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 Tic-tac-toe.

Tic-tac-toe is an easy game to describe, which makes it easy to learn for humans and easy to program for computers. As anyone who has played with small children knows, basic tic-tac-toe is not hard for humans. It's also not hard for computers. So in this project your program will also need to play a more complicated version of tic-tac-toe called "Nine-board Tic-tac-toe."

Please Note: There is a long history of computer programs that play Tic-tac-toe and similar games, including in CSC242. If you want to learn anything, avoid searching for information about the game beyond the Wikipedia pages linked above until you have done the basic implementation **YOURSELF**.

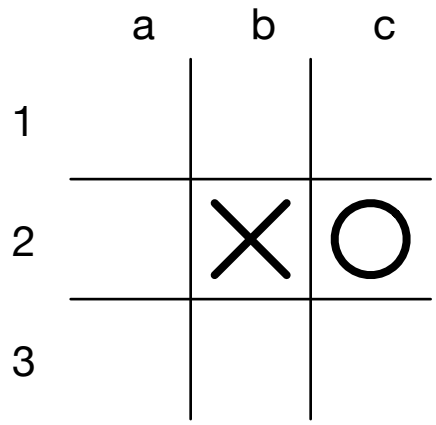

Figure 1: Basic 3x3 tic-tac-toe board after two moves

Rules of Tic-tac-toe

Basic Tic-tac-toe

- Basic Tic-tac-toe is played by two players, X and O, who take turns putting their marks in a 3x3 grid of spaces, as shown in Figure 1.
- Columns are labelled with letters starting with “a”. Rows are labelled with numbers starting with 1. Squares are referred to by column and row, from “a1” to “c3” (on an 3x3 board). Note that this labelling starts at one (1), not zero.
- We will use the convention that X plays first.
- The goal of the game is for a player to form a line of three of their marks either horizontally, vertically, or diagonally.
- The game ends when one player has created such a line, or when the board is full, in which case the game is a tie (draw).

Nine-board Tic-tac-toe

- Nine-board tic-tac-toe is played using a 3x3 grid of 3x3 tic-tac-toe boards, as shown in Figure 2.

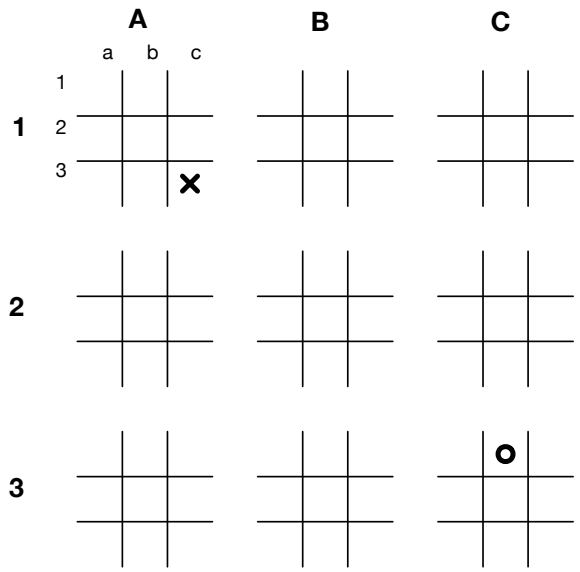

Figure 2: Nine-board Tic-tac-toe board after two moves

- Columns are labelled with letters starting with “A”. Rows are labelled with numbers starting with 1. SquaresBoards are referred to by column and row, from in “A1” to “C3” (on an 3x3 Ultimate tic-tac-toe board).
- The goal of the game is to win a on any *single* board, just like in regular tic-tac-toe.
- However there is one crucial constraint: If a player has just played at some position on some board, then the next player must play on the board in *the corresponding position* in the grid.

  For example, if a player marks the bottom right *position*, “c3”, of any board (as X did in the example shown in the figure), then the next player must mark any open space on the bottom right *board* (the board “C3”, as O has done in the example).
- If the board required by the preceding rule is full, the player can play on any board.
- The first player to play can also play on any board.

Requirements

Part 1 (60%)

Develop a program that plays basic Tic-tac-toe on a 3x3 board. This is not a hard game. It is simple enough that you can see how your program is playing. And you can probably play very well yourself, even if you've never played Tic-tac-toe.

- You **MUST** use an adversarial state-space search approach to solving the problem, as seen in the textbook and in class.
- Design your data structures using the formal model of the game. We **MUST** see classes corresponding to the elements of the formal model (see below).
- You **MUST** use the appropriate standard algorithm to select *optimal* moves. Do not use algorithms that make *imperfect* decisions for this part of the project.
- You should be able easily to search the entire tree for 3x3 Tic-tac-toe and hence your program should be able to play perfectly and quickly (on modern computers).
- 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 is possible and what isn't.

Part 2 (40%)

Develop a program that plays Nine-board Tic-tac-toe using a 3x3 grid of 3x3 boards. This can be a separate program, or you can have a single program that asks the user which game to play.

- You **MUST** again use an adversarial state-space search approach to solve the problem.
- In fact, if you design this right for Part 1, you will be able to reuse 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* for full points on this part of the project.

- Your program should be able to play well. In particular, it should not take too long to choose a move. Your program should give the user a chance to specify the search limit, using one or more of the ideas described in the textbook and discussed in class (*e.g.*, depth limit, time limit, *etc.*).

It is not a requirement, but there is another version of Tic-tac-toe called Ultimate Tic-tac-toe. It's the same arrangement as Nine-board, but the goal is to win three-in-a-row *boards* rather than getting three-in-a-row on a single board. If you've designed your program properly, it should be easy to make it play that game also. But not a requirement (also not extra credit, beyond the extra that you learn by doing it).

Your programs **MUST** use standard input and standard output to interact with the user. An example is shown at the end of this document. 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.

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. C/C++: you're on your own. See also "Programming Requirements," 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? We've seen them in class and in the textbook. They can be used to formalize any game, not just Tic-tac-toe.

In Java, you would probably turn these into interfaces. Python now has some things similar to interfaces, but Python is philosophically opposed to careful specfication of behavior (see "duck typing"), which is one reason why I don't recommend Python for large projects.

Now, how would you represent the specifics of Tic-tac-toe using this framework? Design classes that implement the elements of the abstract model 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 "Given a state, what is the best move (for the player whose turn it is to move in that state)?" This usually involves generating all the possible moves in a state, so you should probably start there, and then figure out how to pick the *best* move. 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 of these "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 bigger 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 to make *imperfect real-time decisions*. This

may require some “additional information,” which you will need to decide on and then add to your game-playing agent(s). The algorithm(s) apply to any game. The “additional information” is specific to a given game, or more precisely, to a specific way of playing a specific game. You might want to try several ways.

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 use Java, C/C++, 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.
- Do not copy code from other students or from the Internet.

Detailed information follows. . .

Programming Requirements

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

  - In Java, C++, or Python, you MUST have well-designed classes.
  - Yes, even in Python.
  - In C, you must have well-designed "object-oriented" data structures (refresher: C for Java Programmers guide and tutorial)
- No giant `main` methods or other unstructured chunks of code.
- Your code should use meaningful variable and function/method names and have plenty of meaningful comments. But you know that...

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: 16.0.2 (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 **C/C++** projects:

- You MUST use at least the following compiler arguments:

  ```
  -Wall -Werror
  ```

- 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:

  ```
  gcc -Wall -Werror -o EXECUTABLE *.c
  ```

  where `EXECUTABLE` is the name of the executable program that we will run to execute your project.
- And for running:

  ```
  ./EXECUTABLE [arguments if needed per README]
  ```

- You **MUST** provide these instructions in your README.
- Your code should have a clean report from `valgrind`. If we have problems running your program and it doesn't have a clean report from `valgrind`, we are going to assume that your code is wrong. If you are a C/C++ programmer and don't know about `valgrind`, look it up. It is an essential tool.

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: 3.9.6 (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. 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:

- 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 MUST submit a README (only) indicating who their collaborators are.
- All members of a collaborative group 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.

Full Game Transcript

Here's the transcript of a full game of basic 3x3 Tic-tac-toe. You can see from the output exactly when it realized that it could win.

```
Tic-Tac-Toe by George Ferguson
Choose your game:
1. Basic 3x3 Tic-tac-toe
2. Nine-board Tic-tac-toe
3. Ultimate Tic-tac-toe
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 alpha-beta and a fixed depth cutoff
Your choice? 3
Do you want to play X or O? x

   a   b   c
1    |   |
  ---+---+---
2    |   |
  ---+---+---
3    |   |
Next to play: X

Your move [col row]? a1
Elapsed time: 5.771 secs
X@a1

   a   b   c
1  X |   |
  ---+---+---
2    |   |
  ---+---+---
3    |   |
Next to play: O

I'm thinking...
  visited 59704 states
  best move: O@b2, value: 0.0
Elapsed time: 0.053 secs
O@b2

   a   b   c
1  X |   |
  ---+---+---
2    | O |
  ---+---+---
```

```
3    |   |
Next to play: X

Your move [col row]? a2
Elapsed time: 8.717 secs
X@a2

   a   b   c
1  X |   |
  ---+---+---
2  X | O |
  ---+---+---
3    |   |
Next to play: O

I'm thinking...
  visited 934 states
  best move: O@a3, value: 0.0
Elapsed time: 0.001 secs
O@a3

   a   b   c
1  X |   |
  ---+---+---
2  X | O |
  ---+---+---
3  O |   |
Next to play: X

Your move [col row]? c1
Elapsed time: 9.607 secs
X@c0

   a   b   c
1  X |   | X
  ---+---+---
2  X | O |
  ---+---+---
3  O |   |
Next to play: O

I'm thinking...
  visited 46 states
  best move: O@b1, value: 0.0
Elapsed time: 0.000 secs
O@b1

   a   b   c
1  X | O | X
  ---+---+---
2  X | O |
```

```
  ---+---+---
3  O |   |
Next to play: X

Your move [col row]? c3
Elapsed time: 9.940 secs
X@c3

   a   b   c
1  X | O | X
  ---+---+---
2  X | O |
  ---+---+---
3  O |   | X
Next to play: O

I'm thinking...
  visited 3 states
  best move: O@b3, value: 1.0
Elapsed time: 0.000 secs
O@b3

   a   b   c
1  X | O | X
  ---+---+---
2  X | O |
  ---+---+---
3  O | O | X
Next to play: X

Winner: O
Total time:
         X: 34.036 secs
         O: 0.054 secs
```