Description

In this assignment you will take what you have learned about modeling problems with classes and design patterns, and build an application to play the game Santorini. You will design the class structure yourself, using design patterns to implement the game logic. Then you will write simple AI players.

Santorini is an abstract strategy game (think checkers/chess) designed by Dr. Gordan Hamilton. The rules of the game are very simple, so you can spend most of your time thinking about design rather than implementing complex internal logic.

Rules summary

Here is a PDF of the official rules.

We will only implement the basic game described on the first page. We will skip the god powers described in pages 3-5. It's an interesting exercise to think about how you could handle some of their interesting rules twists in a modular design, but it is out of scope for this assignment.

We will simplify the setup rules a bit. We are only concerned with the 5x5 grid, and we will put the workers in default starting positions (shown below), so that the players do not have to pick where to start.

Board representation

The physical game has lovely pieces designed to mimic the iconic architecture of the real island of Santorini, but you will implement an ASCII representation so that the game can be played through a command line interface (CLI).

```
+--+--+--+--+--+
|0 |0 |0 |0 |0 |
+--+--+--+--+--+
|0 |0Y|0 |0B|0 |
+--+--+--+--+--+
|0 |0 |0 |0 |0 |
+--+--+--+--+--+
|0 |0A|0 |0Z|0 |
+--+--+--+--+--+
|0 |0 |0 |0 |0 |
+--+--+--+--+--+
```

This is the starting setup. The 5x5 board is viewed top down. A number in each space represents the level of the building. They all start unbuilt at level 0. A complete tower with a blue dome on top is shown as level 4. Next to the level number there is either a space, or the symbol for one of the 4 workers. The white player controls the A and B workers, while the blue player controls Y and Z.

Move representation

On a player's turn they indicate through the CLI which worker they wish to move, which cardinal direction they wish to move, and which cardinal direction they wish to build (from the new space). For example, the white player chooses to move worker A north, then build in the space they just left.

```
Select a worker to move
A
Select a direction to move (n, ne, e, se, s, sw, w, nw)
n
Select a direction to build (n, ne, e, se, s, sw, w, nw)
s
+--+--+--+--+--+
|0 |0 |0 |0 |0 |
+--+--+--+--+--+
|0 |0Y|0 |0B|0 |
+--+--+--+--+--+
|0 |0A|0 |0 |0 |
+--+--+--+--+--+
|0 |1 |0 |0Z|0 |
+--+--+--+--+--+
|0 |0 |0 |0 |0 |
+--+--+--+--+--+
```

Running your code

Your code should take in arguments from the command line to configure the types of players and whether undo/redo and score display should be enabled or not. Note that you will want history off when running two computers against each other so that you don't have to type next every turn. Up to four arguments may be passed in to `main.py` from the command line and you cannot give an argument without supplying the ones before it. The arguments that are omitted from the end will have the default values. Therefore `python3 main.py random` is equivalent to `python3 main.py random human off off`

arg	meaning	allowed values	default
argv[1]	white player type	human, heuristic, random	human
argv[2]	blue player type	human, heuristic, random	human
argv[3]	enable undo/redo	on, off	off
argv[4]	enable score display	on, off	off

Requirements:

- **10 points.** Construct 5x5 board and set it up as described above. Begin the program by printing it to stdout.
- **35 points.** Enable two human players (or one playing both sides) to play the game via command line inputs. At the start of each turn, display the board, the turn number, the current player, and possibly a score (see below). A player is prompted to select a piece, move direction and build direction as described above. Invalid moves and builds should not be allowed by your program. See the sample input/outputs at the end of the assignment.
    - If a user enters anything other than A, B, Y, or Z, print `Not a valid worker` and repeat the prompt.
    - If a user selects an opponent's piece, print `That is not your worker` and repeat the prompt.
    - If a user enters anything other than n, ne, e, se, s, sw, w, nw, print `Not a valid direction` and repeat the prompt.
    - If a user enters a valid direction that the worker cannot move into, print `Cannot move <direction>` where <direction> was the user's selection and repeat the prompt.
    - If a user enters a valid direction that the worker cannot build on, print `Cannot build <direction>` where <direction> was the user's selection and repeat the prompt.
- **10 points.** At the start of a turn, check if the game has ended. If a player has a worker on a level 3 building, they win. This is a slight change from the official rules that say you win instantly if your worker moves onto level 3 of a building, but it doesn't make a difference since you can always at least build in the space you moved out of. If neither of the current player's workers are able to make a move and build then that player loses. This involves enumerating all possible moves for the current player, which will also help in the next part. When the game ends, simply print `white has won` or `blue has won` and exit the program.
- **10 points.** Write two types of basic computer opponents to play against. Create a **Random** computer player that will randomly choose a move from the set of allowed moves. Two bots should be able to play against each other, in which case it will run through all the turns without prompting for additional input, until the game ends. The computer players should print the move they took, so you can see what they did. This includes the worker that moves, the move direction, and the build direction. For example, the move shown earlier should be printed as `A,n,s`.
- **15 points.** Create a **Heuristic** computer player that will assess the available moves and choose the one it thinks is best based on a few criteria. This player does not need to look multiple turns ahead to make this decision. This isn't an AI class, so we'll keep this pretty simple. We are primarily concerned with good software design for players that can take different strategies, but this would be a good start for a fun AI project on your own time. The three criteria are the height of your workers, how close to the middle they are, and how far they are from the opponent's workers.

  - It is better to get workers higher so they can try to win, but also because it is easier to move down than up. `height_score` is the sum of the heights of the buildings a player's workers stand on.
  - Having workers near the middle of the board also gives more flexibility. We value the center space as 2, the ring around the center as 1, the edge spaces as 0, and we add these values for each of a player's workers to get the `center_score`.
  - Having a worker near your opponents can help stop them from winning by building a dome. `distance_score` is the sum of the minimum distance to the opponent's workers. So for blue, it would be min(distance from Z to A, distance from Y to A) + min(distance from Z to B, distance from Y to B). For example, if Z was right next to A and Y was right next to B, then blue's distance_score is 2. If Z is adjacent to both A and B, but Y is further away from both, then the distance_score is still 2. Since we want higher scores to be better, we then subtract from 8 (the maximum distance), so in both of the previous examples, the final distance_score becomes 6.
  - These can be combined into a `move_score` with some weight on each individual score as shown below. Unless you're an expert player, it's hard to say how each of these should be weighted (even then it may be unclear). Training an AI could aim to optimize these weights. Feel free to choose your own and test it out, but you can start with something like c1=3, c2=2, c3=1.
    - `move_score = c1*height_score + c2*center_score + c3*distance_score`
  - Your **Heuristic** player should look at each available move, calculates a move_score, and pick the highest one, breaking any ties randomly.
  - If a move would put a worker at height 3, this would win the game, so instead of the normal move_score, it should be made infinitely large to ensure that it is chosen.
  - In order for you to debug your heuristic scores and for us to verify that it is working correctly, we will use a command line argument to turn on printing the current score each turn in the format `(height_score, center_score, distance_score)`. See the examples below. This is not the score for the move that was just taken, but rather the score for the current player before taking their next move. This gives an approximation of who is winning, but also makes it easier to look at the board to confirm the scores are calculated correctly.
- **20 points.** Implement undo/redo functionality. When enabled with the history command line argument, the game should give the options `undo`, `redo`, or `next` before every turn (including AI turns). Undo should roll back to the previous game state. Redo reverses the most recent undo. Therefore, you can undo and redo multiple times. Undo does nothing if at turn 1 and redo does nothing if it is already the latest turn. After using undo, taking any new turn should invalidate any turns that could have been redone. To continue taking turns as usual, the user should enter `next`.
- **5 points.** Create a UML diagram for your program demonstrating the relationships between the classes you created.

- **15 points.** Submit a brief write-up (upper limit of 600 words) describing how you made use of design patterns we talked about in class. I expect you will use at least 4 distinct design patterns (and maybe some more than once), but you're welcome to include more as long as they are appropriate to the task. Iterator counts, but only if you have created your own custom iterator that does something more than just stepping through a list.

Guidance

1. There is no code structure predefined for this assignment because this is an opportunity to practice design techniques. If you find yourself stuck getting started then please come to office hours to talk through the design process.
2. Think about modularity and abstraction in your design. What parts are specific to the rules of the game and keep those encapsulated. The rest can act like it is any 2-player game played in turns with pieces on a grid board with full information. That's a lot to have in common as far as games go, but there are many in this category, such as checkers, chess, go, reversi, and tak.
3. Your main file should take in the command line args, then use them to set up a manager object that will drive the core game loop.
4. I recommend working through the requirements above in order, since they build on one another.
5. Having two computer players play each other repeatedly may help you catch obscure bugs.

Testing

You are strongly encouraged to write unit tests for your code. Even some simple test cases written early on (setting up the board, printing it out) will be helpful as you go. However, we will not specifically check your tests in this assignment.

Submissions

Submit your project to gradescope with whatever files are necessary to run `main.py`. In the same submission, include your write-up and UML diagram in PDF format. If you worked with a partner, do not forget to add the additional individual statements.

Some public test cases will be added to Gradescope soon to give you some sanity checks particularly for formatting. You can expect blackbox testing that checks whether the interactions work properly, and sets up specific scenarios in the game to check that rules are enforced correctly.