Homework #2 – Checkers

DD2380: Artificial Intelligence

Contents

1 Introduction **1**
2 The game **2**
2.1 Board and pieces 2
2.2 Valid moves 3
2.3 End of game 3
3 Assignment **3**
4 Provided code **4**
4.1 C++ 5
4.2 Java 6
4.3 Python 7
5 Check your solution and grade online **8**
6 Submit your solution **9**
6.1 C++ notes 10
6.2 Java notes 10
6.3 Python notes 10
7 Questions **11**
Appendices **12**
A The protocol **12**

1 Introduction

In this **individual** assignment, you must implement a program that plays the game of checkers (also known as English draughts). Your goal is to implement a strategy that allows your program to win (or not lose) as often as possible, not to write a program that makes valid moves, since a skeleton that does just that is already provided for you (see section 4). You will also need to **briefly** answer a couple of questions (see section 7).

> **Warning:** Read all the instructions carefully, since failing to follow them might lead to your submission being rejected

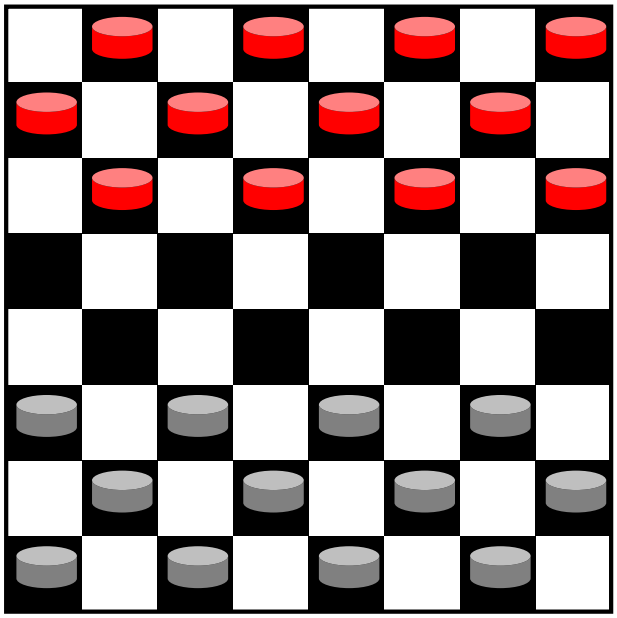

Figure 1: Initial board layout

2 The game

The game of checkers is played by two players, on opposite sides of a squared board. Each player has twelve pieces, which can move diagonally. The goal of the game is to leave your opponent without any movement possibility. This is achieved usually by capturing all of the opponents pieces by jumping over them.

2.1 Board and pieces

The board is square, containing 8x8 square cells. Cells are alternatively colored black and white, with the lower left corner colored black. Cells connected by a corner are the same color.

For each player, the row that is closer to them will be considered the row 1, and the row that is further from them will called the row 8 or *last* row. *Next* row will refer, for each row, to the row that has the lowest higher number, and *previous* row will be the one that has the highest lower number. For example, with respect to the row 4, the *next* row will be the row 5, and *previous* row will be row 3.

Red (or black) pieces are placed in one player's first three rows, and white pieces will be placed in the other player's first three rows (see Figure 1). Each player will control the pieces of the color which are initially closer to them, which will be called *their* pieces.

The initial board layout can be seen in figure 1

There are two kinds of pieces: normal pieces, and kings. Initially, all pieces are normal, but any piece that reaches the last row will become a king.

2.2 Valid moves

The players move in turns, alternatively.

In their own turn, a player can move one of their own pieces. All movements are performed diagonally, which means that since all pieces start in black squares, all pieces will always be in black squares only.

There are two types of moves:

Normal move Pieces can be moved diagonally forward, to an adjacent empty square in the next row. Kings can also move backward, to an empty square in the previous row.

Jumping If the adjacent square in the next row is occupied by an opponent's piece, and the square immediately and directly opposite that square is empty, the piece can "jump" over the opponent's piece. Normal pieces can only jump forward, while kings can also jump backward. The piece that is jumped over is captured and removed from the board. It is possible to perform multiple jumps in the same turn, if when the piece lands, there is another immediate piece that can be jumped, even if the jump is in a different direction.

Jumping is mandatory: whenever it is possible to perform a jump, it is not possible to perform a normal move. When more than one jump or multiple jump is available, the player can choose which piece to jump with, and which sequence of jumps to perform. It is not necessary to perform the multiple jump that captures the most pieces. However, it is mandatory to perform all the jumps in the chosen sequence: the moved piece cannot end in a position where another jump would be possible.

If a piece moves into the last row, that piece is "crowned" and becomes a king. The piece that becomes a king after a jump, cannot immediately jump backward over another piece.

2.3 End of game

A player wins by capturing all the opponent's pieces, or by leaving the opponent with no valid moves. If no pieces are captured for 50 turns (25 of each player), the game is considered a draw.

You can find more information in the English Wikipedia (`http://en.wikipedia.org/wiki/Checkers`[1]), where most of the description provided here comes from.

3 Assignment

Your assignment consists of writing a program that plays checkers as one of the players.

[1] In the last days there has been changes back and forth in Wikipedia about a King being able to move more than one square per turn. No matter what the actual version in Wikipedia says, such movement is not allowed in our version

However, you are not going to start from scratch. You will receive a skeleton in your language of choice, which will take care of communication with the server, board representation, and calculating the possible moves from each board position. This way, you will only have to focus on implementing the strategy for deciding which is the best move to make, where the core AI lies for this problem.

This is the sequence of events for a game (again, most of this will be handled by the provided skeletons. It is here for informational purposes, so that you know what's going on):

1. Each program will connect to the server, which will inform the players of who is going to move first (let's call it `Player 1`), and give both players **20 seconds to perform initialization** (in case they need it).
2. The server tells Player 1 that it has **10 seconds to perform the movement**.
3. After Player 1 sends its movement to the server, Player 2 is told by the server that it has 10 seconds to answer back with its next move.
4. Repeat steps 2 and 3 until a movement results in a victory or a draw
5. The server then informs both players that the game has ended, and tells them who won.

The server, when communicating moves to the players, will always use the cell numbering relative to the player it is talking to. This way, for you, **the lower right black cell of the board will always be cell 0, as seen in 2, independently of who moves first**.

The server also has a standalone mode, where you play against a really simple player built into the server, so that you have an easy way to debug your code without having to play against other clients.

Your program must take three arguments from the command line. The first two are the host and port of the server (which for testing purposes will be 130.237.218.85 and 5555). The third one is a key to pair your program against another one. If the key is omitted, your program will connect to the server in the just-described standalone mode, and this is what you should use for testing.

When you receive them, the skeletons will already by able to play against another player, but they will choose the moves at random, which is not a very good strategy. This is what you have to do: modify the code that chooses the next move so that it wins as much as possible.

4 Provided code

A basic program is provided for you in each of the supported programming languages. The program, as it is provided, is fully functional, but plays a random legal move each turn. You **must** implement a better strategy. The programs (also called skeletons) are

written in such a way that they are easy to understand, which is not always the most efficient. Their use is not compulsory: you can modify them in any way you want or just write your own, as soon as you are compliant with the protocol (see Section A).

Skeletons provide all the code for communication with the server, as well as classes for representing the board and obtaining the possible moves in a given position.

4.1 C++

The documentation for each class is available in each header file in the doxygen format [2].

To implement your solution, you must modify the class **CPlayer** (in files cplayer.h and cplayer.cc). There are three functions which you can modify as a starting point to add your own functionality. Of course, you can, and probably should, add your own functions to this class, or create additional classes.

Play(const CBoard &pBoard,const CTime &pDue) tells the player to perform a move. pBoard contains the current board position, and pDue is the time by when the function must return. You must return a CMove object from this function, containing the move you are playing. *This is your main task; you must change this function*

Initialize(bool pFirst, const CTime &pDue) tells the player to start initialization. pFirst will be true if the player will be the first one to move, and pDue is the time by when the initialization must be complete. *You don't need to change this function, but you can improve your performance by doing so.*

Idle(const CBoard &pBoard) this function will be called by the client while the other player is performing a move. You can perform some extra "thinking" in this function, but you should return from time to time, so that the client can check if the other player has already moved. If it has not, then this function will be called again, unless you return false. In this case, the function will not be called again until the next move. *You don't need to change this function, but you can improve your performance by doing so.*

There are three main helper classes that you need to care about. You will probably find useful to use some of their public functions (so please read their interfaces), but that you don't need to change their code:

CBoard represents a board position, that is, the position of all the pieces in the board. It has functions for checking what is in each square, for obtaining the possible moves from that position, and for transforming the board by performing a move. *You don't need to change this class.*

[2] `http://en.wikipedia.org/wiki/Doxygen`

CMove represents a possible move. It is returned by the board class when querying the possible moves, and then, you can pass it back to the board to transform it by making this move. Most of the time, this is the only way you will use this class, and you don't need to care about its interface. *You don't need to change this class.*

CTime represents a time. It can be used to know the time remaining to perform a move. *You don't need to change this class.*

CBoard contains methods for printing named Print() and PrintNoColor(). By default PrintNoColor() is used, since it is more compatible to different terminals[3].

4.2 Java

The documentation for each class is available in each class file in the doxygen format [4].

To implement your solution, you must modify the class **Player**. There are three functions which you can modify as a starting point to add your own functionality. Of course, you can, and probably should, add your own functions to this class, or create additional classes.

Play(Board pBoard,Date pDue) tells the player to perform a move. pBoard contains the current board position, and pDue is the time by when the function must return. You must return a CMove object from this function, containing the move you are playing. *This is your main task; you must to change this function*

Initialize(boolean pFirst, Date pDue) tells the player to start initialization. pFirst will be true if the player will be the first one to move, and pDue is the time by when the initialization must be complete. *You don't need to change this function, but you can improve your performance by doing so.*

Idle(Board pBoard) this function will be called by the client while the other player is performing a move. You can perform some extra "thinking" in this function, but you should return from time to time, so that the client can check if the other player has already moved. If it has not, then this function will be called again, unless you return false. In this case, the function will not be called again until the next move. *You don't need to change this function, but you can improve your performance by doing so.*

There are two main helper classes that you need to care about. You will probably find useful to use some of their public functions (so please read their interfaces), but that you don't need to change their code:

Board represents a board position, that is, the position of all the pieces in the board. It has functions for checking what is in each square, for obtaining the possible moves from that position, and for transforming the board by performing a move. *You don't need to change this class.*

[3] Print() requires a terminal which supports ANSI colors

[4] `http://en.wikipedia.org/wiki/Doxygen`

Move represents a possible move. It is returned by the board class when querying the possible moves, and then, you can pass it back to the board to transform it by making this move. Most of the time, this is the only way you will use this class, and you don't need to care about its interface. *You don't need to change this class.*

Board contains methods for printing named Print() and PrintNoColor(). By default PrintNoColor() is used, since it is more compatible to different terminals[5].

4.3 Python

The documentation for each class is available in comments inside each class file.

To implement your solution, you must modify the class **CPlayer** in cplayer.py. There are three functions which you can modify as a starting point to add your own functionality. Of course, you can, and probably should, add your own functions to this class, or create additional classes.

play(pBoard,pDue) tells the player to perform a move. pBoard contains the current board position, and pDue is the time by when the function must return. You must return a CMove object from this function, containing the move you are playing. *This is your main task; you must to change this function*

initialize(pFirst,pDue) tells the player to start initialization. pFirst will be true if the player will be the first one to move, and pDue is the time by when the initialization must be complete. *You don't need to change this function, but you can improve your performance by doing so.*

idle(pBoard) this function will be called by the client while the other player is performing a move. You can perform some extra "thinking" in this function, but you should return from time to time, so that the client can check if the other player has already moved. If it has not, then this function will be called again, unless you return false. In this case, the function will not be called again until the next move. *You don't need to change this function, but you can improve your performance by doing so.*

There are two main helper classes that you need to care about. You will probably find useful to use some of their public functions (so please read their interfaces), but that you don't need to change their code:

CBoard represents a board position, that is, the position of all the pieces in the board. It has functions for checking what is in each square, for obtaining the possible moves from that position, and for transforming the board by performing a move. *You don't need to change this class.*

[5] Print() requires a terminal which supports ANSI colors

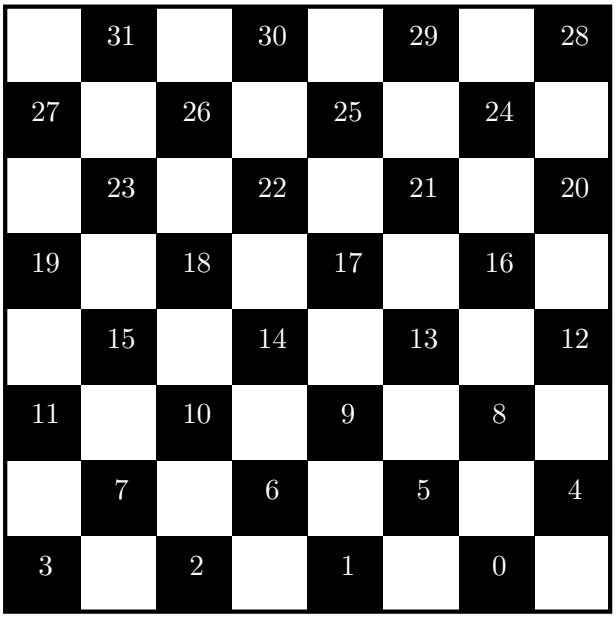

Figure 2: Cell numbering

CMove represents a possible move. It is returned by the board class when querying the possible moves, and then, you can pass it back to the board to transform it by making this move. Most of the time, this is the only way you will use this class, and you don't need to care about its interface. *You don't need to change this class.*

CBoard contains methods for printing named print_out() and print_nocolor(). By default print_nocolor() is used, since it is more compatible to different terminals[6].

5 Check your solution and grade online

To see how well your checker program does against our bots and against other people in the class, visit:

`http://130.237.218.85/~marin/hw2check/`

You will have to submit your solution as described in Section 6. Your grade will depend on how your program does against our bots:

- Beating `Mr. Dumb` – **8 points**
- Beating `Mr. OK` – **5 points**
- Beating `Mr. Smart` – **5 points**

[6] print_out() requires a terminal which supports ANSI colors

2 extra points will be awarded for providing a correct answer to the questions, so the maximum score for this assignment will be 20 points.

6 Submit your solution

You need to submit a `zip` archive *usern_lang*`_hw2.zip`, where *usern* is your KTH username and *lang* is you programming language of choice (either `cpp`,`java` or `py`). For example, `johang_cpp_hw2.zip`.

The server will compile and run your solution under Ubuntu 11.04. Therefore your solution must compile and run in such a system. The skeletons provided do fulfill these requirements. Moreover, they should work in Windows and MacOSX as well. If you work under an environment different from Ubuntu, you should just avoid adding OS specific instructions in your code.

The contents of the `zip` archive should be:

- `rulecode.txt` – This file should contain your *rule list code*, nothing else. You will find your rule list code on the first page of this document, near the document title.
- `email.txt` – This file should contain your KTH email, nothing else.
- `questions.txt` – This file should contain the answer to the questions in section 7 (no more than 300 words, length automatically checked).

If you submit your solution in C++, one or more .cc and zero or more .h files. All .cc files will be compiled and linked together to generate the program.

If you submit your solution in java, a Client.java file containing the main function, and more .java files containing auxiliary classes.

If you submit your solution in python, a main.py file that will be the program to run, and optionally more .py support files.

Attention:

Do not put **any other files** in your `zip` file.
Do not put **any subfolders** in your `zip` file.
Do not make your `zip` file bigger than **5 MB**.
Use the online form to verify the `zip` file.

6.1 C++ notes

The only supported libraries are the libraries which Linux links against by default[7]. Your solution cannot depend on third party libraries to run.

Here is a description of the system we will use to test your program: Our evaluation system will run Ubuntu Linux. We will use the `g++` compiler version `4.5` from the GCC compiler suite to compile your solution.

- The compilation commands will be:

```
g++ -o client *.cc
```

- The program will be executed as:

```
./client host port key
```

6.2 Java notes

You can program your solution on Windows, MacOS X or Unix using a reasonably recent version of Java.

Your solution should be self-contained and cannot depend on external `java` files or third party libraries to run. The only supported library is the standard Java library. Windows or MacOS specific libraries or Java extensions are not allowed.

Here is a description of the system we will use to test your program: Our evaluation system will run Ubuntu Linux. We will use the official Oracle's `javac` compiler for Java 6.

- The compilation commands will be:

```
javac *.java
```

- and it will be executed as follows

```
java Client host port key
```

6.3 Python notes

You can program your solution on Windows, MacOS X or Unix using a resonably recent version of Python 2 (e.g. 2.7) or 3. The skeleton we provide is compatible with (at least) versions 2.7 and 3.2 of Python. Evaluation will be done using Python 3, so if you decide to do your development in Python 2 you must be careful not to use any construction that is Python 2-specific[8]

You cannot depend on libraries outside the standard Python distribution and you can only provide .py files, which must all lie in the same folder.

[7] standard C++ library and the POSIX-compatible C library in Linux

[8] There are tools to convert your python2 code to python3: `http://docs.python.org/library/2to3.html`

Here is a description of the system we will use to test your program: We will use Python version `3.2` to run your solution. Our evaluation system will run Ubuntu Linux.

- Your code will be executed as:

```
python3 main.py host port key
```

7 Questions

Answer **briefly** (no more than **300 words** in total) the following two questions:

- What does your method do? (describe briefly the strategy implemented)
- If your method was allowed an infinite amount of time to make each move, would it be impossible to defeat it? (reason your answer)

Appendices

A The protocol

> **Important:** If you use the provided skeletons, you don't even have to look at this

This appendix describes the protocol used for communication between your client and the server.

It is a text and line-based protocol, i.e. each message consists of a single line of text. Lines are terminated with an LF ('\n') character. The server will also accept a CR-LF pair, but it will not return it.

Each message must be sent as it appears here, with no leading or trailing whitespace, and appending the LF character to them.

After connecting to the server, your client sends one of the two following messages:

- MODE STANDALONE
- MODE KEY *key*

The first one connects to the server in standalone, and the second one is used for pairing. The *key* must be the one provided to your program in the command line.

The server will reply with:

- *time first*

where *time* is the time before which you must have performed the initialization (in microseconds since the Epoch), and *first* is an integer which will be non-zero if you will move first, zero otherwise.

After you have performed the initialization, you must send:

- INIT

Then the game itself starts. All communication from here on are move exchanges. The server will send a move command before every one of your moves, informing you of the move of your opponent. If you play first, you will also receive a move command with a special code before this move.

A move is encoded as a series of integers.

The first integer will take the value -2 to indicate beginning of game, -1 to indicate end of game, 0 to indicate a normal move, and 1 or more to indicate 1 or more jumps.

If the first integer is -2 or -1, no more integers will follow. If it is 0, the series will include two more integers, indicating the source and destination of the piece that moves. If the number indicates n jumps, n+1 numbers will follow, indicating the source and each of the points the piece jumps to.

These positions will be encoded using the numbers seen in 2, which will always be relative to you: 0 will always be the square on your lower right.

Some example:

- -2
- 0 8 13
- 2 6 15 22

When the server sends a move, it will precede it with another integer, which will indicate the time before which you must make the next move.

When the server decides that the game is finished (because either player has won, or the draw rule kicks in), it will send an end of game move to both players.

This is an example exchange of messages:

client MODE STANDALONE

server 123456789 1

client INIT

server 234567890 -2

client 0 9 13

server 345678901 0 22 17

client 1 13 22

server 456789012 1 25 18

client 0 8 12

...

server 901234567 -1

After sending the end of game command, the server will immediately close the connection