CSC242: Intro to AI
Project 1: Game Playing

This project is about designing and 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.

But note: this project is not really about the game. I don't really care if you can write a program that plays one specific simple game.

This project is about understanding the **formal model** of adversarial search and **applying** it to one particular problem, namely playing some specific game. As seen in the textbook and as we will see in class, the algorithms for solving adversarial search problems are **independent** of the specific details of the problem (that is, of the game). Use that important fact as you design and implement your program.

The game for this term is Pawntastic. Pawntastic is based on chess, but it is much simpler than chess. You do not need to know anything about chess to do the project. Everything that you need to know is described in this document. But if you do play chess, you may be interested to know that more complicated versions of Pawntastic are used by real chess players for practice. The simple version that we will use in this project is interesting enough to be challenging for humans and can be hard enough to be challenging for computers.[1]

Please Note: There is a long history of computer programs that play chess and similar games. Alan Turing wrote a chess program for his newly-invented "machines" and hand-simulated its execution to play against a colleague. And of course Deep Blue beating Garry Kasparov in 1997 was a watershed moment for AI. If you want to learn anything, avoid searching for information about the game beyond the page linked above until you have done the basic implementation **YOURSELF**.

[1]When you need a break from the project, I recommend either Chess, the musical set in the 1980's during the Cold War or the Netflix series The Queen's Gambit from 2020 (which is based on a novel from 1983).

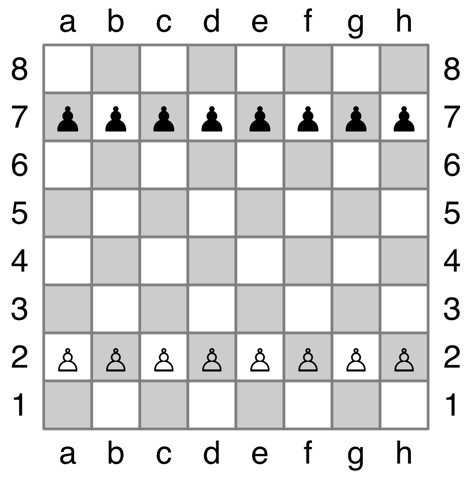

Figure 1: Initial arrangement of pieces for standard 8x8 Pawntastic

Pawntastic

Pawntastic is a two-player game. The two players are traditionally named “black” and “white” for the color of their game pieces.

Board and Pieces

- Pawntastic is played on a chess board: a rectangular board divided into squares. The size of a standard chess board is 8x8, but Pawntastic can be played on (almost) any size and (rectangular) shape of board.
- In standard chess notation, columns (traditionally called “files”) are labelled with letters starting with “a”. Rows (traditionally called “ranks”) are labelled with numbers starting with “1” **at the bottom**. Squares are referred to by column and row, from in “a1” in the bottom left corner to “h8” in the top right corner (on an 8x8 board).
- In standard chess, there are many types of pieces. Pawntastic uses only one of them: the pawn. Pawns are traditionally shown using the Unicode symbols “♟” (U+265F) for a black pawn and “♙” (U+2659) for a white pawn.
- As in standard chess, the black pawns begin on the second row from the top (row 7) and the white pawns begin on the second row from the bottom (row 2). This is shown in Figure 1. You can generalize this to any size greater than or equal to four.

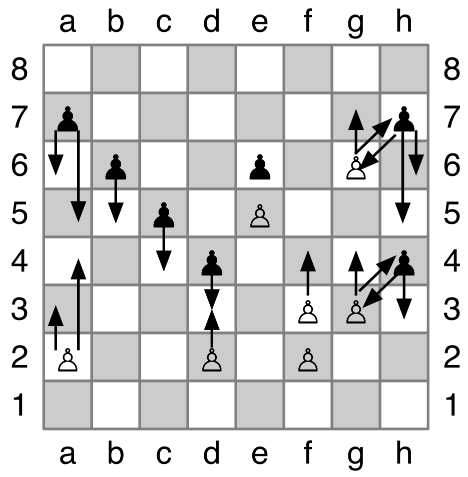

Figure 2: Possible moves for pawns (in chess and Pawntastic)

Moves

- Pawns only move forward. Black pawns move down the board; white pawns move up the board. Pawns stay in the same column (file) unless capturing.
- A pawn may move one square forward if the space ahead is empty.
- On its first move **only**, a pawn may also move two spaces forward, if both spaces are empty.
  - In Figure 2, the black pawn at a7 and the white pawn at a2 may move either one or two spaces forward on their first moves.
  - The black pawns at b6 and c5 may only move one square forward since they have already moved.
  - The white pawn at d2 may only move one square forward on its first move since d4 is not empty.
  - Neither the black pawn at e6 nor the white pawn at e5 may move since the space ahead of them is not empty. The white pawn at f2 may not move since the space ahead of it is not empty.
- A pawn may **capture** an opponent's pawn on a diagonally adjacent square by moving to that square.[2] A captured pawn is removed from the board.
  - In the figure, the black pawn at h7 may capture the white pawn at g6 and vice-versa, in addition to their other possible forward moves.
  - The white pawn at g3 may capture the black pawn at h4 and vice-versa, in addition to their other possible forward moves.

[2] You are not required to implement "en passant captures," although you are welcome to do so.

Start and End of Game

As in standard chess, the player with the white pieces moves first.

The goal of the game is to move **any one of your pawns** to the opponent's back row (so black is trying to get to the bottom row, and white to the top row). The first player to do this wins the game.

If a player has no legal moves when it is their turn, the game ends in a **stalemate** and there is no winner (so a tie or draw).

Requirements

1. You **must** develop **a single program** that plays Pawntastic on any size of square board (non-square boards optional). Your program **must** ask the user for the size.
2. You **must** use an adversarial state-space search approach.
3. You **must** design your data structures using the formal model of adversarial state-space search. We **must** see classes corresponding to the elements of the formal model (see below for some suggestions about this).
4. You **must** implement the MINIMAX algorithm for finding moves that are *optimal* with respect to the game's utility function.
5. You **must** implement heuristic H-MINIMAX with α/β pruning for finding moves that are good (but not necessarily optimal).
   - You **must** implement a *heuristic evaluation function* (AIMA 5.3.1 4th ed., although material advantage as shown in that example may or may not be a good idea for Pawntastic).
   - You **must** implement a *cutoff test* (AIMA 5.3.2 4th ed., also seen in class). I recommend using a depth cutoff, but there are other possibilities.
6. Your program **must** ask the user which algorithm to use and any other necessary information, such as the cutoff depth for H-MINIMAX.
7. Your program **must** validate the user's input and only allow the user to make legal moves. This is not as hard as it may seem. Think about it. Your program knows a lot about what moves are possible.
8. Your program **must** use standard input and standard output to interact with the user (`System.in` and `System.out` in Java). An example is included at the end of this document. You are welcome to develop graphical interfaces if you like, but that is not the point of this course and will not be considered in grading.

Evaluation

Your project will be graded as follows:

1. **(70%)** Program plays 4x4 and 5x5 Pawntastic **perfectly** using full MINIMAX.
    - The starting configuration for these games is shown in Figure 3. The black pawns always start on the second row from the top, and the white pawns always start on the second row from the bottom.
    - This game is simple enough that you can see how your program is playing. And it is also not too hard for humans to play well.
    - Your implementation **must** be able to search the entire tree for 4x4 and 5x5 Pawntastic and hence your program **must** be able to play perfectly and relatively quickly for these small games.

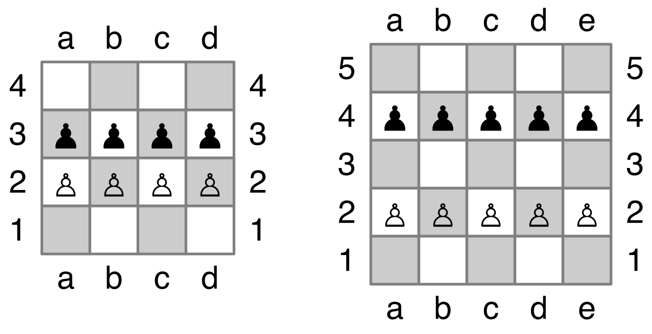

Figure 3: Initial board configurations for 4x4 and 5x5 Pawntastic

2. **(30%)** Program plays standard 8x8 Pawntastic well (but perhaps not perfectly) using H-MINIMAX with α/β pruning.
    - Your program **must** suggest reasonable choices when it asks for the search parameters (cutoff depth, *etc.*.), so that the program chooses its moves reasonably quickly.

There is no opportunity for extra credit on this project.

How To Do This Project

I will assume that you are using Java for this. Why wouldn't you use Java for a program that involves representations of many different types of objects with complicated relationships between them and algorithms for computing with them? Seriously. That's the kind of thing that Java was designed for. If you want to ignore my advice and use Python, well ok, but it has to be well-designed and object-oriented, not a mishmash of lists and dictionaries. See "Programming Practice," below.

Start by designing your data structures.

What are the elements of **the state-space formalization of a two-player, perfect knowledge, zero sum game**? They are in the textbook and we will see (or have seen) them in class. They can be used to formalize any game, not just Pawntastic. That is so cool.

In Java, these can be specified using interfaces or abstract classes. For example: `Game`, `Player`, `State`, `Board` (for a board game like Pawntastic), `Location` (on a `Board`), `Move` (from one `Location` to another), *etc.*

Now implement the elements of the formal specification for Pawntastic. You should have concrete classes for each of the elements of the model, and methods for each of the functions in the model. You can write simple test cases for these as you go and include them in each class' `main` method.

Looking at the functions in the formal model:

- You need to be able to compute the applicable (possible) `Move`s in a `State`.
- You need to be able to compute the new `State` that results from making a move (performing an action) in a given `State`. Remember to create a **new** state rather than mutating (changing) the old one, unless you adjust the search algorithms to "undo" changes properly as they backtrack.
- You need to be able to test whether a `State` is a terminal state. For Pawntastic, this depends in part on the `Board` and in part on which moves are possible (applicable) for the current `Player`. And both of those are part of the `State`, right?
- Finally, you need be able to compute the utility of a terminal `State` for a given `Player`, which is easy once you know who has won (if anyone has won).

Next: The algorithms for solving the problem of picking a good move are defined purely in terms of the formal model. So you can implement them using **only the abstract interfaces**. Seriously. Nothing in the definitions of MINIMAX and H-MINIMAX involve anything about any specific game. Even for H-MINIMAX, the game-specific (problem-specific) knowledge is only inside the heuristic function.

Write one or more classes with a method that answers the question "Given a state in a game, what is the best move (for the player whose turn it is to move in that state)?" Hint: A simple way to do this is to choose randomly among the possible actions. Such a program is guaranteed to play legally and may even beat you sometimes. But eventually you need to implement MINIMAX and H-MINIMAX (**and test them**, probably using very simple preset states).

Once you have one or more of these "agents" for playing the game, write the program that puts the pieces together to generate gameplay similar to that shown in the example transcript.

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?

Additional Requirements and Policies

The short version:

- You may **only** use Java or Python. I **STRONGLY** recommend Java.
- You **must** use good object-oriented design (yes, even in Python).
- There are other language-specific requirements detailed below.
- You must submit a ZIP including your source code, a README, and a completed submission form by the deadline.
- You **must** tell us how to build your project in your README.
- You **must** tell us how to run your project in your README.
- Projects that do not compile will receive a grade of **0**.

- Projects that do not run or that crash will receive a grade of **0** for whatever parts did not work.
- Late projects will receive a grade of **0** (see below regarding extenuating circumstances).
- **You will learn the most if you do the project yourself**, but collaboration is permitted in groups of up to 3 students.

Detailed information follows. . .

Programming Requirements

- You may use Java or Python for this project.
  - I **STRONGLY** recommend that you use Java.
  - Any sample code we distribute will be in Java.
- You **must** use good object-oriented design.
  - You **must** have well-designed classes (and perhaps interfaces).
  - Yes, even in Python.
- No giant `main` methods or other unstructured chunks of code.
  - Yes, even in Python.
- Your code should use meaningful variable and function/method names and have plenty of meaningful comments.
  - I can't believe that I even have to say that.
  - And yes, even in Python.

Submission Requirements

You **must** submit your project as a ZIP archive containing the following items:

1. The source code for your project.
2. A file named `README.txt` or `README.pdf` (see below).
3. A completed copy of the submission form posted with the project description (details below).

Your README **must** include the following information:

1. The course: “CSC242”
2. The assignment or project (*e.g.*, “Project 1”)
3. Your name and email address
4. The names and email addresses of any collaborators (per the course policy on collaboration)
5. Instructions for building and running your project (see below).

The purpose of the submission form is so that we know which parts of the project you attempted and where we can find the code for some of the key required features.

- **Projects without a submission form or whose submission form does not accurately describe the project 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.

Project Evaluation

You **must** tell us in your README file how to build your project (if necessary) and how to run it.

Note that we will **not** load projects into Eclipse or any other IDE. We **must** be able to build and run your programs from the command-line. If you have questions about that, go to a study session.

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 your grade will be.** It is **your** job to make the building of your project easy and the running of its program(s) easy and informative.

For **Java** projects:

- The current version of Java as of this writing is: 20.0.2 (OpenJDK)
- If you provide a `Makefile`, just tell us in your README which target to make to build your project and which target to make to run it.

- Otherwise, a typical instruction for building a project might be:

  ```
  javac *.java
  ```

  Or for an Eclipse project with packages in a `src` folder and classes in a `bin` folder, the following command can be used from the `src` folder:

  ```
  javac -d ../bin `find . -name '*.java'`
  ```

- And for running, where `MainClass` is the name of the main class for your program:

  ```
  java MainClass [arguments if needed per README]
  ```

  or

  ```
  java -d ../bin pkg.subpkg.MainClass [arguments if needed per README]
  ```

- You **must** provide these instructions in your README.

For **Python** projects:

I strongly recommend that you **not** use Python for projects in CSC242. All CSC242 students have at least two terms of programming in Java. The projects in CSC242 involve the representations of many different types of objects with complicated relationships between them and algorithms for computing with them. That's what Java was designed for.

But if you insist. . .

- The latest version of Python as of this writing is: 3.11.4 (python.org).
- You must use Python 3 and we will use a recent version of Python to run your project.
- You may **NOT** use any non-standard libraries. This includes things like NumPy or pandas. Write your own code—you'll learn more that way.
- We will follow the instructions in your README to run your program(s).
- You **must** provide these instructions in your README.

For **ALL** projects:

We will **NOT** under any circumstances edit your source files. That is your job.

Projects that do not compile will receive a grade of 0. There is no way to know if your program is correct solely by looking at its source code (although we can sometimes tell that is incorrect).

Projects that do not run or that crash will receive a grade of 0 for whatever parts did not work. You earn credit for your project by meeting the project requirements. Projects that don't run don't meet the requirements.

Any questions about these requirements: go to study session **BEFORE** the project is due.

Late Policy

Late projects will receive a grade of 0. You **must** 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

I assume that you are in this course to learn. You will learn the most if you do the projects **yourself**.

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

- Teams of no more than 3 students, all currently taking CSC242.
- Any team member **must** be able to explain anything they or their team submits, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the team should submit code on the team's behalf in addition to

their writeup. Other team members **must** submit a README (only) indicating who their collaborators are.

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

Academic Honesty

I assume that you are in this course to learn. You will learn nothing if you don't do the projects **yourself**.

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.

Sample Output

Here’s a quick example of my program running. I suggest that you make yours look like mine. Your program **must** print moves and the board as shown here using standard notation for the squares (you can use “x” and “o” instead of unicode symbols for pawns if you want).

```
Pawntastic by George Ferguson
Note:  en passant capture is still TODO
Choose your game:
 4.  Tiny 4x4 Pawntastic
 5.  Very small 5x5 Pawntastic
 6.  Small 6x6 Pawntastic
 8.  Standard 8x8 Pawntastic
10.  Jumbo 10x10 Pawntastic
Or enter any size >=4 to play that size game
Your choice?  5
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
5.  An agent that uses H-MINIMAX with a fixed depth cutoff and alpha-beta pruning
Your choice?  2
Trace opponent agent?  (y/n) n
Do you want to play BLACK (B) or WHITE (W)? (WHITE plays first) b

   a b c d e
  +-+-+-+-+-+
5 | | | | | | 5
  +-+-+-+-+-+
4 |♟|♟|♟|♟|♟| 4
  +-+-+-+-+-+
3 | | | | | | 3
  +-+-+-+-+-+
2 |♙|♙|♙|♙|♙| 2
  +-+-+-+-+-+
1 | | | | | | 1
  +-+-+-+-+-+
   a b c d e
Next to play:  WHITE/♙
```

```
I'm thinking...
  visited 42625162 states
  result:  ⟨1.0, ♙:b2-b3⟩
Elapsed time:  46.836 secs
♙:b2-b3

   a b c d e
  +-+-+-+-+-+
5 | | | | | | 5
  +-+-+-+-+-+
4 |♟|♟|♟|♟|♟| 4
  +-+-+-+-+-+
3 | |♙| | | | 3
  +-+-+-+-+-+
2 |♙| |♙|♙|♙| 2
  +-+-+-+-+-+
1 | | | | | | 1
  +-+-+-+-+-+
   a b c d e
Next to play:  BLACK/♟
Your move (?  for help):  c4 b3
Elapsed time:  70.275 secs
♟:c4xb3

   a b c d e
  +-+-+-+-+-+
5 | | | | | | 5
  +-+-+-+-+-+
4 |♟|♟| |♟|♟| 4
  +-+-+-+-+-+
3 | |♟| | | | 3
  +-+-+-+-+-+
2 |♙| |♙|♙|♙| 2
  +-+-+-+-+-+
1 | | | | | | 1
  +-+-+-+-+-+
   a b c d e
Next to play:  WHITE/♙
I'm thinking...
  visited 44110817 states
  result:  ⟨1.0, ♙:c2-c4⟩
Elapsed time:  1.719 secs
```

```
♙:c2-c4

   a b c d e
  +-+-+-+-+-+
5 | | | | | | 5
  +-+-+-+-+-+
4 |♟|♟|♙|♟|♟| 4
  +-+-+-+-+-+
3 | |♟| | | | 3
  +-+-+-+-+-+
2 |♙| | |♙|♙| 2
  +-+-+-+-+-+
1 | | | | | | 1
  +-+-+-+-+-+
   a b c d e
Next to play:  BLACK/♟
Your move (?  for help):
```

A Scaling experiments

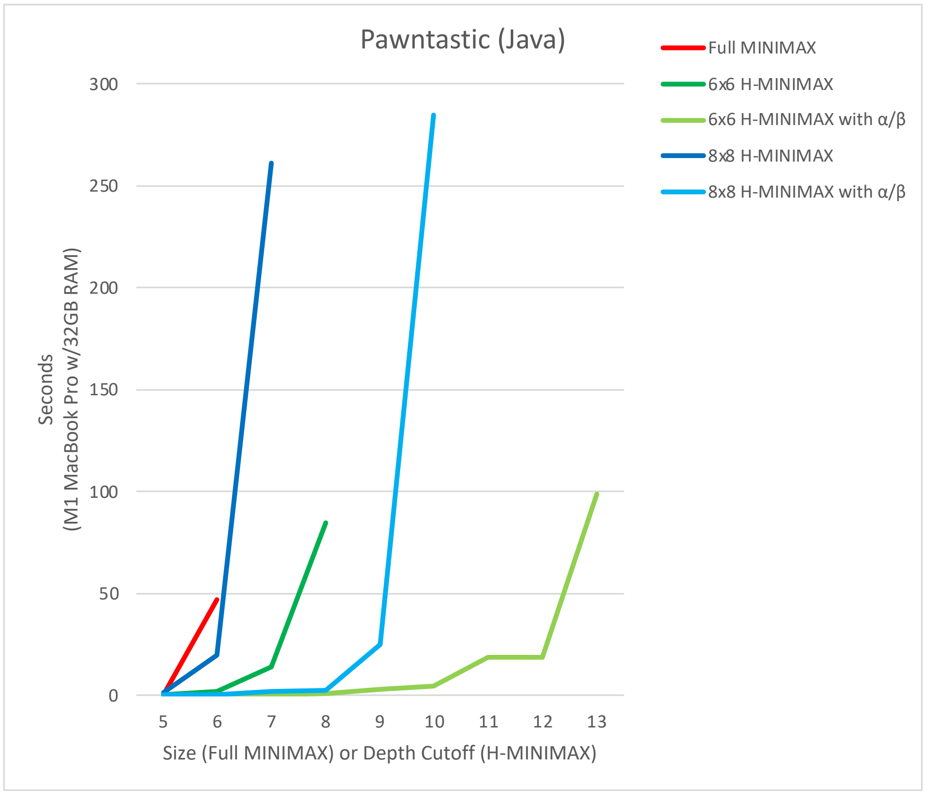

Figure 4: Experimental results for my implementation choosing the first move in Pawntastic

For full MINIMAX, the horizontal axis is the size of the board. Fast for 4x4. Not bad for 5x5. I waited more than five minutes for 6x6 before giving up. So that curve should look more like a very, very steep exponential than a small line segment. ;-)

For H-MINIMAX, the horizontal axis is the depth cutoff. The blue curves are for 6x6 and the green curves are for 8x8. You can see the exponential growth very clearly.

You can also see the very clear benefit of α-β pruning, as seen in class and in the textbook. For both 6x6 (light blue) and 8x8 (light green), the pruning allows the program to search **about 50% deeper** in the same amount of time, for **almost no effort** in terms of code.

Of course it's still exponential in the long run.