Project Part B

Playing the Game

COMP30024 Artificial Intelligence

April 2024

1 Overview

In this second part of the project, you will write an agent program to play the full two-player version of Tetress. Before you read this specification it is a good idea to re-read the 'Rules for the Game of Tetress' document to re-familiarise yourself with the rules. A link to this as well as a browser based app of the game can be found on the LMS where you found this document.

The aims for Project Part B are for you and your project partner to (1) practice applying the game-playing techniques discussed in lectures and tutorials, (2) develop your own strategies for playing Tetress, and (3) conduct your own research into more advanced algorithmic game-playing techniques; all for the purpose of creating the best Tetress–playing program the world has ever seen!

Similar to Part A, you can (and should) regularly submit to Gradescope as a means to get immediate feedback on how you are progressing. The autograder will be equipped with simple (not particularly clever) "test" opponent. See the *Submission* section at the end of this document for details.

Both you and your partner are expected to read this specification *in full* before commencing the project, then at your earliest convenience, you should both meet up and come up with an action plan for tackling it together (see Section 4.2 for our expectations regarding teamwork).

1.1 The Task

Your task is twofold. Firstly, you will **design and implement an agent program** to play the game of Tetress. That is, given information about the evolving state of the game, your program will decide on an action to take on each of its turns (we provide a "referee" program to coordinate a

game of Tetress between two such programs so that you can focus on implementing the game-playing strategy). Section 2 describes this programming task in detail, including information about how the referee program will communicate with your agent program and how you can run the referee program.

Secondly, you will **write a report** discussing the strategies your program uses to play the game, the algorithms you have implemented, and other techniques you have used in your work, highlighting the most impressive aspects. Section 3 describes the intended structure of this document.

The rest of this specification covers administrative information about the project. For assessment criteria, see Section 4. For submission and deadline information see Section 5. Please seek our help if you have any questions about this project.

2 The program

You have been given a template **Python 3.12** program in the form of a **module** called `agent`. Alongside this module is the "driver" module named `referee`, which is what is used in the submission (and tournament) environment to verse two agents against other and enforce the rules of the game. We've given this to you so you can test your agent locally, but it's also a good idea to make periodic submissions to Gradescope like you did in Part A of the project. We have provided a simple (not very clever) agent that you can playtest your work against in this environment.

Before continuing, download the template and follow the "Running the template code" guide on the assignment LMS page. Once your local development environment is set up, try running the command `python -m referee agent agent` This will play the template `agent` module against itself (naturally this will result in a failed game as it's not implemented yet!).

Further details regarding how to use the `referee` module, how it interacts with your game playing agent(s), as well as the high level process involved in playing a game are specified in the following subsections. It is important you read these carefully to make the most of what we have provided you and hence minimise wasted effort.

2.1 The `Agent` class

Within the `agent` module that comes with the template project you will find a `program.py` file inside that defines a Python class called `Agent`. This class should not be instantiated directly, rather, the methods of this class are invoked by the `referee` throughout a game of Tetress and hence serve as an interface for your agent to play the game.

The `Agent` class defines the following three methods which you must implement:

1. `def __init__(self, color: PlayerColor, **referee: dict):` Called once at the beginning of a game to initialise your player. Use this method to set up an internal representation of the game state.

   The parameter `color` will be `PlayerColor.RED` if your program will play as Red, or the string `PlayerColor.BLUE` if your program will play as Blue. Note that that the `PlayerColor enum` is imported from the `referee.game` module – you will see numerous types like this in the template. We discuss the `**referee` param later on, as this is common to all methods.

2. `def action(self, **referee: dict) -> Action:` Called at the beginning of your agent's turn. Based on the current state of the game, your program should select and return an action to play. The action must be represented based on the instructions for representing actions in the next section.

3. `def update(self, color: PlayerColor, action: Action, **referee: dict):`
   Called at the end of each player's turn, *after* the referee has validated and applied that player's action to its game state. You should use this method to update your agent's internal representation of the game state so it stays in sync.

   The parameter `player` will be the player whose turn just ended, and `action` will be the action performed by that player. If it was your agent's turn that just ended, `action` will be the same action object you returned through the `action` method. You may assume that the `action` argument will always be valid since the referee performs validation before this method is called (your `update` method does **not** need to validate the action against the game rules).

Provided that you follow the above interface, it is possible to define multiple agent classes with different modules/names and play them against each other. This is helpful for benchmarking and comparison purposes as you refine your work.

You may optionally use the `referee` parameter in these methods (strictly speaking this parameter represents keyword arguments as a dictionary, and may be expanded if desired). It contains useful metrics passed from the referee, current as of the **start** of the method call:

- `referee["time_remaining"]`: The number of seconds remaining in CPU time for your agent instance. If the referee is not configured with a time limit, this will be equal to `None`.
- `referee["space_remaining"]`: The space in MB still available for use by your agent instance, otherwise `None` if there is no limit or no value is available. This will only work if using the "Dev Container" method to work on your project (or otherwise use a Linux based system).
- `referee["space_limit"]`: This is a **static** space limit value available on any system. It might be handy to have in the `__init__(...)` method if you pre-compute any very large data structures. If no limit is set in the referee, it will equal `None`.

2.2 Representing actions

To construct actions, you should use the `dataclass` definitions in `referee/game/actions.py` as well as `referee/game/coord.py`. You should already be familiar with these structures from Part A of the project. This time, instead of generating a list of actions you should return just one `PlaceAction` object from the aforementioned `action` method:

```
PlaceAction(Coord(r1, c1), Coord(r2, c2), Coord(r3, c3), Coord(r4, c4))
```

The four `Coord` arguments are coordinates on the game board and must be adjacent to each other, representing a valid tetromino as per the game rules. The referee will also use this representation when notifying your agent of the last action taken (i.e., when calling the `update` method).

2.3 Playing a game

To play a game of Tetress with your `agent` module, we provide a "driver" program – a Python module called `referee` which sits alongside it in the template.

You don't need to understand exactly how the referee works under the hood (suffice to say parts of it are quite complex), however, it's important that you are aware of the high-level process it uses to orchestrate a game between two `agent` classes, summarised as follows:

1. Set up a Tetress game and create a sub-process for each player's agent program. Within each sub-process, instantiate the specified agent classes for each of Red and Blue, as per the command line arguments (this calls their `.__init__()` methods). Set the **active player** to Red, since they always begin the game as per the rules.
2. Repeat the following until the game ends:
   (a) Ask the **active player** for their next action by calling their agent object's `.action(...)` method.
   (b) Validate the action and apply it to the game state if is allowed, otherwise, end the game with an error message. Display the resulting game state to the user.
   (c) Notify *both* agent objects of the action by calling their `.update(...)` methods.
   (d) Switch the **active player** to facilitate turn-taking.
3. After detecting one of the ending conditions, display the final result of the game to the user.

To play a game, the referee module (the directory `referee/`) and the module(s) with your `Agent` class(es) should be within your current working directory (you can type `ls` within your terminal to confirm this). You can then invoke the `referee`, passing the respective modules as follows:

```
python -m referee <red module> <blue module>
```

...where `<red module>` and `<blue module>` are the names of the modules containing the classes to be used for Red and Blue, respectively. The referee comes with many additional options to assist with visualising and testing your work. To read about them, run '`python -m referee --help`'.

Avoid modifying the `referee` module as this risks inconsistencies between your local environment and the assessment environment (Gradescope). An original copy of the `referee` is used on Gradescope which means any modifications you make to it will not apply during assessment, even if uploaded with your submission.

2.4 Program constraints

The following **resource limits** will be strictly enforced on your program during testing. This is to prevent your agent from gaining an unfair advantage just by using more memory and/or computation time. These limits apply to each player agent program for an entire game:

- A maximum computation time limit of **180 seconds per player, per game**. This is measured in accumulated CPU time across the span of the game, though there is also a hard "wall clock" timeout of the same duration for any given action (this is to handle cases where an agent gets stuck in an excessively long computation or infinite loop).
- A maximum ("peak") memory usage of **250MB per player, per game**, not including any imported libraries mentioned in Section 2.5.

You **must not** attempt to circumvent these constraints. Do not use multiple threads or attempt to communicate with other programs/the internet to access additional resources. Saving to and loading from disk is also prohibited.

For help measuring or limiting your program's resource usage, see the referee's additional options (`--help`). Note that memory usage can only be tracked locally when running the `referee` in the given Dev Container (or another Linux based system).

2.5 Allowed libraries

Your program should use **only standard Python libraries**, plus the optional third-party library **NumPy**. With acknowledgement, you may also include code from the AIMA textbook's Python library, where it is compatible with Python 3.12 and the above limited dependencies. Beyond these, **your program should not require any other libraries in order to play a game**.

However, while you develop your agent program, you are free to use other tools and/or programming languages. This is all allowed **only if** your `Agent` class does not require these tools to be available when it plays a game.

For example, let's say you want to use machine learning techniques to improve your program. You could use third-party Python libraries such as scikit-learn/TensorFlow/PyTorch to build and train a model. You could then export the learned parameters of your model. Finally, you would have to (re)implement the prediction component of the model yourself, using only Python/NumPy/SciPy. Note that this final step is typically simpler than implementing the training algorithm, but may still be a significant task.

3 The report

Finally, you must discuss the strategic and algorithmic aspects of your game-playing program and the techniques you have applied in a separate file called `report.pdf`.

This report is your opportunity to highlight your application of techniques discussed in class and beyond, and to demonstrate the most impressive aspects of your project work.

3.1 Report structure

You may choose any high-level structure of your report. Aim to present your work in a logical way, using sections with clear titles separating different topics of discussion.

Below are some suggestions for topics you might like to include in your report. Note that not all of these topics or questions will be applicable to your project, depending on your approach – that's completely normal. You should focus on the topics which make sense for you and your work. Also, if you have other topics to discuss beyond those listed here, feel free to include them.

- **Describe your approach:** How does your game-playing program select actions throughout the game?

  Example questions: What search algorithm have you chosen, and why? Have you made any modifications to an existing algorithm? What are the features of your evaluation function, and what are their strategic motivations? If you have applied machine learning, how does this fit into your overall approach? What learning methodology have you followed, and why? (Note that it is **not** essential to use machine learning to design a strong player)

- **Performance evaluation:** How effective is your game-playing program?

  Example questions: How have you judged your program's performance? Have you compared multiple programs based on different approaches, and, if so, how have you selected which is the most effective?

- **Other aspects:** Are there any other important creative or technical aspects of your work?

  Examples: algorithmic optimisations, specialised data structures, any other significant efficiency optimisations, alternative or enhanced algorithms beyond those discussed in class, or any other significant ideas you have incorporated from your independent research.

- **Supporting work**: Have you completed any other work to assist you in the process of developing your game-playing program?

  Examples: developing additional programs or tools to help you understand the game or your program's behaviour, or scripts or modifications to the provided driver program to help you more thoroughly compare different versions of your program or strategy.

You should focus on making your writing succinct and clear, as the overall quality of the report matters. The appropriate length for your report will depend on the extent of your work, and how novel it is, so aiming for succinct writing is more appropriate than aiming for a specific word or page count, though there is a hard *maximum* as described below.

Note that there's probably no need to copy chunks of code into your report, except if there is something particularly novel about how you have coded something (i.e., unique to your work). Moreover, there's no need to re-explain ideas we have discussed in class. If you have applied a technique or idea that you think we may not be familiar with, then it would be appropriate to write a brief summary of the idea and provide a reference through which we can obtain more information.

3.2 Report constraints

While the structure and contents of your report are flexible, your report must satisfy the following constraints:

- Your report **must not be longer than 6 pages** (excluding references, if any).
- Your report can be written using any means but **must be submitted as a PDF document**.

4 Assessment

Your team's Project Part B submission will be assessed out of 22 marks, and contribute 22% to your final score for the subject. Of these 22 marks:

- **11 marks** will be allocated to the performance of your **final** agent (you can only submit **one**, so pick your best if you developed a few agents).

  Marks are awarded based on the results of testing your agent against a suite of hidden 'benchmark' opponents of increasing difficulty, as described below. In each case, the mark will be based on the number of games won by your agent. Multiple test games will be played against each opponent with your agent playing as Red and Blue in equal proportion.

  5 marks available: Opponents who choose randomly from their set of allowed actions each turn, or use some form of weighted random distribution to pick moves.

  3 marks available: 'Greedy' opponents who select the most promising action available each turn, without considering your agent's possible responses (for various definitions of 'most promising').

  3 marks available: Opponents using any of the adversarial search techniques discussed in class to look an increasing number of turns ahead.

  The tests will run with **Python 3.12** on **Gradescope**. Programs that do not run in this environment will be considered incorrect and receive no marks for performance. Like in Part A of the project, **you should submit to Gradescope early and often** – you can already test your work against an agent which is live now. While the agent is not very clever, it reliably plays valid actions!

- **11 marks** will be allocated to the successful application of game playing techniques demonstrated in your work.

  We will review your report (and in some cases your code) to assess your application of adversarial game-playing techniques, including your game-playing strategy, your choice of adversarial search algorithm, and your evaluation function. For top marks, we will also assess your exploration of topics beyond just techniques discussed in class. **Note that your report will be the primary means for us to assess this component of the project,** so please use it as an opportunity to highlight your successful application of techniques. For more detail, see the following rubric:

  0–5 marks: Work that does not demonstrate a successful application of important techniques discussed in class for playing adversarial games. For example, an agent just makes random moves would likely get 0 marks.

  6–7 marks: Work that demonstrates a successful application of the important techniques discussed in class for playing adversarial games, possibly with some theoretical, strategic, or algorithmic enhancements to these techniques.

8–9 marks: Work that demonstrates a successful application of the important techniques discussed in class for playing adversarial games, along with *many* theoretical, strategic, or algorithmic enhancements to these techniques, possibly including some *significant* enhancements based on independent research into algorithmic game-playing or original strategic insights into the game.

10–11 marks: Work that demonstrates a *highly* successful application of important techniques discussed in class for playing adversarial games, along with *many significant* theoretical, strategic, or algorithmic enhancements to those techniques, based on independent research into algorithmic game-playing or original strategic insights into the game, leading to excellent player agent performance.

As per this marking scheme, it is possible to secure a satisfactory mark by successfully applying the techniques discussed in class. Beyond this, the project is open-ended. Every year, we are impressed by what students come up with. **However, a word of guidance:** We recommend starting with a simple approach before attempting more ambitious techniques, in case these techniques don't work out in the end.

Despite appearing simple on the surface, Monte Carlo Tree Search (MCTS) can be quite challenging to apply successfully. You should first develop a reliable "benchmark" agent to test against if you are planning to attempt this. Bear in mind there is no requirement for it to be used in order to secure a strong mark.

4.1 Code style/project organisation

While marks are not *dedicated* to code style and project organisation, you should write readable code in case the marker of your project needs to cross-check discussion in your report with your implementation. In particular, avoid including code that is **unused**. Marks may be indirectly lost if it's difficult to ascertain what's going on in your implementation as a result of such issues.

4.2 Teamwork

Part B of the project is to be completed in the same team of two as in Part A. Once again, both you and your partner are expected to contribute an equal amount of work throughout the entire duration of the project. While each person may focus on different aspects of the project, both should understand each other's work *in full* before submission (including all code).

Both partners are *also* expected to be proactive in communicating with each other, including meeting up early in the process and planning ahead. There will inevitably be deadlines in other subjects for one or both of you, and you'll need to plan around this (extensions won't be granted on this basis). Ensure that you set up regular ongoing meetings so that you don't lose track of what each person is doing.

We recommend using a code repository (e.g., on GitHub) to collaborate on the coding portion of the project. For the report, you may wish to use cloud based document editing software such as Google docs. This not only assists with keeping your work in sync and backed up, but also makes "auditing" easier from our end if there ends up being a dispute over contributions.

Where there is clear evidence that one person hasn't contributed adequately, despite their partner acting in good faith to collaborate with them as equals, individual marks will be awarded to better reflect each person's work.

In the event that there are teamwork issues, please **first** discuss your concerns with your partner *in writing* comfortably before the deadline. If the situation does not improve promptly, please notify us as soon as possible so that we can attempt to mediate while there is still time remaining (an email to the lecturers mailbox will suffice).

4.3 Academic integrity

Unfortunately, we regularly detect and investigate potential academic misconduct and sometimes this leads to formal disciplinary action from the university. Below are some guidelines on academic integrity for this project. Please refer to the university's academic integrity website [1] or ask the teaching team, if you need further clarification.

1. You are encouraged to discuss ideas with your fellow students, but **it is not acceptable to share code between teams, nor to use code written by anyone else**. Do not show your code to another team or ask to see another team's code.

2. You are encouraged to use code-sharing/collaboration services, such as GitHub, *within* your team. However, **you must ensure that your code is never visible to students outside your team**. Set your online repository to 'private' mode, so that only your team members can access it.

3. You are encouraged to study additional resources to improve your Python skills. However, **any code adapted or included from an external source must be clearly acknowledged**. If you use code from a website, you should include a link to the source alongside the code. When you submit your assignment, you are claiming that the work is your own, except where explicitly acknowledged.

4. If external or adapted code represents a significant component of your program, you should also acknowledge it in your report. Note that for the purposes of assessing your successful application of techniques, using substantial amounts of externally sourced code will count for less than an original implementation. However, it's still better to properly acknowledge all external code than to submit it as your own in breach of the university's policy.

[1] Link: academicintegrity.unimelb.edu.au

5. If you use LLM tools such as ChatGPT, these **must be attributed** like any other external source – you should state exactly how you've used them in your report (under "References"). Technology to detect use of such tools is constantly evolving, and we will endeavour to use what is available come marking (or even retrospectively) to detect *dishonest* use of it. We do, however, believe such tools can be useful when used in the context of proper understanding of a subject area – in short, use them responsibly, ethically, and be aware of their limitations!

5 Submission

The deadline is **11:00PM on Monday the 13th May, Melbourne time (AEST).** You may submit multiple times, but only the latest submission will be marked.

The procedure for submission via Gradescope is almost identical to that of Part A. Once again, remember to include your (unmodified) `team.py` file in the top level directory of your submission, and make sure you zip all files before uploading.

Note that only **one** team member needs to submit the final work. Once submitted, they must then link their partner to the submission within the Gradescope interface (top right corner). If both team members submit individually we will randomly pick one of the submissions to mark and link team members based on information in the `team.py` file. Projects won't be remarked if the wrong one was picked in such a scenario (i.e., if one was an old submission).

Here's how the file tree for your submission should look:

```
/
├── team.py ..................... The exact same file as the original you submitted
├── report.pdf ..................... Your report for this assignment (must be a PDF)
├── agent ..................... Your final agent program for this assignment
│   ├── __init__.py ..... The original file from the template unmodified
│   ├── program.py
│   └── ..... You may have other .py source files (optional)
└── other_agent(s) ..... You may include other agents, but we won't run these
```

It is **not** necessary to include the `referee` in your submission.

You may submit **multiple times** on Gradescope, and you are in fact **strongly encouraged** to do this early and often in order to test your work in the assessment environment. If you do make multiple submissions, we will mark the **latest** submission made.

Late submissions will incur a penalty of **two marks per day** (out of the 22 total marks allocated for this part of the project).

Extensions

If you require an extension, please send an email to `comp30024-lecturers@lists.unimelb.edu.au` using the subject 'COMP30024 Extension Request' at the earliest possible opportunity. If you have a medical reason for your request, you will be asked to provide a medical certificate. Requests for extensions received after the deadline will usually be declined.