QUORIDOR GAME

A documentation for the Artificial intelligence project

Spring 99 – Dr. Salimi Badr

Group 4:

Mehrshad Saadatinia – Mohammad Reza Amini

The project consists of classes explained below:

Quoridor : this class takes care of the main game logics and has a Players array that holds the players of the game (which are instances of the Player class) and a char[][] field that holds the game board. In the beginning, the game board is made using “makeBoard()” method and the pawns of each player are put in their respective initial positions. The game takes place inside the “play()” method which is going to be explained later on.

Quoridor4player: similar to Quoridor but handles the 4 player mode of the game with all human players.

Human: extends player class and uses the “getMove()” method to get the users move in the {number:move} format for example if the user enters 1 then asks for the direction and forms a code like this {1:up} or if the user enters 2 then asks for the wall coordinates to put a wall there the code will look like {2:7&14} . the code is passed to the ”makeMove(move, player)” method of the Quoridor class inside the “play()” and the move is made , for example a wall is placed at board[7][14] in response to the code {2:7&14}.

AI: extends Player and uses “getMove()” to return a move to the Quoridor class’ makeMove() method but uses MINIMAX algorithm in the minmax method to get the best move , the minimax algorithm uses alpha beta pruning to improve the performance and go deeper in the game tree but using only alpha beta pruning we could get only as deep as 3 layers into the tree. In order to go deeper we used “filter()” method inside the Quoridor class to remove some of the unwanted or “likely to be bad” moves from the availableMoves set. Using the filtering we could reduce the time and go as deep as 6 layers down but for better speed we use only 5 layers , the minimax algorithm used will be explaind more below:

Minimax algorithm and heuristic function :

The minimax() method return the best move in form of a String[] which it’s 0 index holds the move code and the 1 index holds it’s heuristic score , the minimax checks all the moves in the availableMoves set in the Quoridor class and chooses

the best according to the eval() method that calculates how good the move is based on the heuristic function.

Heuristic function consists of 4 factors;

1. Maximizing player's Manhattan distance to it's goal apart from the walls

2. Minimizing player's Manhattan distance to it's goal apart from the walls

3. Shortest path to the next row for the maximizer player considering the number of walls it has to get past

4. The same as 3 but for the minimizing player.

We want to minimize our distance to the goal so we subtract cases 1 and 3 from 9 and as a result if the maximizing player is further away from the goal we get a lower score e.g. (9 – distance_max)

In the phase two of the project we manually set coefficients (genes) of the heuristic function but later in phase 3 using genetic algorithm we determine the best coefficients or genes and pass them to the chromosome field of AI player class.

Filter method and simplifications:

we removed the following states from the availableMoves set so the decision making will become faster and as a result we can go deeper:

1. States where a wall is put behind the opponent
2. States where a wall is placed in front of the player by itself.
3. States where a wall is placed right beside the player.
4. States where a wall is placed too far from the opponent.

Making the above simplifications we can go as deep as 6 in minimax tree

Genetic : this class implements genetic algorithm to determine the best chromosome so we can use it in the AI player heuristic and get the best performance against human opponent.

We created a population of chromosomes with the initial size of 7 and using the method “populate()” we populated the chromosome array with 7 chromosomes each having 4 genes using random numbers less than 80.

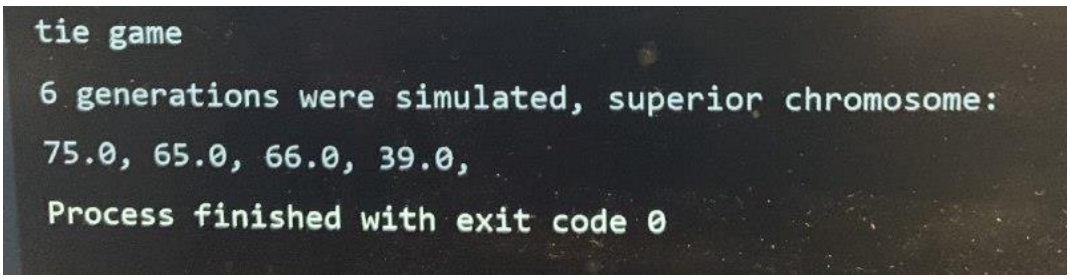
Then we created fitness array with the size of the population , each element of the fitness array holds the fitness value of each of the population members initially all are set to zero.

Then in the playoff() method each of population’s chromosomes are passed to an AI player and plays a round (two games) against every other chromosome.

Fitness of each chromosome is determined by the number of wins they get when playing against other chromosomes. AI players can make a game last forever by moving back and forth so in order to avoid this we used a moveCap integer field and set it to 120 so a game can last at most 120 moves and after that it’s declared tie game . and no scores are given to the players.

In the main method of the Training class we determine the number of generations that the simulation will do and a variable called ‘evolution number’ which determines how many of the bad chromosomes from the previous generation are replaced by the new offspring chromosomes, in fact { evolution number * (evolution number - 1) / 2 } is that number.

In “findBestChromosome()” for the number of generations we call playoff and then call “evolution(evolutionNumber)” method which chooses as many as evolution number from the best chromosomes in the “selection()” method and then calls crossover() on every two of them to create offspring then we determine the chance of mutation with a random parameter and the create as many as { evolution number * (evolution number - 1) / 2 } new chromosomes to replace the worse ones form previous generation and the new population forms the next generation this operation continues for 6 generations and these are the results:



Then we copied these numbers to the AI players chromosome field to play against human opponent.

Crossover(): chooses 2 parents and a random number between 0 and 3 is created to determine the break point then in father chromosome from 0 to the number and in mother chromosome from the number to the end is copied to the child chromosome called offspring.

Mutation(): when the offspring is created in order to create more diversity we generate a random number in range of 0 to 1 and if the number is bigger than 0.6 then mutation happens (P_m) and in the mutation() method we randomly choose a gene and replace it with a random number between 0 and 80.

The longer we can run the simulation the better results we can get considering that we don't know the fitness limit. These are the results for 6 generations with evolution number of 4 and population size of 7.