Lines of Action Project

Xiangci Li xl1066

a. This project is written in Processing 3.0, an open source language based on Java. To run this project, Java 8 and Processing 3.0.2 or higher is required. Processing is available at https://processing.org/download/?processing . The source code can be directly executed in Processing like Python IDLE. It can also be exported (by clicking file→export) and run as a single executable file.

   The GUI is designed for common players, so the required stats are only printed in the console when executed by Processing.

b. The program consists of 6 files. State class implements the state of the game. It contains the arrangement of the checkers on the checker board. It contains methods of cloning a state, moving checkers and determining legal moves for each checker. Block class implements each small piece on the checker board, including the color information of the checker on this piece. Player class implements the behavior of each player, including both human and AI. Alpha-Beta-Search algorithm is implemented here. GUI class implements everything related to drawing. Button class implements buttons, and the interactive functionality. LinesOfAction is the main file. It drives objects to calculate moves, draw the user interface and receive input from user.

   The condition of cutoff is either the execution time is more than 10 seconds, or the depth of the search tree reached the pre-assigned maximum depth. The evaluation function is based on the number of the pieces formed by contiguous checkers (A) of each player as well as the number of checkers left (B) for each player. For example, at the initial condition for 5*5 Lines of Action game, {A=2, B=6}. For a player, let the utility value be 100 if that player wins and -100 if the player loses. For other cases, in increasing order, starting from 1, rank {A=2, B=2}, {A=2, B=3}, …, {A=2, B=6}, {A=3, B=3}, {A=3, B=4}, …, {A=3, B=6}, {A=4, B=4}, …, {A=4, B=6}, {A=5, B=5}, {A=5, B=6}, {A=6, B=6} and assign them uniformly decreasing utility value.

   n=5 case. Each column is A and each row is B.

B\A	2	3	4	5	6
2		/	/	/	/
3			/	/	/
4				/	/
5					/
6					

   As showed above, specifically, let the size of the board to be n. n=5 in the default

case. The utility value for each A and B = 100 – 200/a * r, where r is the ranking of each case and a = x(x+1)/2+1, where x=(n-2)*2-1.
For different levels of difficulty, the maximum depth of the alpha-beta search tree (D) is different. For easy mode, D=1. Normal mode has D=10 and hard mode has D=1000.

The implementation of alpha-beta-search is slightly modified. Function alphaBetaSearch() returns an action. While functions maxValue() and minValue() returns a utility value. In order to find the action that corresponds to the highest utility value, I implemented alphaBetaSearch() to also include maxValue() only when the depth=0. Thus the modified alphaBetaSearch() calls minValue(), minValue() calls maxValue(), maxValue() calls minValue() and so on.

The project can be modified to be 6*6 Lines of Action game by changing the source code. In LinesOfAction file, just change SizeOfBoard in line 1 from 5 to 6.