ECE 12: Assignment 7

(two week assignment)

March 2, 2004

Due Friday, 3/19/2004, at 12:00pm

1 The Game of Chess [80 points]

This is the final assignment for this class. As such, it is designed to be an interesting exercise where you can practice all elements of programming learned in the class so far, and more!

This exercise consists of two parts. The first part is the basic requirement which, if you complete it correctly, will earn you the 80 credits available for this two-week assignment. The second part is a challenge in which you can earn up to 50 extra credits if you master it in some smart way.

The goal of this programming exercise is to develop a chess program in which an interactive user can play chess against the computer.

Of course, this is a big task, but it is not as difficult as it may sound. We will simplify the rules of the chess game by eliminating some special and complex moves. We will also structure the programming task in an organized manner and provide some initial program code as a starting point.

Let's go step by step!

1.1 Step 1: The rules of chess

As a first step, you should make yourself familiar with the rules of the chess game. The rules of chess are found online on the web pages of the US Chess Federation in the Beginners Area at:

`http://www.uschess.org/beginners/letsplay.php`

In case the official website is not accessable, a copy of this page is available on the web pages for this course.

Using the information from this web site, learn the rules of chess. Make yourself familiar with the goal of the game, the game play and the chess board. Then, learn about the six different pieces and how they can move on the board.

For this exercise, we will consider only the basic moves, as follows:

- The queen can move horizontally, vertically, and diagonally across the board.
- A rook can move horizontally and vertically across the board.
- A bishop can move diagonally across the board.
- A knight can jump to eight different squares which are two steps forward plus one step sideways from its current position.
- The king can move in any direction, but only one step at a time. Also, the king must never move into check. (We will ignore the special "castling" move for the king.)
- A pawn can move only forward towards the end of the board, but captures sideways. From its initial position, a pawn may make two steps, otherwise only a single step at a time. If the pawn reaches the end of the board, it is automatically promoted to a queen. (We will ignore promotion to any other piece, as well as the special "en passant" move for the pawn.)

The chess game ends as soon as one king is trapped in "checkmate". That is, there is no move for the player possible which would get his king out of check. So, the player loses.

1.2 Step 2: Setup the program environment

In order to get you started with the chess program, a set of files with chess program conventions have been prepared. These files are stored in a `gzip`-compressed `tar` archive named `Chess_PKG.tar.gz` in the directory `~ece12/pychess/`.

You can copy the package into your directory for `hw7` and extract the contents as follows:

```
% cp ~ece12/pychess/Chess_PKG.tar.gz .
% gtar xvzf Chess_PKG.tar.gz
```

After that you should have the following Python files in your `hw7` directory:

- `chess.py`:
  a Python module with chess conventions: this module contains definitions of chess players, chess pieces, the chess board (`class ChessBoard`) and the chess game (`class ChessGame`) for logging the moves of a game; this module is imported by all the other files
- `game.py`:
  the main chess program: this is the main program to run for playing the chess game; in order to play chess, type `python game.py` at the shell prompt
- `interactive_player.py`:
  this module defines an *interactive* (human) player; that is, the user is asked to enter the moves to play; this module is imported by `game.py` to play the white pieces
- `random_player.py`:
  this module defines a *dumb* computer player which always makes random moves; however, this player obeys the rules of the game by making only valid moves; this module is imported by `game.py` to play the black pieces
- `smart_player.py`:
  this module is a template file for a *smart* computer player; that is, the contents of this file are to be filled by you!
- `move1.pyc` and `move2.pyc`:
  these are binary Python modules used by the `random_player` and `game` modules; the contents of these files are internal, so don't worry about them

1.3 Step 3: Run the chess program

OK, you have the files, let's play chess!

Type `python game.py` at the shell prompt. You will be presented with an illustrated chess board and are asked to make your move. For example, you may type `e2e4` to move the white pawn forward by two steps. The computer will check your move and make it, if it is valid. It will then print the updated chess board on the screen.

Next, it is the turn for the black player which is setup as the *dumb* random player. Here, the computer will compute all possible moves and randomly select one. Then, the board is updated and the white player can make his next move.

The scenario described above looks as follows:

```
% python game.py

The chess game begins!
          +----+----+----+----+----+----+----+----+
        8 | bR | bN | bB | bQ | bK | bB | bN | bR |
          +----+----+----+----+----+----+----+----+
        7 | bP | bP | bP | bP | bP | bP | bP | bP |
          +----+----+----+----+----+----+----+----+
        6 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        5 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
```

```
        4 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        3 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        2 | wP | wP | wP | wP | wP | wP | wP | wP |
          +----+----+----+----+----+----+----+----+
        1 | wR | wN | wB | wQ | wK | wB | wN | wR |
          +----+----+----+----+----+----+----+----+
            a    b    c    d    e    f    g    h

Move  1, White:
Player White, please enter your move: e2e4
White wants to move e2e4 ... OK.
          +----+----+----+----+----+----+----+----+
        8 | bR | bN | bB | bQ | bK | bB | bN | bR |
          +----+----+----+----+----+----+----+----+
        7 | bP | bP | bP | bP | bP | bP | bP | bP |
          +----+----+----+----+----+----+----+----+
        6 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        5 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        4 |    |    |    |    | wP |    |    |    |
          +----+----+----+----+----+----+----+----+
        3 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        2 | wP | wP | wP | wP |    | wP | wP | wP |
          +----+----+----+----+----+----+----+----+
        1 | wR | wN | wB | wQ | wK | wB | wN | wR |
          +----+----+----+----+----+----+----+----+
            a    b    c    d    e    f    g    h

Move  1, Black:
Black wants to move a7a5 ... OK.
          +----+----+----+----+----+----+----+----+
        8 | bR | bN | bB | bQ | bK | bB | bN | bR |
          +----+----+----+----+----+----+----+----+
        7 |    | bP | bP | bP | bP | bP | bP | bP |
          +----+----+----+----+----+----+----+----+
        6 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        5 | bP |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        4 |    |    |    |    | wP |    |    |    |
          +----+----+----+----+----+----+----+----+
        3 |    |    |    |    |    |    |    |    |
          +----+----+----+----+----+----+----+----+
        2 | wP | wP | wP | wP |    | wP | wP | wP |
          +----+----+----+----+----+----+----+----+
        1 | wR | wN | wB | wQ | wK | wB | wN | wR |
          +----+----+----+----+----+----+----+----+
            a    b    c    d    e    f    g    h

Move  2, White:
Player White, please enter your move:
```

1.4 Step 4: Analyze the chess program

Now that you know how the program runs, let's analyze the program so that you can then start improving it (after all, we're not only here for the fun, aren't we?).

1.4.1 File chess.py

First, take a look at the file `chess.py`. In here, you will find many definitions, functions and classes which help you manage the chess game. There are conventions for naming the players and chess pieces.

Then, there is a class `ChessBoard` which represents a chess board on which pieces are placed. The class provides methods for creating and setting up a chess board, as well as for printing it on the screen.

There are also methods for handling the positions (coordinates in terms of rows and columns) and square codes (coordinates in terms of letters and digits). For example, the white king is initially placed in row 7, column 4, which is also known as square code "e1".

Of course, methods for moving pieces on the board are provided as well. Note that there is an internal method `MovePiece` which makes a move without much checking, so this one should not be used by you. Instead, use one of the other two methods which both support the promotion of a pawn to a queen if he reaches the end of the board. So, use `MakeMove` if you work with coordinates as (row,column) tuples, or `Move` if you prefer square codes.

Finally, there is a class `ChessGame` that provides methods for logging the moves of a game. You don't need to worry about this one.

1.4.2 File game.py

Now, take a look at the module `game.py`. This is the top-level program that actually runs the chess game. The file starts by importing the `chess` conventions and the programs representing the two chess players.

Initially, the two players are set up as the `interactive_player` to play white and the `random_player` to play black. However, this can easily be changed to a different setup. You may try to let two `random_player`'s play against each other. Please, however, don't do this too often as it will keep the computer really busy imitiating two dumb players, and therefore will slow down programs of other users on the same machine.

Eventually, you will need to import your `smart_player` here to play against yourself as the human player (`interactive_player`).

Next, the file `game.py` defines some settings for the chess program (which you may adjust to your taste). The `max_moves` variable determines the maximum number of moves for each player before the game will be terminated in a tie. The `verbose` variable indicates whether the chess players may print their thoughts to the screen, or if they are to be quiet. A value of 0 will keep them quiet, any value greater will make them more and more verbose. It is a good idea to increase this value temporarily when you are debugging your code.

Then, two functions are defined that specify how the chess game is played. It should be pretty obvious from the code what is going on here. Note, however, that every move of each player is checked if it is actually a valid move. If so, it is allowed. Otherwise, the game aborts with an exception that indicates why the intended move is not valid.

1.4.3 File interactive_player.py

Next, take a look at the file `interactive_player.py`. This file contains the program for a human player who enters his moves when prompted. The file first imports the chess conventions and then defines the function `ComputeMove` which will be called by `game.py` when it is up to this player to make his turn.

The function `ComputeMove` will be called with three arguments. The first argument (`board`) represents the chess board with the pieces at their current position. The second argument (`player`) indicates which pieces (the white or black ones) the player is supposed to move. Finally, the third argument (`verbose`) indicates whether the player is allowed to print something on the screen.

The function `ComputeMove` must return the move that the player wants to make. The move must be a string indicating the source and target square of the move. For example, `"e2e4"` would indicate to move the piece on square `"e2"` to the square `"e4"`. If a piece of the opponent is on the target square, the piece is captured and removed from the board.

1.4.4 File random_player.py

The file `random_player.py` contains the same functionality as the file `interactive_player.py`, except that the computer is deciding the move to make automatically. So, the function `ComputeMove` receives the same arguments as before, but the return value is computed by some algorithm.

Note that the algorithm for the `random_player` is separated into two major steps. First, the computer enumerates all possible moves that would be legal for him to make. Then, he makes a decision by choosing one of these moves.

The first step of the algorithm is implemented in the function `ComputeAllLegalMoves`. By examining the board that is passed as an argument, this function computes a list of all moves that would be legal for the player to make at this point. This list of moves (encoded as square codes) is then returned as the result of this function.

In the second step of the algorithm, the player then has to choose one of the possible moves. Obviously, he should choose a move that would give him an advantage in the chess game so that he can win the game.

The `random_player`, however, is dumb. He does not try to find a good move, he just chooses a random one and in most cases, that will make him lose the game in the end.

1.4.5 File smart_player.py

The file `smart_player.py` is your chance to actually create a computer player that acts *smarter* than the dumb `random_player`. It is up to you to fill in the program here, so that playing against the machine actually becomes more interesting and challenging.

The file provided is just a template. You will need to fill in code wherever four dots (....) are placed as space holders.

1.5 Step 5: Programming strategy

OK, you know your task now, but before you start coding, you should define a programming strategy that lets you write your code in a systematic and organized way. In other words, the big programming task will become much easier to handle, if you have it partitioned into smaller, well-defined subtasks.

For this exercise, we will partition the programming task into two pieces that we have briefly mentioned earlier already. The computation of the next move on the chess board can be divided into two separate subtasks, as follows. First, we will determine the set of moves that are possible and legal to make. Then, we will choose one move out of this set that gives us the greatest advantage over the opponent.

Note that the `random_player` follows the same strategy. He first enumerates all legal moves and then makes his choice. On the first subtask, he does a good job (he will always make a legal move), but for the second job, he performs really badly with his random choice.

The first subtask, finding all legal moves, is the basic requirement of this programming assignment. It will, if implemented correctly and completely, earn you the 80 credits of the assignment. The second subtask, making a *smart* choice, gives you the chance to earn up to 50 extra credits. The better your program performs when competing with other players (the `random_player`, players implemented by other students, or the `interactive_player`), the more credits you will earn. Keep in mind, however, that the first subtask is required for the second one to succeed, because any player who makes invalid or illegal moves will be disqualified from the competition.

In the next two sections, we will give you some hints on tackling the two subtasks. After that, you should be well-prepared to write a good program for the `smart_player`.

1.6 Step 6: Computing all legal moves

The task of computing all legal moves on the chess board can again be sub-divided into smaller subtasks. Here is the idea: there are maximal 16 pieces on the board that belong to the player who needs to make his move. So, we will consider every piece separately, and then just concatenate the lists of legal moves for each piece. This will then result in the list of all legal moves available for the player.

Further, it is very helpful if we distinguish between *possible* and *legal* moves. For every piece, the list of *possible* moves includes all reachable squares. For example, a rook can reach all squares horizontally or vertically, which are not blocked by another piece. This is the list of possible moves.

Now, not all possible moves are also legal. A move is *legal* only if it leaves the players king protected. If, after a move would have been made, the players king is threatened (in ”check”), then the move is illegal and must not be made.

With this distinction of possible and legal moves, it becomes obvious to compute all legal moves. We simply enumerate first all possible moves, and then filter out the ones that are illegal.

1.6.1 Step 6a: Computing all possible moves for each piece

Let’s start by computing the possible moves for each piece. Consider the following chess board as a test case:

```
  +----+----+----+----+----+----+----+----+
8 | bR |    | bB |    | bK | bB |    | bR |
  +----+----+----+----+----+----+----+----+
7 | bP |    | bP |    |    | bP |    |    |
  +----+----+----+----+----+----+----+----+
6 |    | bP |    |    |    | bN |    |    |
  +----+----+----+----+----+----+----+----+
5 |    |    |    | bQ |    |    |    |    |
  +----+----+----+----+----+----+----+----+
4 |    | wR | wB |    | wP |    | wQ |    |
  +----+----+----+----+----+----+----+----+
3 |    |    | bN |    |    | wP |    |    |
  +----+----+----+----+----+----+----+----+
2 |    |    | wP |    |    |    | wP | wP |
  +----+----+----+----+----+----+----+----+
1 |    |    | wB |    | wK |    | wN | wR |
  +----+----+----+----+----+----+----+----+
    a    b    c    d    e    f    g    h
```

For example, the possible moves for the black knight on square ”c3” are indicated in the following figure:

```
  +----+----+----+----+----+----+----+----+
8 | bR |    | bB |    | bK | bB |    | bR |
  +----+----+----+----+----+----+----+----+
7 | bP |    | bP |    |    | bP |    |    |
  +----+----+----+----+----+----+----+----+
6 |    | bP |    |    |    | bN |    |    |
  +----+----+----+----+----+----+----+----+
5 |    | ** |    | bQ |    |    |    |    |
  +----+----+----+----+----+----+----+----+
4 | ** | wR | wB |    |(wP)|    | wQ |    |
  +----+----+----+----+----+----+----+----+
3 |    |    | bN |    |    | wP |    |    |
  +----+----+----+----+----+----+----+----+
2 | ** |    | wP |    | ** |    | wP | wP |
  +----+----+----+----+----+----+----+----+
1 |    | ** | wB | ** | wK |    | wN | wR |
  +----+----+----+----+----+----+----+----+
    a    b    c    d    e    f    g    h
```

The knight can reach 6 squares which are free, as indicated by the two stars (`**`). He can also capture the white pawn on square ”e4” as indicated by the parentheses around the pawn. The knight cannot, however, jump to square ”d5” because this square is occupied by his own queen.

In terms of our chess program in Python, the list of reachable positions for the knight is the following:

```
['b5', 'a4', 'e4', 'a2', 'e2', 'b1', 'd1']
```

Or, in terms of (row,column) tuples:

```
[(3,1), (4,0), (4,4), (6,0), (6,4), (7,1), (7,3)]
```

Now, write an initial Python program that computes these possible positions for the black knight on the test board. Your program should do the following:

1. import the chess module
2. define a function `ReachablePos4Knight(board,source)` that computes the list of reachable positions for a knight at position source (source is a tuple of (row,column)) and returns this as a list of pairs of (row,column)
3. create a chess board
4. setup pieces on the board as indicated in the figure above
5. print the board
6. call your function `ReachablePos4Knight`
7. print the position list on the screen
8. use the method `PrintWithMarkedPos` to print the board with your positions indicated

When you run your program, it should produce the same figure as shown above. Save the output of your program in a file `knight.script` as your first assignment submission.

Next, repeat the same procedure for the other five pieces, namely the king, the queen, the bishop, the rook and the pawn. Write the appropriate function and test it by picking one figure on the test board. Save each program output in a separate script file for submission.

1.6.2 Step 6b: Computing all possible moves on the board

Now, in order to compute all possible moves for a player, you only need to put your functions together. Define a new function `ReachablePos(board, source)` which computes all reachable positions for a figure at the given source position. Of course, this function will call the individual functions defined earlier according to the type of piece found at the source position.

Next, define a function `PossibleMoves(board, player)` which determines all possible moves for the given player. As indicated earlier, this function should check every position on the board and if it finds a piece of the given player, determine the possible positions reachable from there. This way, this function can easily enumerate all possible moves.

1.6.3 Step 6c: Filtering out illegal moves

As defined above, you have to filter out the illegal moves from the list of possible moves. Remember, illegal moves are the ones, which leave the king in ”check” and that’s not OK in chess.

In order to do this, you can simply check every possible move if it is also legal. Otherwise, you remove that move from the list.

To check if a move is legal, you can use the following approach. Make a copy of the current board and make the move in question on that board (the copy is important so that you do not destroy the current board!). Then, find the king of the current player on the resulting board, and check if it can be reached by any of the opponents pieces.

Note that, in order to determine if an opponents piece can attack your king, you can again use your function `ReachablePos` by calling it for the opponents piece. If the resulting position list contains the position of your king, your king is in ”check” and the move is illegal.

Implement this approach in a function `ComputeAllLegalMoves` which first collects all possible moves by calling `PossibleMoves` and then eliminates the illegal moves. The function should return the list of all moves that are possible *and* legal.

Plug in this function (and all others) into the template file `smart_player.py` at the indicated position. Then, modify the file `game.py` so that it uses your `smart_player` instead of the `interactive_player`. Also, you may want to turn on the `verbose` flag in `game.py` (i.e. set it to 2) so that you can see what your player is thinking.

Finally, test your program and see how well it performs against the dumb `random_player`. Create a script of the game called `game.script` for submission.

1.7 Step 7: Choosing the best move

Most probably, your `smart_player` will not be any smarter than the `random_player` at this point... that's exactly the next task to take on!

Creating a *smart* chess program is a great challenge! Many people have worked on this, and many good ideas have been implemented. However, human chess players still tend to outperform even the best super-computers.

It is not expected from this assignment, that you come up with a solution that will be better than anything invented so far. However, there are many ways to make this choice for the "best" move, and it is up to you to make some reasonable effort to make your chess program smarter.

A general idea is to evaluate every legal move by assigning some value to it. The higher the value, the better the move. Then, it is easy to choose the move with the highest value as the "best" move.

The value of a move in general depends on many factors, but even simple observations can make a difference. For example, you may choose to add value to moves which capture an opponents piece. Or, you may add value to moves which put the opponents king into "check". Or, you may subtract value from moves, which leave your own pieces under attack. Or, ... or... or...

Think about it, and come up with some good idea! Implement it and test it! Improve it if you feel you can do even better.

For submission, submit your program code in the file `smart_player.py`. Also, briefly describe your optimization approach in a text file `smart.txt`. Finally, submit a script file in which your `smart_player` wins a chess game against the `random_player`.

2 What to turn in

Use the command

```
% python ~ece12/tools/submit.pyc
```

to turn in your solution to this assignment. Your files should be one level above the `hw7` directory. Except for the template file that you have extended, you don't need to turn in any of the files that you have extracted from the prepared `tar` archive.

You should submit the following files:

- `king.script`
- `queen.script`
- `bishop.script`
- `knight.script`
- `rook.script`
- `pawn.script`
- `game.script`
- `smart_player.py`
- `smart.script` (for extra credit)
- `smart.txt` (for extra credit)