CSC242: Intro to AI
Project 1: Game Playing

This project is about designing an implementing an AI program that plays a game against human or computer opponents. You should be able to build a program that beats you, which is an interesting experience.

The game for this term is Checkers, also known as English Draughts. Checkers has a long history, including a long history in the world of computing.

Arthur Samuel's 1956 Checkers program was not only one of the first game-playing programs, it was also one of the first machines to "learn." It played against itself to improve, a strategy that has more recently been employed by AlphaGo and other top-level programs.

Jonathan Schaeffer and a team from the University of Alberta developed a program, Chinook, that beat the human world champions in 1994 and 1995. Schaeffer described the work, including their championship matches, in the book *One Jump Ahead: Challenging Human Supremacy at Checkers* (ISBN-13: 978-0387949307). The group went on to completely solve the game in 2007 (Schaeffer, J., *et al.*, "Checkers is Solved," *Science* 317(5844): pp. 1518–22). He also updated his book to include the later work: *One Jump Ahead: Computer Perfection at Checkers* (ISBN-13: 978-0387765754).

Please Note: Obviously, people have written Checkers programs for about as long as there have been computers. I wrote one myself for my undergraduate AI course. If you want to learn anything, avoid searching for information about the game beyond Wikipedia until you have done the basic implementation **YOURSELF**.

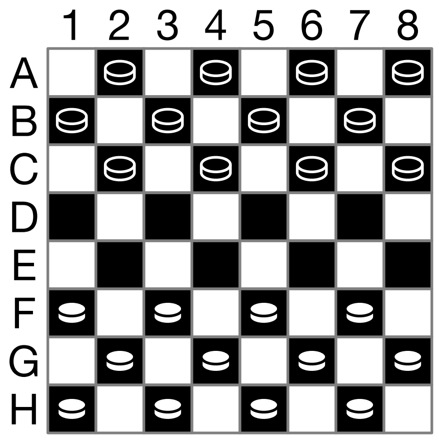

Figure 1: Initial arrangement of pieces for standard 8x8 checkers

Rules of Checkers (English Draughts)

Board and Pieces

- The game is played on a rectangular board divided into squares that alternate between black and white (dark and light) as shown in Figure 1. The standard size is 8x8.
- Rows are labelled with letters starting with A at the top. Columns are labelled with number starting with 1 at the left. Squares are referred to by row and column, from in “A1” to “H8” (on an 8x8 board).[1]
- Only the dark squares are used.
- One of the players has the black (dark) pieces; the other has the white (light) pieces.
- The initial configuration of the board for standard 8x8 checkers is shown in the figure.

Moving

- Players take turns moving.

[1]The traditional notation numbers the dark squares. This is much harder for novices to understand.

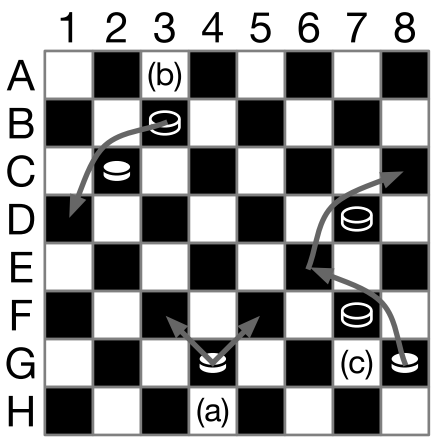

Figure 2: (a) Basic moves for white: G4-F3 and G4-F5; (b) Single capture for black: B3xD1; (c) Multiple capture for white: G8xE6xC8

- The player with the black (dark) pieces moves first.
- A player may move a single piece diagonally forward to an unoccupied (dark) square as shown in Figure 2(a).
- A simple move from square S to square D is written as "S-D", such as "G4-F3".

Capturing

- If the diagonally adjacent square is occupied by an opponent's piece AND the square behind it (in the same direction) is empty, the opponent's piece may be captured by jumping over it and landing in the further square (see Figure 2(b)).
- Captured pieces are removed the from the board.
- Captures are written as "SxD", such as "G3xD1".
- If another opponent piece is in a square adjacent to the landing square (in either forward direction), and the square behind it in the same direction is empty, that opponent's piece may also be captured (see Figure 2(c)). This may be repeated indefinitely in a single move and is written as "SxD_1x$D_2 \ldots$", such as "G8xE6xC8".
- A player MUST capture the maximum possible number of their opponent's pieces in any turn. They MAY NOT decline to capture a capturable piece, or decline to perform a longest possible sequence of captures. Pro Tip: Exploiting this rule is the key to playing good Checkers.

Kings

- When a piece reaches the farthest row forward (the opponent's edge of the board), it becomes a king (it is "crowned" or "kinged").
- Starting on their next move, kings may move and capture forward or backward, although still only diagonally.
- Kings must still follow the maximum capture rule, including their additional directions of movement.

End of Game

- A player wins by capturing all their opponent's pieces.
- A player loses if they cannot make a legal move.
- The game may end in a draw if both sides agree, or as a result of somewhat complicated rules (which vary between organizations). You don't have to worry about handling draws for this project unless you want to. You may assume that if there is no winner after 50 moves, the game is a draw. That actually isn't true in Checkers, but it's ok for this project. You are welcome to experiment with this limit, or implement the WCDF rules for detecting draws. Just tell us what you did in your README if you do something interesting.

Requirements

1. Develop a program that plays Checkers on a 4x4 board. Each player starts with two pieces on their own first row (see Figure 3). This is not a hard game, but it is simple enough that you can see how your program is playing (and play pretty well yourself, even if you've never played Checkers).
   - You MUST use an adversarial state-space search approach to solving the problem.
   - Design your data structures using the formal model of the game. We MUST see classes corresponding to the elements of the formal model (see "Method" below).
   - You MUST use the appropriate standard algorithm(s) to select optimal moves. You should be able to search the entire tree for 4x4 Checkers.

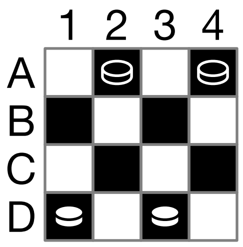

Figure 3: Initial board configuration for 4x4 Checkers

- Your program should be able to play quickly and perfectly.

2. Develop a program that plays standard 8x8 Checkers.
   - You MUST again use an adversarial state-space search approach to solve the problem.
   - If you design this right for Part 1, you will be able to resuse it with *almost no work*. In fact, you will be able to adapt your program to *any* two-player, perfect knowledge, zero-sum game with very little work. How cool is that?
   - Choosing the best move in this game is significantly harder than the smaller game of Part 1. (You should be able to figure out at least an upper bound on how much harder it is.) You MUST use heuristic MINIMAX with alpha-beta pruning, if you didn't do that already for Part 1.
   - Your program should be able to play well. In particular, it should not take too long to choose a move (unless the user tells it to do so).

Your programs should use standard input and standard output to interact with the user. An example is shown on the next page. You are welcome to develop graphical interfaces if you like, but that's not the point of this course and will not be considered in grading.

Here's a quick example of what your program might look like when it runs. A complete transcript is at the end of this document.

```
Checkers by George Ferguson
Choose your gamne:
1. Small 4x4 Checkers
2. Standard 8x8 Checkers
Your choice? 1
Choose your opponent:
1. An agent that plays randomly
2. An agent that uses MINIMAX
3. An agent that uses MINIMAX with alpha-beta pruning
4. An agent that uses H-MINIMAX with a fixed depth cutoff
Your choice? 4
Depth limit? 5
Do you want to play BLACK (b) or WHITE (w)? w

  1 2 3 4
 +-+-+-+-+
A| |b| |b|
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: BLACK

I'm thinking...
  visited 62 states
  best move: b:A2-B1, value: 0.0
Elapsed time: 0.045 secs
b:A2-B1

  1 2 3 4
 +-+-+-+-+
A| | | |b|
 +-+-+-+-+
B|b| | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): d1-c2
```

Method

Start by designing your data structures.

What are the elements of the state-space formalization of a two-player, perfect knowledge, zero sum game? We've seen them in class and in the textbook.

If you're using Java, turn these into interfaces. Hint: An interface doesn't actually have to define any methods, if that is useful. If you're not using Java, you still have the abstract specifications to guide you (although I *strongly* recommend Java for large programs with many interconnected pieces).

Now, how would you represent the specifics of Checkers using this framework? Don't forget to think about supporting different board sizes and/or starting configurations, per the requirements. Design classes that implement the abstract specifications (interfaces) for this specific game (problem domain). You can write simple test cases for these as you go and include them in each class' `main` method.

Next: The algorithms for solving the problem of picking a good move are defined in terms of the formal model. So you can implement them using only the abstract interfaces. Write one or more classes that answer the question "What move should I make in this state?" Hint: An agent that picks randomly but legally is not hard, using what you get from the formal model. So start there, but you should know what algorithms you really need for this problem.

Once you have one or more agents for playing the game, write the program that puts the pieces together to generate the gameplay shown in the example transcript. This will get you through Part 1.

What do you predict will happen if you use this program to play the full version of the game required in Part 2? Try it. You should know from class and from the textbook what algorithms and techniques you need if you haven't implemented them already. This may require some "additional information," which you will need to decide on and then add to your game-specific classes (because it's specific to this problem, right?).

The great thing about using the state-space search framework is that you can try different algorithms using the same representation of the problem. Even better, you can use the same algorithms to play different games just by changing the game-specific implementation of the abstract interfaces derived from the formal model of state-space search. And if the descriptions of the games were themselves machine-readable... general game playing perhaps?

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements
2. All source code for your project. Eclipse projects must include the project settings from the project folder. Non-Eclipse projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.
3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

Writeups other than the instructions in your README and your completed submission form are **not** required.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Practice

Use good object-oriented design. No giant `main` methods or other unstructured chunks of code. Comment your code liberally and clearly.

You may use Java, Python, or C/C++ for this project. I recommend that you use Java. Any sample code we distribute will be in Java. Other languages (Haskell, Clojure, Lisp, *etc.*) by arrangement with the TAs only.

You may **not** use any non-standard libraries. Python users: that includes things like NumPy. Write your own code—you'll learn more that way.

If you use Eclipse, make it clear how to run your program(s). Setup Build and Run configurations as necessary to make this easy for us. Eclipse projects with poor configuration or inadequate instructions will receive a poor grade.

Python projects must use Python 3 (recent version, like 3.7.x). Mac users should note that Apple ships version 2.7 with their machines so you will need to do something different.

If you are using C or C++, you should use reasonable "object-oriented" design not a mish-mash of giant functions. If you need a refresher on this, check out the C for Java Programmers guide and tutorial. You **must** use "`-std=c99 -Wall -Werror`" **and** have a clean report from `valgrind`. Projects that do not follow both of these guidelines will receive a poor grade.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will get the most out of this project if you write the code yourself.

That said, collaboration on the coding portion of projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC242.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.

- One member of the group should submit code on the group's behalf in addition to their writeup. Other group members should submit only a README indicating who their collaborators are.

- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

Example Output

Here's the full transcript of the 4x4 game shown earlier. I played white (second to move). The computer was using a depth limit of 5.

When drawing the board, I show normal pieces using lowercase `b` or `w`. The computer, playing black, gets a king after its second move. I show that with uppercase `B`.

Note the display of useful information about what the program was doing, such as elapsed time, number of states, utility of the chosen move (estimated utility, in this case), and so on. Your program doesn't have to do exactly what mine did, but it should do something similarly informative and interactive.

```
Checkers by George Ferguson
Choose your gamne:
1. Small 4x4 Checkers
2. Standard 8x8 Checkers
Your choice? 1
Choose your opponent:
1. An agent that plays randomly
2. An agent that uses MINIMAX
3. An agent that uses MINIMAX with alpha-beta pruning
4. An agent that uses H-MINIMAX with a fixed depth cutoff
Your choice? 4
Depth limit? 5
Do you want to play BLACK (b) or WHITE (w)? w

  1 2 3 4
 +-+-+-+-+
A| |b| |b|
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: BLACK

I'm thinking...
  visited 62 states
  best move: b:A2-B1, value: 0.0
Elapsed time: 0.045 secs
b:A2-B1

  1 2 3 4
 +-+-+-+-+
A| | | |b|
 +-+-+-+-+
B|b| | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): d1-c2
Elapsed time: 19.546 secs
w:D1-C2

  1 2 3 4
 +-+-+-+-+
A| | | |b|
 +-+-+-+-+
B|b| | | |
 +-+-+-+-+
C| |w| | |
 +-+-+-+-+
D| | |w| |
 +-+-+-+-+
Next to play: BLACK

I’m thinking...
  visited 4 states
  best move: b:A4-B3, value: -1.0
Elapsed time: 0.002 secs
b:A4-B3

  1 2 3 4
 +-+-+-+-+
A| | | | |
 +-+-+-+-+
B|b| |b| |
 +-+-+-+-+
C| |w| | |
 +-+-+-+-+
D| | |w| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): c2xa4
Elapsed time: 11.948 secs
W:C2xA4

  1 2 3 4
 +-+-+-+-+
A| | | |W|
 +-+-+-+-+
B|b| | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D| | |w| |
 +-+-+-+-+
Next to play: BLACK

I'm thinking...
  visited 2 states
  best move: b:B1-C2, value: -1.0
Elapsed time: 0.001 secs
b:B1-C2

  1 2 3 4
 +-+-+-+-+
A| | | |W|
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| |b| | |
 +-+-+-+-+
D| | |w| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): d3xc2
Invalid move!
Valid moves are: [w:D3xB1]
Try again!
Your move (? for help): d3xb1
Elapsed time: 26.796 secs
w:D3xB1

  1 2 3 4
 +-+-+-+-+
A| | | |W|
 +-+-+-+-+
B|w| | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D| | | | |
 +-+-+-+-+
Next to play: BLACK

Winner: WHITE
Total time:
     BLACK: 0.048 secs
     WHITE: 58.289 secs
```

Here's another transcript. This time I gave the computer a depth limit of eight, and it beat me. You can see where it's evaluation went from 0.0 (draw) to -1.0 (for me, meaning win for it).

```
Checkers by George Ferguson
Choose your gamne:
1. Small 4x4 Checkers
2. Standard 8x8 Checkers
Your choice? 1
Choose your opponent:
1. An agent that plays randomly
2. An agent that uses MINIMAX
3. An agent that uses MINIMAX with alpha-beta pruning
4. An agent that uses H-MINIMAX with a fixed depth cutoff
Your choice? 4
Depth limit? 8
Do you want to play BLACK (b) or WHITE (w)? w

  1 2 3 4
 +-+-+-+-+
A| |b| |b|
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: BLACK

I'm thinking...
  visited 164 states
  best move: b:A4-B3, value: 0.0
Elapsed time: 0.009 secs
b:A4-B3

  1 2 3 4
 +-+-+-+-+
A| |b| | |
 +-+-+-+-+
B| | |b| |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |w| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): d3-c2
Elapsed time: 9.416 secs
```

```
w:D3-C2

  1 2 3 4
 +-+-+-+-+
A| |b| | |
 +-+-+-+-+
B| | |b| |
 +-+-+-+-+
C| |w| | |
 +-+-+-+-+
D|w| | | |
 +-+-+-+-+
Next to play: BLACK

I’m thinking...
  visited 159 states
  best move: b:B3-C4, value: 0.0
Elapsed time: 0.003 secs
b:B3-C4

  1 2 3 4
 +-+-+-+-+
A| |b| | |
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| |w| |b|
 +-+-+-+-+
D|w| | | |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): c2-b3
Elapsed time: 13.403 secs
w:C2-B3

  1 2 3 4
 +-+-+-+-+
A| |b| | |
 +-+-+-+-+
B| | |w| |
 +-+-+-+-+
C| | | |b|
 +-+-+-+-+
D|w| | | |
 +-+-+-+-+
Next to play: BLACK

I’m thinking...
  visited 324 states
  best move: B:C4-D3, value: 1.0
```

```
Elapsed time: 0.004 secs
B:C4-D3

  1 2 3 4
 +-+-+-+-+
A| |b| | |
 +-+-+-+-+
B| | |w| |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |B| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): b3-a4
Elapsed time: 6.928 secs
w:B3-A4

  1 2 3 4
 +-+-+-+-+
A| |b| |W|
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |B| |
 +-+-+-+-+
Next to play: BLACK

I'm thinking...
  visited 209 states
  best move: b:A2-B3, value: 1.0
Elapsed time: 0.002 secs
b:A2-B3

  1 2 3 4
 +-+-+-+-+
A| | | |W|
 +-+-+-+-+
B| | |b| |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| |B| |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): a4xc2
Elapsed time: 10.182 secs
```

```
W:A4xC2

  1 2 3 4
 +-+-+-+-+
A| | | | |
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| |W| | |
 +-+-+-+-+
D|w| |B| |
 +-+-+-+-+
Next to play: BLACK

I’m thinking...
  visited 3 states
  best move: B:D3xB1, value: 1.0
Elapsed time: 0.000 secs
B:D3xB1

  1 2 3 4
 +-+-+-+-+
A| | | | |
 +-+-+-+-+
B|B| | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D|w| | | |
 +-+-+-+-+
Next to play: WHITE

Your move (? for help): d1-c2
Elapsed time: 15.126 secs
w:D1-C2

  1 2 3 4
 +-+-+-+-+
A| | | | |
 +-+-+-+-+
B|B| | | |
 +-+-+-+-+
C| |w| | |
 +-+-+-+-+
D| | | | |
 +-+-+-+-+
Next to play: BLACK

I’m thinking...
  visited 1 states
  best move: B:B1xD3, value: 1.0
```

```
Elapsed time: 0.000 secs
B:B1xD3

  1 2 3 4
 +-+-+-+-+
A| | | | |
 +-+-+-+-+
B| | | | |
 +-+-+-+-+
C| | | | |
 +-+-+-+-+
D| | |B| |
 +-+-+-+-+
Next to play: WHITE

Winner: BLACK
Total time:
     BLACK: 0.019 secs
     WHITE: 55.055 secs
```

This game illustrates how forcing your opponent make a move that they know is bad is the key to playing good Checkers. I had to capture with A4xC2 due to the maximum capture rule, after which it took my king with D3xB1. Then I had to move D1-C2 since that was my only move, after which it took my other piece with B1xD3. Sigh. Just like when I was an undergrad. . . ;-)