Checkers AI Program

Jongoh (Andy) Jeong / November 2, 2019

==========

Description

Implementation of Checkers game with artificial intelligence using iterative deepening and alpha-beta search algorithm (minimax search with alpha-beta pruning) , along with a heuristic function.

Environment

This was tested on `Eclipse Oxygen (for C/C++) 2018` version as well as with `g++ compiler (on Ubuntu 16.04.11 with kernel version 5.4.0)` on a Unix-based terminal.

Functionalities

1. User-defined interaction: the user can interactively specify details such as `which player(s) is(are) computer(s)`/`time limit for search (3-60 sec)`/`read board from external file`. This allows the user to play against oneself or computer, improve AI performance by deeper search, and resume from previous point in a game.
2. Colored display: the board is structured by line segments and colored foreground/background texts using ANSI escape codes (see the sample board).
3. Recursive jumps: as part of Checkers game, multiple consecutive jumps are permitted, and this was recursively searched and actively marked till the end of jumps. Each jump is conditioned by the following criteria:

    1. check for valid positions at jumping position and ending position
    2. check if the jumping position and ending position is not empty
    3. check if the jumping position has the opposite color (to take over)

4. Search depth limit: the maximum number of search depth is preset to a value (e.g. 25) so that the program does not spend exceedingly greater search time. Time limit also set by the user at the start of the game.
5. Heuristic function: the evaluation of the state is performed with the heuristic function, as discussed below in `search algorithm` section.

Run Procedure

```
cd {root_directory}
make && ./game.exe
```

> **Makefile**

```
CXX=g++CXX=g++
CFLAGS = -std=c++0x -c -ggdb
DEPS = board.h game.h

%.o: %.cpp $(DEPS)
	$(CXX) $(CFLAGS) -o $@ $<

game.exe: board.o game.o
	$(CXX) -o game.exe board.o game.o

debug:
	$(CXX) $(CFLAGS) -g gameDebug.exe board.cpp game.cpp

clean:
	rm *.exe *.o *.stackdump *~
```

> **Available test boards: [1-4].txt (see format below)**

```
-0-0-0-0        // each row is 8-character long line
3-0-0-0-        // '-': invalid space
-4-2-3-0        // '0': empty space
0-0-0-0-        // '1': Player 1 Pawn
-4-2-4-0        // '2': Player 2 Pawn
0-0-0-0-        // '3': Player 1 King
-1-4-2-0        // '4': Player 2 King
0-0-3-0-
1 // which player to start
10 // time limit (in seconds)
```

Sample run

```
Player 1 -- Pawn: o / King: %
Player 2 -- Pawn: * / King: @
Player 1's turn

Available moves:
Move 1: B3 --> A2
Move 2: B3 --> C4
Move 3: D3 --> C2
Move 4: D3 --> E4
Move 5: F3 --> E4
Move 6: F3 --> G2
Move 7: H3 --> G2

AI is making a move...
Completed search to depth 8.
Ran out of time searching to depth 9.
```

```
Search duration: 1 seconds
Move done: F3 -> E4
```

Search algorithm

The search algorithm is implemented with a minimax search with alpha-beta pruning, and iterative deepening, along wiht a heuristic function. In each depth-limited search, the optimal path in the tree is recursively fed in, to determine the search path for the next depth limited search, allowing early alpha-beta cutoffs to reach deeper levels in a time limited iterative deepening search.

Heuristic function

> **Weights**

- a1: P2 pawn/king
- a2: P1 pawn/king
- b: distance to becoming a king
- c: P2 pawn/king vs. P1 pawn/king
- d: corner diagonal statistic
- e: random variance (negative for P1)

The evaluation of the ongoing state is determined by several weights that contribute to the overall score, or the heuristic for the search. These statistics are arbitrarily chosen (values) by how much each contributes to measuring strength of each player, all of which are summed at the end. The simple descriptions are mentioned in the above bullet points. Each statistic value is arbitrarily chosen to either reward or penalize by the player over the entire game space (8x8 grid), and then later multiplied by a constant value to scale. Notice that d value is determined by whether a piece of certain status (pawn/king, player1/player2) is placed near the two leftmost/rightmost columns, and e is generated randomly (scaled).