrier if he's in your reach. And a good heuristic function can help you to determine a more optimal solution. Thus, our algorithm achieve a 99.7 percents success rate in 1000 runs against the random agent.

4. Disadvantages

- **The Depth of the Search is limited**. It only takes into account the next possible states from the current state, and we did not simulate the game until the game ends, thus the solution might not be optimal in the long run.

- **The Algorithm is too greedy**. We select the next State with the highest heuristic value as the best next step, but in some cases we need to choose a sub-optimal step in order to win the game.

- **Does not perform very well in the half-closed zone**. By observation, if our agent is trapped in a half-closed zone, it will not be able to exit quickly. This is also because we do not roll out the result to the end game. If we conduct multiple roll out to each child state, it should be able to choose a more optimal move.

5. Improvement

Since the depth of our search is too shallow, we can solve it by adopting the simulation and backtracking part of Monte-Carlo tree. After expanding the current state, we will obtain all the possible next state for the current state, and for each step, we can randomly rollout each next state to the end of the game, and backtrack the how many block we win/lose to each state as the score. Repeat previous step several times, and embrace this score into our heuristic function. In this way, we can increase the depth of our search so as to increase the accuracy of the algorithm, while the time complexity will still be low because we will not rollout every outcomes from the current state.

What is more, we currently put more weight on the number of barriers around the adversary than on the number of barriers around our player when determining the heuristic value, which means our player would automatically choose to trap adversary rather than earn more blocks when it can reach the adversary. This weight is fixed and sometimes it can choose sub-optimal step, so we can adjust our heuristic function by applying a changeable weight on each factor, the weight will change according to the result of rollouts.

6. Other Approaches

We also tried MCTs and Minimax algorithm on student agent and obtain a good result. The codes are attached under the "/agent" repository.

6.1 Monte-Carlo algorithm

6.1.1 Procedure:

We want to visualize search algorithms as building a search tree in which the root node represents the state from which the search began. The nodes represent the states of game, same as the node in CSP. Monte-Carlo algorithm determines the most optimal move from a set of moves by Selecting, Expanding, Simulating and Updating the nodes in the tree. It is repeated until the time is up, which is 2s in our case.

- **Selection:**

  The selectPromisingNode(RootNode) method was used for the selection phase. RootNode represents the current game state. Starting from rootNode and selecting successive child nodes with the greatest UCT value until a leaf node no longer has child nodes.

- **Expansion:**

  When we reach the bottom node of the tree, we will expand the node by finding all of its next possible state, and then we randomly select a child node we just expanded to rollout.

- **Simulation:**

  The simulateRandomPlay(node C) method is used to perform a random rollout from node C, namely making random moves until the game has ended using the checkendgame(chess-board, my-pos, advanced-pos) method to determine whether it is going to end. The result of a rollout will be the number of blocks owned by our player - number of blocks owned by adversary. In this way, we expect the agent to make the move which maximize the block it occupies.

- **Backpropagation:**

  Backpropage the rollout result we just obtained back to the node which we just explored, using method backPropogation(nodeToExplore, playoutResult). The number of simulations stored in each node is incremented and if the new node's simulation yields a win, the number of wins is also updated.

6.1.2 Comparison

In general, Monte-Carlo tree algorithms are accurate to approximately 86% within a 2-second running time. Longer running-times for the Monte Carlo tree would result in improved accuracy since more calculations can be performed during the play-out phase. The accuracy of the algorithm is reduced because not enough nodes are explored in the tree. In conclusion, our final approach has a shorter run-time and higher accuracy compared to the Monte-Carlo tree algorithm. But the search depth of MCT is much greater and could outperforms CSP+BFS if longer processing time is applied.

6.2 Minimax:

6.2.1 Procedure:

MINIMAX is based on the assumption that your opponent will also play optimally. There are two players in MINIMAX: a maximizer and a minimizer. Maximizers attempt to achieve the highest possible score while minimizers aim to achieve the lowest score. In other words, you always choose the result that maximizes your points, while your opponent chooses the result that minimizes your points.

- **Construct the minimax tree** The tree is constructed by minimax(node,isMaxTurn) method. It keep expanding the node to the end game, until every outcomes is explored. Then it back propagate the result back to each node. This function will only be called once at the start of the game. And every each step after it will simply retrieve the node from the tree we just created.
- **Retrieve the optimal step from the tree** Find the node with the highest score by calling the findBestNodeWithScore() function.

6.2.2 Comparsion

MINIMAX should provide the optimal solution when the adversary is optimal, and it will be even more accurate than Monte-Carlo tree algorithms. However, MINIMAX has a much longer runtime of over 30 minutes per game. The game like Colesseum Survival has a high branching factor, which makes it hard to apply Minimax algorithm on it. In contrast, the advantage of CSP and BFS is that it does not explore that many nodes as minimax does, which greatly increases its efficiency. But Minimax can ensure the optimal solution, and could have greater winning rate than both MCT and CSP when the processing time is sufficient.