Project Part A

Single Player Tetress

COMP30024 Artificial Intelligence

March 2024

1 Overview

In this first part of the project, you will write a program to play a simplified "single player" variant of Tetress. Before you read this specification, please make sure you have carefully read the entire 'Rules for the Game of Tetress' document on the LMS. Although you won't be writing an agent to play against an opponent just yet, you should be aiming to become familiar with the game rules, board layout and corresponding coordinate system.

The aims for Project Part A are for you and your project partner to (1) refresh and extend your Python programming skills, (2) explore some of the algorithms you have encountered in lectures, and (3) become familiar with the Tetress game environment and coordinate system. This is also a chance for you to develop fundamental Python tools for working with the game: Some of the functions and classes you create now may be helpful later (when you are building your full game-playing program for Part B of the project).

We will be using **Gradescope** as the testing environment when assessing your code. You can (and should) regularly submit to Gradescope as a means to get immediate feedback on how you are progressing. The autograder is already equipped with a couple of "visible" test cases for this purpose. 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 Single player Tetress

Single player Tetress is a simplified version of two-player Tetress. In this version of the game, you are provided with a game board which is already partially filled by Red and Blue tokens. The game proceeds as normal, except that only Red can play PLACE actions (Blue does not play). While the rules are effectively the same, the winning conditions are different since there is no opponent for Red to "block".

Specifically, Red is tasked with removing a single Blue token at some *target coordinate* on the board. They must exploit the "line removal" rule by completing the row or column that the target cell is situated on, thus removing it and securing a "win". Most importantly, the target token should be removed *in as few moves as possible*. In other words, an agent playing as Red should choose the *shortest* sequence of actions required to eliminate the target token.

There are a number of assumptions you should make when solving this problem:

1. In the given initial board state, there will be one or more Blue tokens on the board. Exactly one of these will be designated as the *target*. There will also be at least one Red token (you should assume it is not their first move of the game).
2. The *cost* of a solution is defined as the number of PLACE actions that must be played by Red to eliminate the *target* Blue token and therefore win the game.
3. If there is a *tie*, that is, there are multiple sequences of PLACE actions that attain a win for the **same** minimum *cost*, then any such sequence is considered an optimal solution.
4. There is no need to worry about turn limits or draws. Optimal solutions for the single player variant of the game shouldn't come anywhere near 150 actions.
5. It is possible for there to be no way for Red to win in some instances of the problem, depending on the given initial board state.

1.2 Your tasks

Using a search strategy discussed in lectures, your tasks are to:

1. **Develop and implement a program** that *consistently* and *efficiently* identifies the shortest sequence of actions required to win, given an arbitrary (valid) board state as its "input".
2. **Write a brief report** discussing and analysing the strategy your program uses to solve this search problem.

These tasks are described in detail in the following sections.

2 The program

You have been given a template **Python 3.12** program in the form of a **module** called `search`. Your task is to complete the `search()` function in `program.py`. You may write any additional functions/classes as needed, just ensure any additional `.py` files are kept within the same module.

When completed, `search()` should return a list of PLACE actions denoting the lowest cost win sequence, given an initial board state as input. Note that we have already supplied the necessary input/output code (in `__main__.py`), so you don't need to worry about this.

To streamline the running of test cases, you can use `<` to redirect a `.csv` file to the program via the standard input stream. More information regarding the test cases is provided in Section 2.5.

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 search < test-vis1.csv`

2.1 Program inputs

The `search(...)` function is given a Python dictionary as input, denoting the initial board state, as well as a *target coordinate*. Dictionary entries have the form `Coord(r, c): PlayerColor`, where:

- `r` and `c` denote the coordinate on the board for a cell, (r, c)
- `PlayerColor` is an enum which can be either:
  - `PlayerColor.RED` (cell contains a Red token), OR;
  - `PlayerColor.BLUE` (cell contains a Blue token)

This means you can look up the colour of a token in a cell using a coordinate structure instance `Coord(r, c)` as a key. Just keep in mind that not all cells are necessarily occupied (the dictionary is a sparse representation), so check that the key exists before using it.

The *target coordinate* is also a `Coord` structure, so it is possible to access the individual components as `target.r` and `target.c`, like any other coordinate structure instance.

The `Coord` and `PlayerColor` types are defined in the provided `core.py` file. You are welcome to reuse types defined in this file anywhere in your solution, but please don't modify them.

For the purposes of assessment, you may assume that all given input coordinates will be valid and that the *target coordinate* will always correspond to a Blue piece on the board.

2.2 Program outputs

The `search(...)` function must **return** the sequence of PLACE actions forming an optimal (minimal cost) solution to the given problem. This should be represented by a list of structures, each of the form `PlaceAction(Coord(r1, c1), Coord(r2, c2), Coord(r3, c3), Coord(r4, c4))`. The skeleton code comes with a hardcoded example showing how to do this.

If there is no way for Red to win for a given input scenario, you should instead return `None` from the function (**not** an empty list).

Please take care when printing anything to "standard output", as this is used for printing the actual result. All lines beginning with `$SOLUTION` will be taken to be part of the final action sequence (see `__main__.py` in the template).

2.3 Efficiency

While correctness and optimality matters first and foremost, you should also consider *efficiency* when designing your solution (think about different input scenarios as discussed in Section 2.5). Consider profiling your solution and compare approaches *in practice* before assuming a theoretical optimisation actually provides a noteworthy efficiency improvement. Similarly, be sure to consider the performance and memory usage of data structures you utilise. Constant factors can and do matter in some instances.

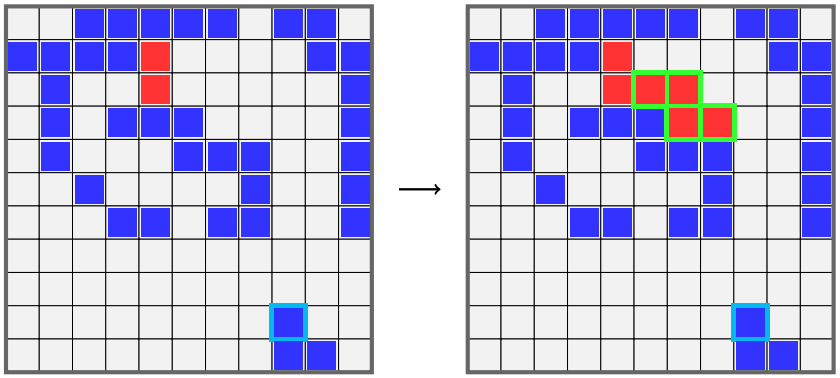
You are welcome to use any data structures that come with the template code as part of your solution, but do so critically and ensure they are appropriate for what you are trying to achieve.

2.4 Example

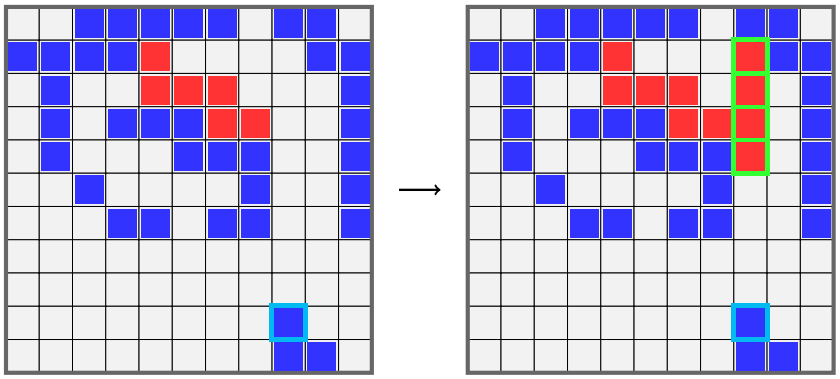
See Figure 1 for an example solution to the `test-vis1.csv` test case which comes with the template code. In this visualisation, the *target coordinate* $(9, 8)$ has been highlighted using a light blue outline. Observe how in three moves, Red is able to form a vertical line of tokens in column 8, thus removing the target token.

You might notice there are alternative action sequences Red could have taken to win in the same number of moves. If your solution happens to compute a different sequence of the same (minimal) cost, that's perfectly fine.

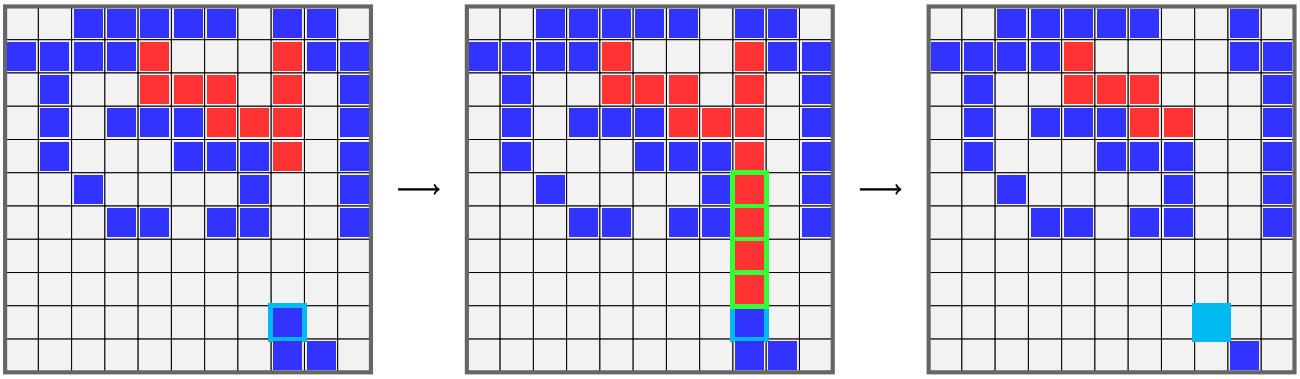
Take a look at the given template code and you'll see this same solution currently "hardcoded" into the `search()` function. Obviously you should write code to solve the *actual* problem yourself but the given example should provide clarity around the structure of the output.



(a) PLACE[(2, 5), (2, 6), (3, 6), (3, 7)]



(b) PLACE[(1, 8), (2, 8), (3, 8), (4, 8)]



(c) PLACE[(5, 8), (6, 8), (7, 8), (8, 8)]

Figure 1: Example solution visualisation for `test-vis1.csv`. After the third PLACE action, a column of tokens is completed, which would remove the highlighted Blue token at $(9, 8)$ and thus solve the problem.

2.5 Testing

Thorough testing of your work is critical in order to ensure that your program functions as expected across a wide array of possible inputs. Testing also helps identify bugs early on during development.

To help get you started, we have provided two sample test cases in the template, which are also live on Gradescope:

- `test-vis1.csv` – a test case with an optimal solution comprising three actions. A solution to this test has already been illustrated in Figure 1.
- `test-vis2.csv` – a test case where there is no possible solution.

If you open these files, you'll see that the initial 11×11 board state is represented as 11 lines of 11 comma-separated characters. Each character is either a "blank" (an empty cell), an `r` (Red token), a `b` (Blue token) or a `B` (the target Blue token). This file format is already parsed for you in the `__main__.py` file that comes in the template. You may wish to read this file to see how this works, but it should *not* be modified.

You can and should write your own tests. The given two cases cover distinct input scenarios but are not "exhaustive" in their own right. In writing your own tests, you might find that some inputs take a very long time to solve. While you are expected to optimise your solution, rest assured we will not assess your work with hidden tests that our sample solution cannot solve quickly (i.e., within a few seconds on Gradescope).

2.6 Gradescope

To help you check that your work is compatible with our marking environment, both of the supplied tests are also live on Gradescope (where you'll submit your project) and can be run at any point before the deadline. Whenever you make a submission, both tests are automatically run on the spot, and the resulting outputs are compared against our sample solution outputs.

You should periodically submit to Gradescope to confirm that your work is compatible with our marking environment. There is no penalty for multiple submissions prior to the deadline, nor is there any hard limit to how many you can make. By default, only the most recent submission will be marked.

When we mark your work, a number of "hidden" tests of varying difficulty will be run in the exact same environment on Gradescope (in addition to the two visible cases). Each test case will be given a maximum execution time of **30 seconds** before being forcibly terminated.

3 The report

You must also briefly discuss your approach to solving this problem in a separate file called `report.pdf`, submitted alongside your program. Your discussion should address the following:

1. With reference to the lectures, which search strategy did you use? Discuss implementation details, including choice of relevant data structures, and analyse the time/space complexity of your solution.
2. If you used a heuristic as part of your approach, clearly state how it is computed and show that it speeds up the search (in general) without compromising optimality. If you did not use a heuristic based approach, justify this choice.
3. Imagine that *all* Blue tokens had to be removed from the board to secure a win (not just one target coordinate). Discuss how this impacts the complexity of the search problem in general, and how your team's solution would need to be modified in order to accommodate it.

Your report can be written using any means but must be submitted as **a PDF document**. Your report should be between 0.5 and 2 pages in length, and must not be longer than 2 pages (excluding references, if any). The quality and readability of your report matters, and marks won't be given where discussion is vague or irrelevant to topics discussed in the subject.

4 Assessment

Your team's Project Part A submission will be assessed out of 8 marks, and contribute 8% to your final score for the subject. Of these 8 marks:

- **5 marks** will be for the correctness of your program, based on running your program through a collection of automated test cases. The tests will run with **Python 3.12** on **Gradescope**. Programs that do not run in this environment will be considered incorrect. There will be a **30 second time limit** per test case, and credit will not be awarded for tests where a timeout occurs. All test cases will be solvable by our sample solution well within this time limit.

  You can minimise the risk of incompatibilities by submitting to Gradescope early and often. You may re-submit as many times as you like, so make sure you take advantage of this. As discussed previously, you should write your own tests in addition to the two given tests, as they don't cover all input scenarios.
- **3 marks** will be for the clarity and accuracy of the discussion in your `report.pdf` file, with 1 mark allocated to each of the three points listed above. A mark will be deducted if the report is longer than 2 pages or not a PDF document.

Your program should use **only standard Python libraries**, plus the optional third-party library **NumPy**[1] (this is just for extra flexibility – use of NumPy is not *required*). 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.

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**. Report 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

This project is to be completed in teams of two. 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).

[1] Currently the latest version, NumPy v1.26, is available in the Gradescope environment

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**.