Checkers

1. Introduction

Checkers is an old chess-playing game with a history of more than 3000 years. It is very popular in western countries. In the middle of last century, with the development of artificial intelligence, people begun to design programs which can play against human.

Until now, AI has a very glorious history playing games against human players. For example, on May 11, 1997, Deep Blue, a chess-playing computer developed by IBM, successfully beat the world champion Garry Kasparov. Checkers has gained even greater success. In July 2007, researchers from University of Alberta announced that their AI Checkers player, Chinook, was impossible to beat by human players, checkers had been solved!

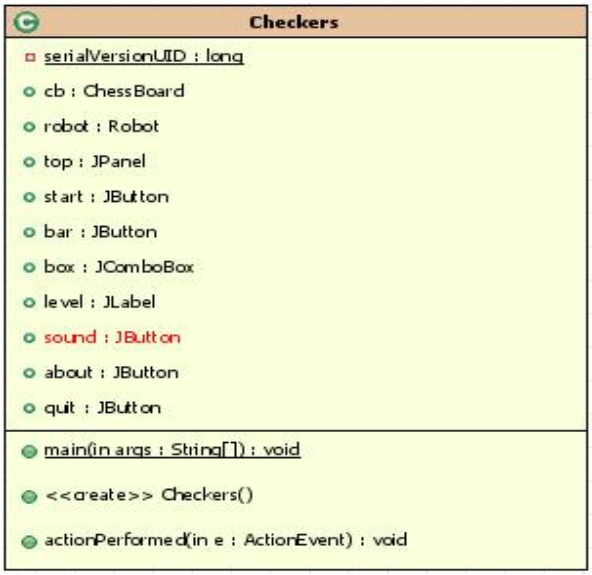
In this project, I designed an AI checkers program. I use traditional game –playing strategies, the minmax algorithm and Alpha-Beta Pruning algorithm. Finally, I are happy to find that our AI player is very hard to beat, especially when the search depth is big enough.

This program is written in java with Eclipse 3.2 as the IDE, the version of java language is JDK 1.6. The operating system is Windows Vista Ultimate.

2. Data Structure

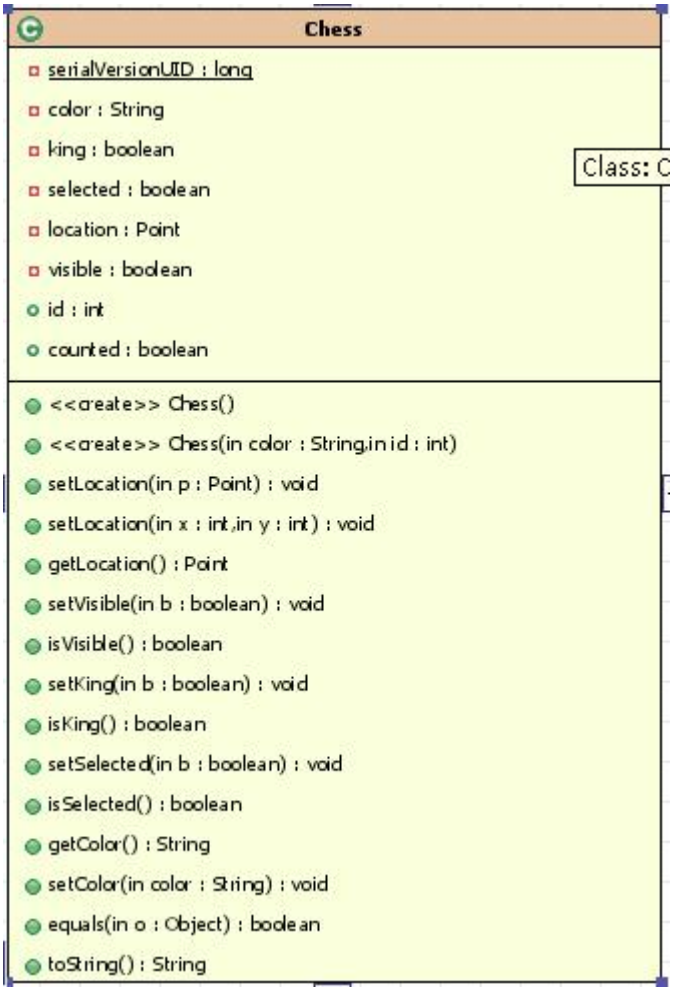
In all, I used six classes. They are

1) Checkers
   This is the main class of the program. Its class diagram is showed below. It calls other classes in the main method to start the program. Some buttons are provided for the users to make some settings.



2) Chess

Class Chess is designed to store the attributes and operations of each chess on board. Below is the class gram of Chess.

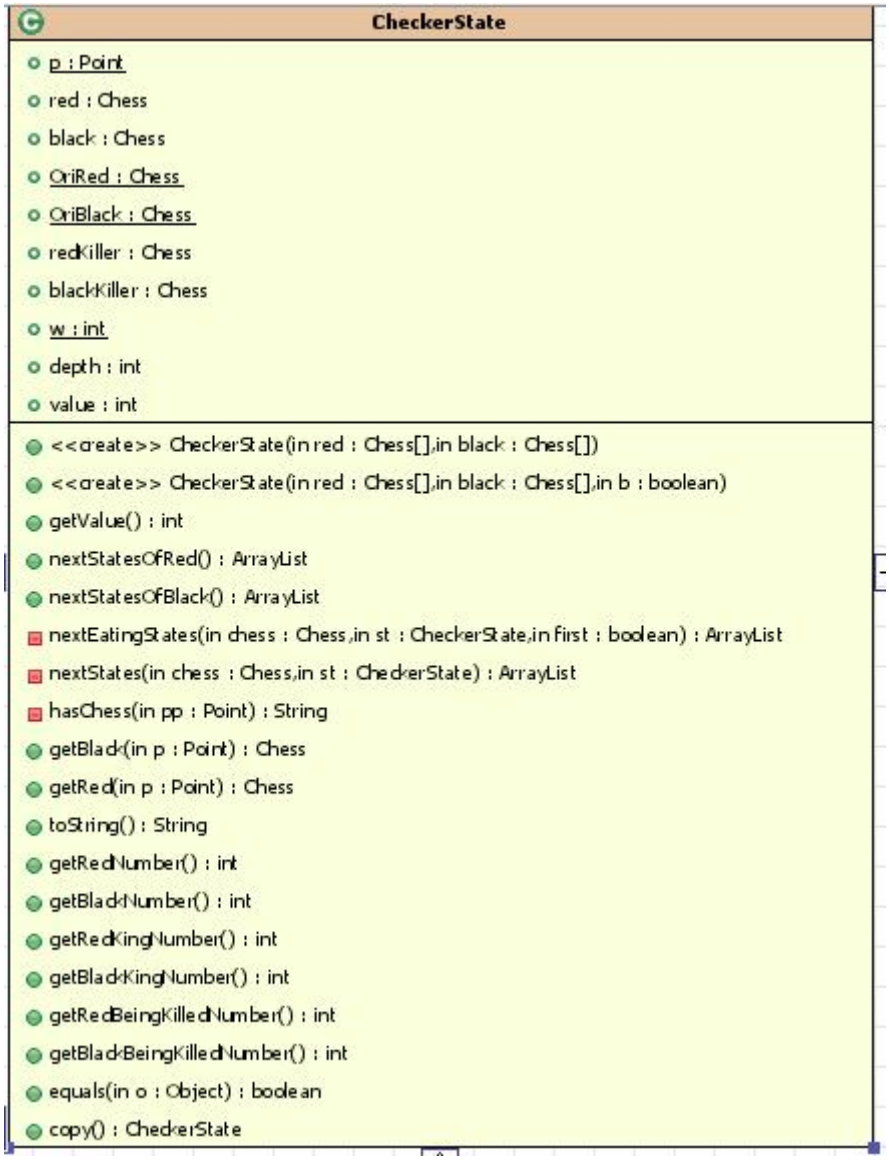


3) ChessBoard

Class ChessBoard stores the attribute of the chessboard, including the images used, the location of each position. Method paint() is used to paint the chessboard and the chesses. It implements mouse listener so that it could deal with mouse pressing operations. Method ifCanGo() judges whether a move is valid.

4) CheckerState

Class CheckerState is designed to store game states, including chesses on board, evaluated value, search depth and so on. The minimax algorithm searches such states to find a best move.



5) Robot

Class Robot is the game player. It uses minimax algorithm and alpha-beta pruning to find a best move.

Method findPath() is used to find the path from the current state to next move.

6) Util

Class Util contains many tool functions, including method to play sound and method to judge if one chess can eat another.

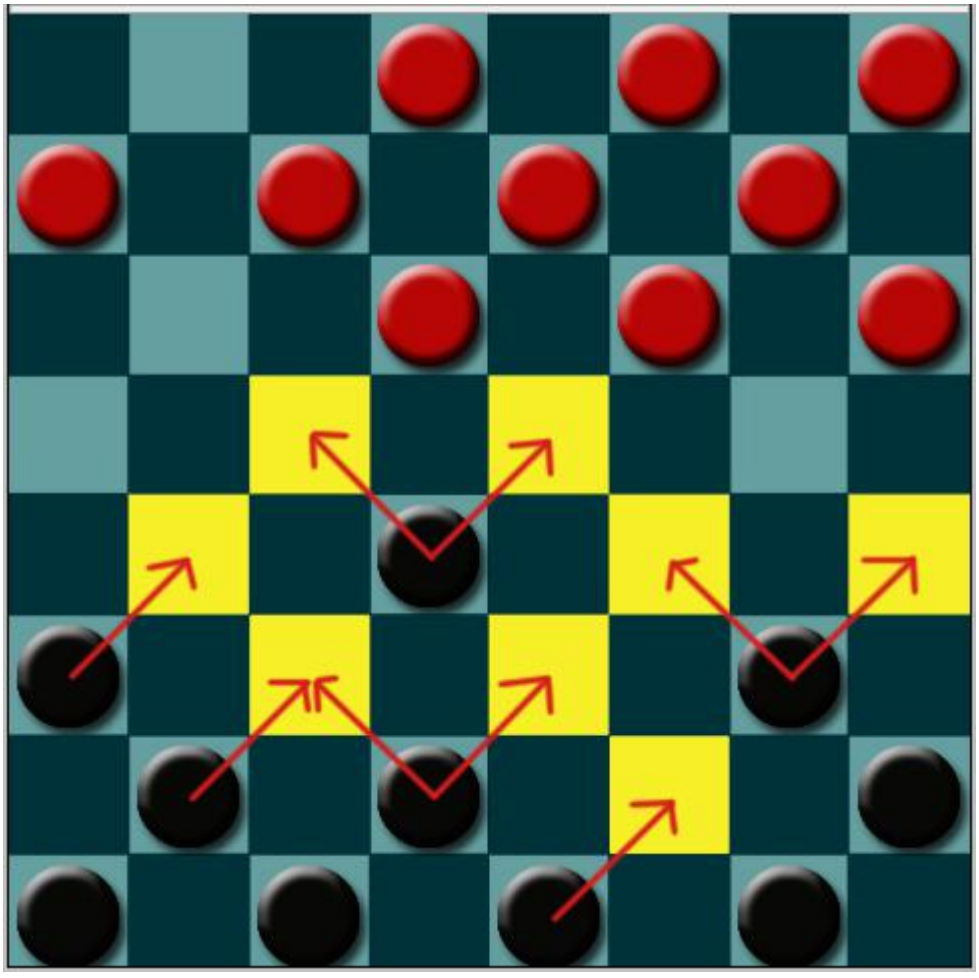
3. Algorithms

1) Finding states

First of all, I need to find out all possible states of both sides. That is to check the valid moves for each chess. This is showed in the picture below. For the black side, there are nine possible moves. Then I need to create eight new instances of Class CheckerState and store them in a list.

If one chess can kill the other, according to the rule, it must kill. So the valid moves are just the killing moves. And also, if after killing one chess, it can still kill others, it must continue to kill. This should be noticed when searching for states.

Method nextEatingStates searches for eating states only. If it returns an empty list, then call nextStates to find other states.



2) Evaluate function

The evaluate function is quite easy. In fact, if you want to design a strong AI player, this is the most import part. I ever saw an evaluate function with more than 4,000 line codes. As I never played checkers before, I don't have a good comprehension on the game states. So it's hard for me to design a good evaluate function.

Our evaluate function is a linear function with seven parameters. I need to extract seven features out of the game state. They are

a. Red chess number on board
b. Black chess number on board.
c. Red king chess number on board.
d. Black king chess number on board.
e. Red chess being killed number
f. Black chess being killed number.

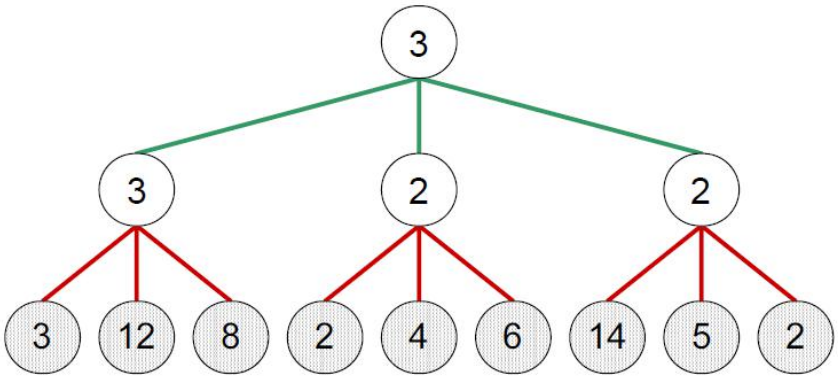
The seven parameters are stored in array w, and the final evaluate value is

Value = w [0] + w [1] * a + w [2] * b + w [3] * c + w [4] * d + w [5] * e + w [6] * f

The parameters are set manually; I tried many times to get a nice result.

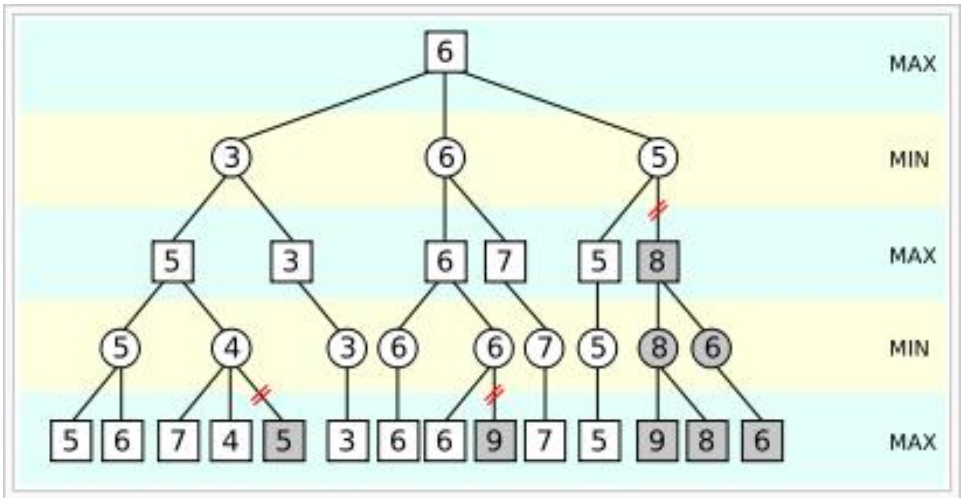
3) The minimax algorithm with alpha-beta pruning

Minimax is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the maximum possible loss. Alternatively, it can be thought of as maximizing the minimum gain. It started from two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves. It has also been extended to more complex games and to general decision making in the presence of uncertainty.



When the search depth is 6, the speed becomes unacceptable. So I need to implement alpha-beta pruning algorithm.

Alpha-beta pruning is a search algorithm which seeks to reduce the number of nodes that are evaluated in the search tree by the minimax algorithm. It is a search with adversary algorithm used commonly for machine playing of two-player games. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. Alpha-beta pruning is a sound optimization in that it does not change the result of the algorithm it optimizes.



After implementing alpha-beta algorithm, the search depth could reach 9 even 10, it's a great progress.

4. Problems

Although it's hard for me to beat this program, there are still a lot of problems.

First of all, I wonder whether this AI player is strong. I did not ever played checkers before, it means nothing that I could not beat it. if I could find some strong human players to test the program, that would be better.

Secondly, I did not use machine learning strategies. This is really a big problem. In fact, I did ever decide to add this. As the time is so limited and I still have other things to do, finally I failed. But if I have time later, I would add it to see if there will be some improvements.

As mentioned before, the evaluate function is too easy. Maybe I should learn to play checkers first and then I could design a good function.

Another problem is that the AI player is good at defense, but bad at attack. Sometimes it just moves back and forth. This is really frustrated. I think this is because the search depth is not deep enough, and also, the evaluate function is too easy.