CSCI-561 - Spring 2026 - Foundations of Artificial Intelligence
Homework 2 – The new game of "Heirs": A Strategic Board Game of Youthful Royalty

Due April 6, 2026 23:59

Image from ChatGPT

Guidelines

This is a programming assignment. Since **each homework is checked via an automated A.I. script**, your output should match the specified format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided below. You should upload and test your code on vocareum.com, and you will also submit it there.

Grading

Your code will be tested as follows: Your program should not require any command-line argument. It should read a text file called "input.txt" in the current directory that contains the current state of the game. It should write a file "output.txt" with your chosen move to the same current directory. Format for input.txt and output.txt is specified below. End-of-line character is LF (since vocareum is a Unix system and follows the Unix convention).

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, or runs out of memory or out of time (details below), **you will get zero points**. Anything you write to stdout or stderr will be ignored and is ok to leave in the code you submit (but it will likely slow you down). Please test your program with the provided sample files to avoid any problem.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe

you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

> **Ok** to use AI such as Cursor, Antigravity, ChatGPT, etc
> **Do not copy** code or written material from another student. Even single lines of code should not be copied.
> **Do not collaborate** on this assignment. The assignment is to be solved individually.
> **Do not copy** code off the web. This is easier to detect than you may think.
> **Do not share** any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
> **Do not copy** code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homeworks of previous years.
> **Do not ask on piazza** how to implement some function for this homework, or how to calculate something needed for this homework.
> **Do not post code on piazza** asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
> **Do not post test cases on piazza** asking for what the correct solution should be.
>
> **Do** ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project description

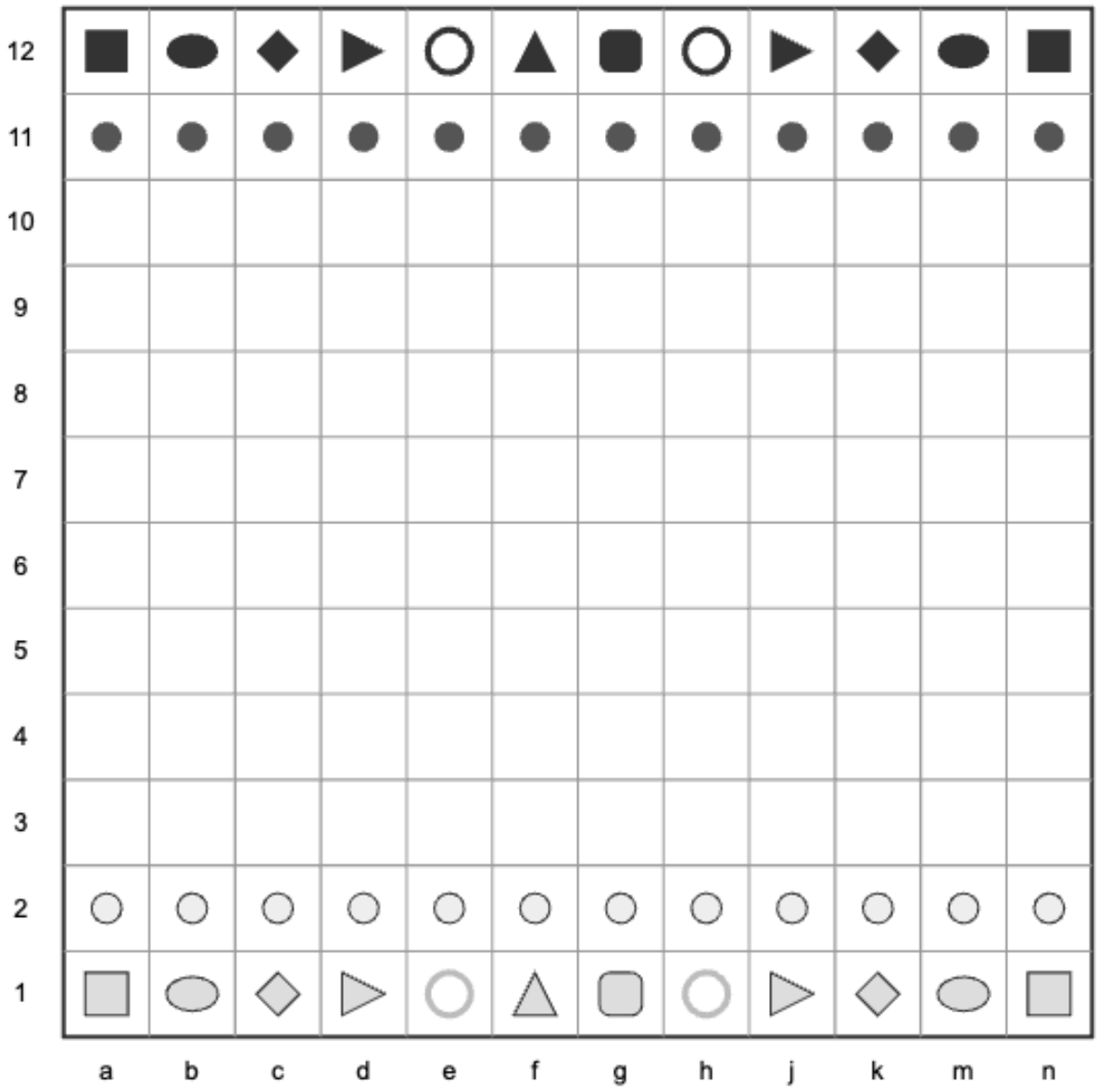
In the realm of inheritance, power is fragile.

The old rulers are gone. What remains are heirs, siblings, tutors, and guards — all too young to rule alone, yet powerful together. Victory does not come from domination, but from careful movement, cooperation, and the fall of a single Prince.

Welcome to ***Heirs***.

Game rules

Heirs is a two-player board game on a 12x12 board. Each player is assigned a color, Black or White. White always gets the opening (first) move. The game pieces and initial board state are as follows:



Pay particular attention to the letters that designate the columns: both “i” and lowercase “L” are skipped to eliminate possible ambiguities. Your code must account for this.

General Rules and Pieces

1. Overview

Heirs is a two-player strategy board game played on a 12×12 board. Each player controls a youthful royal household. The goal is simple: **capture your opponent's Prince**.

There are no special-case rules, no hidden exceptions. This is unlike other board games you may know (e.g., promoting a piece that reaches the opposite end of the board – this does not happen in Heirs).

2. Components

- 12×12 square board
- 24 pieces per player (Black and White)

3. Setup

Each player places:

- **12 Babies** on the front row
- **12 Major Pieces** on the back row

As shown above.

White moves first.

4. Turn Structure

- Players alternate turns
- On your turn, move exactly one piece
- Captures occur by moving onto an occupied enemy square

5. Winning the Game

The game ends immediately when a **Prince** is captured (more details on game over rules later).

6. Piece Guide & Movement

All descriptions and figures below are from the perspective of the White player, which starts with its pieces at the bottom of the board. When we say "forward" below, this means up for the White player (and, conversely, down for the Black player). The graphics shown below should be flipped vertically for the Black player.

Baby (12 per side) - We have all been one of them

- Moves 1 or 2 squares straight forward per turn.
- May choose either distance every turn.
- Cannot move backward.
- Cannot jump over pieces.
- Captures straight forward.

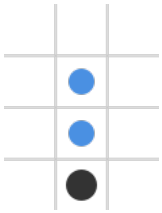


Figure 1: Baby movement. Current location of a White baby is shown as the black disk. The blue disks represent allowed positions that this Baby may move to.

Prince (1 per side) - The most important piece

- Moves 1 square in any direction.

- If captured, the game ends.

- It is ok for the Prince to be in the line of capture of opponent pieces: the Prince is, in principle, not obligated to move away. The only thing that matters in the end is whether the Prince is captured. Of course, if a Prince is in the line of capture of an opponent piece and does nothing about it, likely that Prince will lose the game on the next turn…

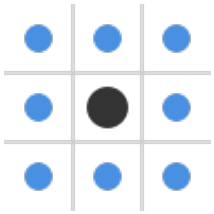


Figure 2: Prince

Princess (1 per side) - Short-range ruler

- Moves 1 - 3 steps in any direction.

- Cannot jump over pieces.

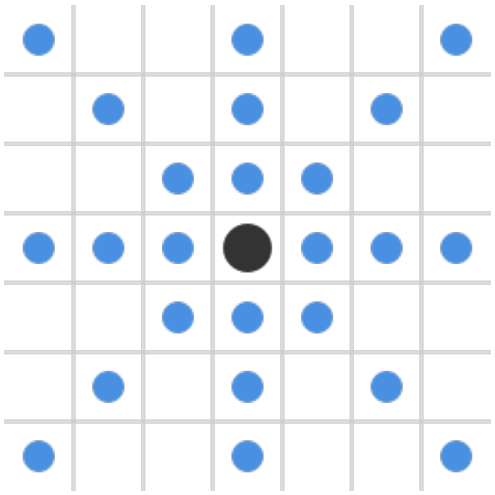


Figure 3: Princess

Pony (2 per side) - Short-leaping attacker

- Moves exactly one step in any diagonal direction.

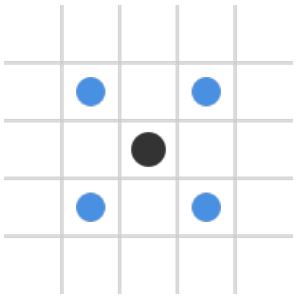


Figure 4: Pony

Guard (2 per side) - Straight-line attacker

- Moves up to 2 squares horizontally or vertically.
- Cannot jump over pieces.

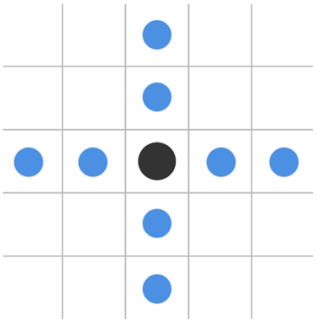


Figure 5: Guard

Tutor (2 per side) - Careful diagonal guide

- Moves up to 2 squares diagonally.
- Cannot jump over pieces.

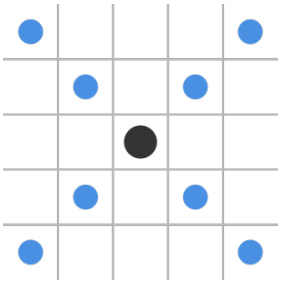


Figure 6: Tutor

Scout (2 per side) - Forward-focused striker

- Moves 1 - 3 squares forward; may also shift by 1 sideways while advancing.
- Jumps over pieces allowed.
- At most one capture may occur, on the landing point, and with no further movement allowed.

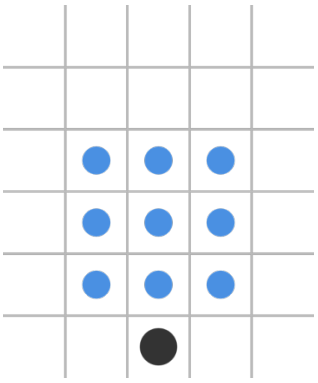


Figure 7: Scout

Sibling (2 per side) - Cooperative fighter

- Moves 1 step in any direction, but must end adjacent (in any of the possible 8 directions) to at least one friendly piece.

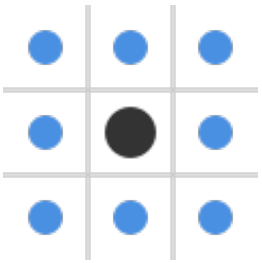


Figure 8: Sibling movement, but a destination location is only allowed if it is adjacent (horizontally, vertically, or diagonally) to one or more existing friendly pieces.

7. Quick Reference Sheet

Piece Summary and Notations Used For Programming

White Piece and Black Piece below denotes the symbol we will use in the board to represent each piece. Blank grid cells in the board will be represented by a dot ".", as further shown below. Generally speaking, White pieces are represented by capital letters, and Black pieces by lowercase letters.

Piece	Move	White Piece	Black Piece
Baby	1-2 forward	B	b
Prince	1 any direction	P	p
Princess	1-3 any direction	X	x
Pony	1 diagonal	Y	y
Guard	1-2 orthogonal	G	g
Tutor	1-2 diagonal	T	t
Scout	1-3 forward (+1 sideways)	S	s
Sibling	1 any direction, adjacent ally required	N	n

Design Philosophy

- Large board, short moves
- Youthful hierarchy
- No special-case rules
- Tactical positioning over raw power

Designer Balance Notes

- The 12x12 board increases spatial tension while shorter movement limits power.
- Babies advancing up to 2 squares maintains tempo without first-move exceptions.
- However, babies advancing too fast may get stuck on the other end of the board and will become useless.
- Princess is deliberately capped at 3 to avoid dominance.
- Pony jump range keeps tactical surprise without board-wide threats.
- Sibling encourages formation play and discourages lone attacks.
- No check/checkmate reduces rules overhead and encourages bold play.

Enjoy **Heirs**.

Playing with agents, grading

In this homework, your agent will play against another agent, either implemented by the TAs, or by another student in the class.

For grading, your agent will play against two different agents implemented by the TAs. 10 full games will be against a ***random agent*** (this should be easy for your agent to beat), and another

10 full games will be against a ***simple minimax agent*** with no alpha-beta pruning. There will be a limited total amount of play time available to your agent for the whole game (e.g., 300 seconds), so you should think about how to best use it throughout the game. This total amount of time may vary from game to game. Your agent must play correctly (no illegal moves, etc.) and beat the reference agents to receive 5 points per game. Your agent will be given the first move on 10 of the 20 games. The conditions for determining a winner when the game is over are detailed below. While playing games, you should think about how to divide your remaining play time across possibly many moves throughout the game.

In addition to grading, we will run a competition where your agent plays against agents created by the other students in the class, with a prize. This will not affect your grade, but it would look very good on your Resume if you finish in the top 10, or are the grand winner!

Game setup and initial state

The setup of the game is as follows:

- Each player plays as White or Black. This will alternate between different games.
- The board consists of a 12x12 grid of squares.
- Before the game starts, the board contains 24 pieces exactly as shown below.
- **White always opens the game.**
- Each player starts the game with a given amount of time (e.g., 300 seconds). As we run an agent, we will measure the time taken on each move and subtract it from that agent's total remaining time. If agent runs out of time, it loses the game irrespective of the number of pieces on the board.

File formats

Input: The file input.txt in the current directory of your program will be formatted as follows:

First line: Player that your agent will play on this turn: **WHITE** or **BLACK.** During a given game, your agent will always play either WHITE or BLACK. But in the 20 games used for grading, your agent will play WHITE 10 times, and BLACK 10 times.

Second line: Two floating point numbers separated by one space to indicate **your remaining play time in seconds**, and your **opponent's remaining play time**. The precision of these numbers is potentially the full precision of double (e.g., 12.3456789).

Next 12 lines: Current state of the board, one row per line. Each line will contain one string of 12 symbols (without any space in between) and an end-of-line (LF) marker, where:

- **B, P, X, Y, G, T, S, N** denotes a cell occupied by a White piece
- **b, p, x, y, g, t, s, n** denotes a cell occupied by a Black piece
- **.** denotes an empty cell.

For example, here is the input.txt for the opening move:

```
WHITE
300 300
gytsnxpnstyg
bbbbbbbbbbbb
............
............
............
............
............
............
............
............
BBBBBBBBBBBB
GYTSNXPNSTYG
```

You are guaranteed that the format of input.txt will be correct (no missing lines, no missing grid cells, etc). You are also guaranteed that **at least one legal move is possible for your player**, i.e., you do not have to worry about pass. Just beware that sometimes the time remaining may show up as an integer if it just so happens that a player has an exact number of seconds of remaining time (as shown with the figure 300 in the above file).

Output: The file output.txt which your program creates in the current directory should be formatted as follows:

First line: Your chosen move, as "<source column><source row><SPACE><destination column><destination row>" using the column and row numberings shown on page 3. Remember to pay particular attention to the letters that designate the columns: both "i" and lowercase "L" are skipped to eliminate possible ambiguities. Your code must account for this.

For example, output.txt may contain:

```
c9 d11
```

Game over tests

- Game is over when a prince is captured, and the captured prince loses the game.

- If an agent runs out of time (its remaining time after a turn becomes negative), then that agent immediately loses the game.

- If an agent produces an illegal move in its output.txt, or no output.txt, then that agent immediately loses the game.

Draw conditions are as follows and will be detected by the game engine that invokes your agent. You do not need to detect a draw condition in your code. If a draw condition occurs, we will end the game and not call your agent anymore.

- A draw occurs if an agent does not have any possible remaining moves.

- A draw occurs if the exact same configuration of the whole board (complete state of the board) occurs three times, with the same player to move.

- A draw is declared if 50 consecutive moves pass without a capture or Baby move.

Scoring

- If an agent wins a game by any of the win/lose rules above, it gets 1 point.

- In case of a draw for any of the above reasons:
    - The agent with the most remaining time wins, or
    - If both remaining times are equal, the agent with the most remaining pieces wins.

At the end of 10 games between two agents, the one with the most points wins. If both agents end up with the same number of points, they both advance to the next round of the competition.

Agent vs agent games

Playing against another agent will be organized as follows (both when your agent plays against the reference minimax agent, or against another student's agent):

A master game playing engine will be implemented by the grading team. This engine will:

- Create the initial board setup according to the above description.
- Assign a player color, Black or White, to your agent. The player who gets assigned White will always have the first move.
- Then, in sequence, until the game is over:
    - The game playing engine will create an input.txt file which contains the current board configuration, which color your agent should play, and how much total play time your agent has left, as described above.
    - We will then run your agent. Your agent should read input.txt in the current directory, decide on a move, and create an output.txt file that describes the move, as shown above. Your time will be measured (total CPU time). If your agent does not return before your remaining time is over, it will be killed and it loses the game.
    - Your remaining playing time will be updated by the game engine, by subtracting the time taken by your agent on this move. If time left reaches negative, your agent loses the game.

- The validity of your move will be checked. If the format of output.txt is incorrect or your move is invalid according to the rules of the game, your agent loses the game.
- Your move will be executed by the game playing engine. This will update the game board to a new configuration.
- The game playing engine will check for a game-over or draw condition. If one occurs, the winning agent will be declared using the game over and draw tests and scoring described above.
- The game playing engine will then present the updated board to the opposing agent and let that agent make one move (with the same rules as just described for your agent; the only difference is that the opponent plays the other color and has its own time counter).
- Game continues until an end condition is reached.

Hence:

1) You do not have to worry about game over or draw conditions. Your agent will not even be called if you cannot make at least one legal move. But you should still be aware of these conditions so that you try to avoid them if possible.
2) You should not worry about updating your remaining time. The game engine will measure the time taken by your agent on each move, and will update your remaining time for you (details below).
3) You do not need to let us know what the board state will be after your move. The game engine will compute that, including all captures. Just let us know in output.txt what move you want to play, as described above.
4) The game engine will not end the game unless a game over or draw condition exists. Thus, it is possible that your agent will be called many times during a game. You should be prepared for that and not time out!

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose ("py" for python, "cpp" for C++17, and "java" for Java).
- Likely (but not guaranteed), total play time for each agent will be 5 minutes (300.0 seconds) when playing against another agent.
- Play time used on each move is the total combined CPU time as measured by the Unix **`time`** command. This command measures pure computation time used by your agent, and discards time taken by the operating system, disk I/O, program loading, etc. Beware that it cumulates time spent in any threads spawned by your agent (so if you run 4 threads and use 400% CPU for 10 seconds, this will count as using 40 seconds of allocated time). Hence, there is no benefit in using multi-threading in this homework, it will only slow you down because of various synchronization primitives (mutex, semaphore, etc). Beware

that in Python and Java, some libraries are already multi-threaded. So, you need to also use the `time` command to correctly measure your runtime during development. For example, your agent may run in 3.2s of wall-clock time, but, if it make many calls to a matrix-vector multiplication which is parallelized behind the scenes, that may consume more than 100% CPU by using several cores, and you may end up seeing your time reduced by a larger amount than 3.2s.

- If your agent runs for more than its given play time (given in input.txt), it will be killed and will lose the game.
- You need to think and strategize how to best use your allocated time. In particular, you need to decide on how deep to carry your search, on each move. In some cases, your agent might be given only a very short amount of time (e.g., 5.2 seconds, or even 0.01 seconds), for example towards the end of a game. Your agent should be prepared for that and return a quick decision to avoid losing by running over time. The amount of play time that will be given in input.txt will always be >0, but it could be very small if you are close to running out of time.
- To help you with figuring out the speed of the computer that your agent runs on, you are allowed to also provide a second program called **calibrate.xxx** (same extension conventions as for homework.xxx). This is optional. If one is present, **we will run your calibrate program once (and only once)** before we run your agent for grading or against another agent. You can use calibrate to, e.g., measure how long it takes to expand some fixed number of search nodes, or to benchmark the CPU speed in any other way you like. You can then save this into a single file called **calibration.txt** in the current directory. When your agent runs during grading or during a game, it could then read calibration.txt in addition to reading input.txt, and use the data from calibration.txt to strategize about search depth or other factors. Please aim for no more than **5 seconds** to run your calibrate program. A few seconds is usually enough to get a good estimate of the CPU speed (e.g., expand 1,000 nodes, or compute your eval function on 1,000 hardwired board configurations).
- You need to think hard about how to design **your eval function** (which gives a value to a board when it is not game over yet). Hint: Different pieces may have different values. Different positions on the board too.
- You are allowed to maintain persistent data across moves during a game, by writing such data to a single file called **playdata.txt** in the current directory. Before a new game starts, the game playing engine will delete any playdata.txt file. So, on your first move, this file will not exist, and you should be prepared for that. Then, you can write some data to that file at the end of a move and read that file back at the beginning of the next move.
- The random agent created by the TAs is not fully random: on every move, it randomly selects one of the available *legal* moves. So, you should not expect that it will lose just by selecting invalid moves. The minimax TA agent will not use alpha-beta and will likely be capped at a low lookahead depth; but it will do adaptive depth choice on every move to avoid running out of time (e.g., use depth 3 when >50s remain, depth 1 when < 3s, otherwise depth 2).
- No Internet connection will be available, so you cannot rely on external agents or APIs.
- No GPU will be available on the machines used for grading.

- No external libraries except standard ones already on vocareum (e.g., numpy), no linker flags, no Makefile, no CMakeLists.txt, but you are allowed to add some libraries from source, as follows (for C++): you include the headers and also the C++ implementation in your file. This may allow you to use some lightweight DNN frameworks if you want to.

homework.cpp:

```
#include <iostream>
#include <library_header1.hpp>
#include <custom_header2.hpp>

int main()
{
  // read input.txt
  // decide next move
  // write output.txt
}

// include C++ implementation files here,
// they will be compiled with your program:
#include <library_implementation1.cpp>
#include <custom_implementation2.cpp>
```