The University of Melbourne
School of Computing and Information Systems
COMP30024 Artificial Intelligence

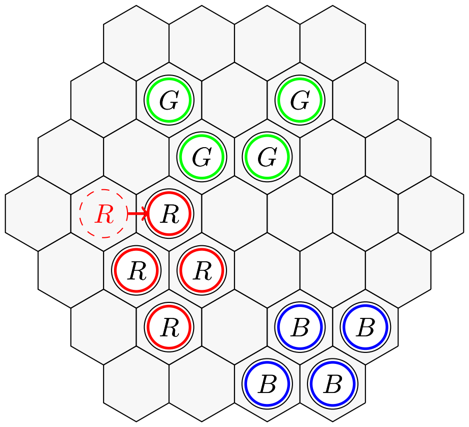
Project Part B: **Playing the Game**

Last updated April 12, 2019

Overview

In this second part of the project we will play the original, three-player version of *Chexers* (as introduced in week 1). Before you read this specification you may wish to re-read the "Rules for the Game of *Chexers*" document.

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



The task

Your task is twofold. Firstly, your team will design and implement a program that 'plays' a game of *Chexers*—given information about the evolving state of a game, your program will decide on an action to take on each of its turns. We provide a driver program that coordinates a game of *Chexers* between three such programs (the 'referee', described in the 'Running your program' section).

Secondly, your team will write a report discussing the strategies your program uses to play the game, the algorithmic techniques you have implemented, and any other creative aspects of your work.

The program

You must create a program to play the game in the form of a Python 3.6 package. The name of this package must be your team name. When imported, your package must define a class named `Player` conforming to the interface described below. Furthermore, to avoid naming conflicts with modules from other teams' packages, all Python modules within your package must use *absolute imports* when importing from other modules within your package.

Your package will therefore be a directory named with your team name. This directory will contain one or more Python modules (`.py` files) and subpackages (subdirectories). One file called `__init__.py` will expose a class called `Player` (though this class may be imported from another module where it is defined). See the provided skeleton code for a starting point, and make sure to seek clarification if you are unsure about how to structure your program.

Representing actions

Our programs will need a consistent way to represent actions. In this part of the project, all actions will be represented as *tuples with two components*; a string representing the **action type**, and a value representing the **action argument** (which may itself be a tuple). We represent the different types of actions as follows:

- To represent a move action, use the action type `"MOVE"` and a nested tuple of coordinates `(`$(q_a, r_a), (q_b, r_b)$`)` as the action argument, where (q_a, r_a) are the coordinates of the moving piece before the move and (q_b, r_b) are the coordinates after the move. For example, the tuple `("MOVE", ((0, 0), (0, 1)))` represents a move action from the hex indexed $(0, 0)$ to the hex indexed $(0, 1)$ in Figure 1.
- To represent a jump action, use the action type `"JUMP"` and a nested tuple of coordinates `(`$(q_a, r_a), (q_b, r_b)$`)` as the action argument, where (q_a, r_a) are the coordinates of the jumping piece before the jump and (q_b, r_b) are the coordinates after the jump. For example, the tuple `("JUMP", ((0, 1), (-2, 3)))` represents a jump action from the hex indexed $(0, 1)$ to the hex indexed $(-2, 3)$ in Figure 1.
- To represent an exit action, use the action type `"EXIT"` and a single tuple of coordinates `(`q`, `r`)`, where (q, r) are the coordinates of the exiting piece before the exit. For example, the tuple `("EXIT", (-2, 3))` represents an exit action from the hex indexed $(-2, 3)$ in Figure 1.
- Sometimes it will be necessary to represent a *pass* rather than an action. In this case, use the action type `"PASS"` and use `None` as the action argument. That is, the tuple `("PASS", None)` represents no action.

While representing actions, we'll index the board's hexes with the *same axial coordinate system*[1] we used in Project Part A. In this system, each hex is addressed by a **column** (q) and **row** (r) pair, as shown in Figure 1.

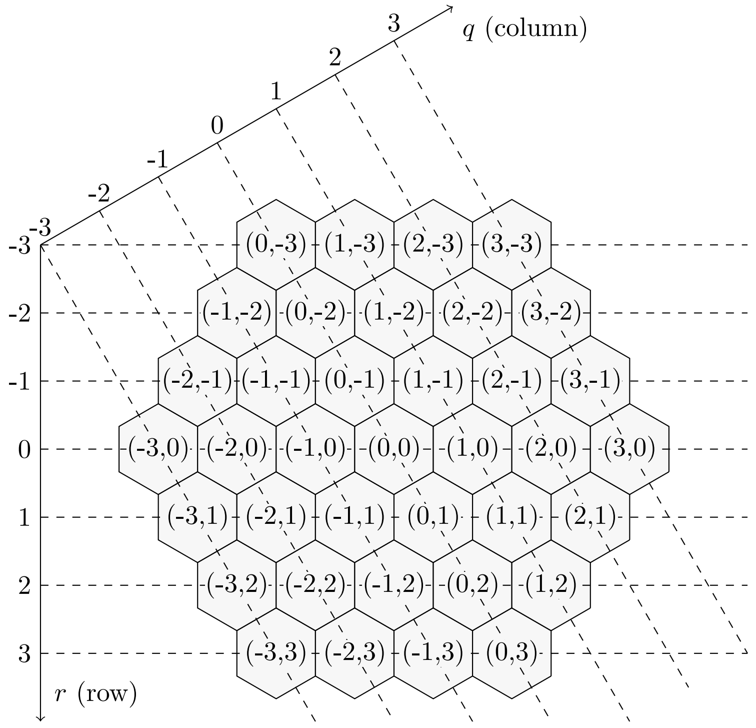


Figure 1: The q and r axes, with (q, r) indices for all hexes.

[1]The following guide to hexagonal grids may prove useful: `redblobgames.com/grids/hexagons/`. In particular, see the 'coordinates' section for tips on using our axial coordinates system. **Don't forget to acknowledge** any algorithms or code you use in your program.

The `Player` class

Your player class, named `Player`, must define at least the following three methods:

1. `def __init__(self, colour):` This method is called once at the beginning of the game to initialise your player. You should use this opportunity to set up your own internal representation of the game state, and any other information about the game state you would like to maintain for the duration of the game.

   The parameter `colour` will be a string representing the player your program will play as (Red, Green or Blue). The value will be one of the strings `"red"`, `"green"`, or `"blue"` correspondingly.

2. `def action(self):` This method is called at the beginning of each of your turns to request a choice of action from your program.

   Based on the current state of the game, your player should select and return an allowed action to play on this turn. If there are no allowed actions, your player must return a pass instead. The action (or pass) must be represented based on the above instructions for representing actions.

3. `def update(self, colour, action):` This method is called at the end of every turn (including your player's turns) to inform your player about the most recent action. You should use this opportunity to maintain your internal representation of the game state and any other information about the game you are storing.

   The parameter `colour` will be a string representing the player whose turn it is (Red, Green or Blue). The value will be one of the strings `"red"`, `"green"`, or `"blue"` correspondingly.

   The parameter `action` is a representation of the most recent action (or pass) conforming to the above instructions for representing actions.

   You may assume that `action` will always correspond to an allowed action (or pass) for the player `colour` (your method *does not* need to validate the action/pass against the game rules).

Program constraints

Resource limits: The following constraints will be strictly enforced on your program during testing. This is to prevent your programs from gaining an unfair advantage by using a large amount of memory and/or computation time. Please note that these limits apply to each player for an entire game. In particular, they *do not* apply to each turn.

- A maximum computation time limit of **60 seconds per player, per game**.
- A maximum memory usage of **100MB per player, per game** (not including space for imported libraries).

Furthermore, any attempt to circumvent these constraints *will not* be allowed. For example, your program must not use multiple threads and must not attempt to communicate with other programs to access additional resources. If you are not sure as to whether some other technique will be allowed, please seek clarification early.

Allowed tools: Your program will be run with **Python 3.6** on the **Melbourne School of Engineering's student unix machines** (for example, `dimefox`[2]). There, the following Python libraries will be available when we test your program. Beyond these, your program should not require any additional tools or libraries in order to play a game:

- The Python Standard Library.
- The third-party Python libraries NumPy and SciPy.
- The library of algorithms provided by the AIMA textbook website.

While you are developing your player program, however, you may use any tools or third-party Python libraries to help you do so. For example, you may like to use third-party Python libraries such as scikit-learn or TensorFlow to help you conduct machine learning. You may even like to use tools based on other programming languages. This is all allowed *as long as your Player class does not require these tools to be available when it plays a game.* If you use any such external tools to help develop your program then these should be acknowledged in your report.

[2]Note that Python 3.6 is not available on `dimefox` by default. However, it can be used after running the command '`enable-python3`' (once per login).

The report

Finally, you must discuss the strategic and algorithmic aspects of your game-playing program in a separate file called `report.pdf` (to be submitted alongside your program). Your discussion may follow the suggested structure described below and include some or all of the following points, or may contain anything else you consider relevant:

- Briefly describe the structure of your game-playing program. Give an overview of the major modules and/or classes you have created and used.
- Describe the approach your game-playing program uses for deciding on which actions to take. Comment on your search strategy, including on how you have handled the 3-player nature of the game. Explain your evaluation function and its features, including their strategic motivations.
- ~~If you have applied machine learning, discuss the learning methodology you followed for training and the intuition behind using that specific technique.~~
- Comment on the overall effectiveness of your game-playing program. If you have created multiple game-playing programs using different techniques, compare their relative effectiveness.
- Include a discussion of any particularly creative techniques you have applied (such as evaluation function design, search strategy optimisations, specialised data structures, other optimisations, or any search algorithms not discussed in lectures) or any other creative aspects of your solution, or any additional comments you wish to be considered by the markers.

Your report can be written using any means but must be submitted as a PDF document. There is no maximum page limit or word limit but you should keep your writing succinct (as a rough guide: a report within a single page may not include enough detail, and a 10-page report likely includes *far too much* detail).

Running your program

To play a game of *Chexers* using your program, we provide a driver program (a 'referee') in a Python package called `referee`, available on the LMS. The referee's main program essentially has the following structure:

1. Set up a new *Chexers* game and initialise a Red, Green and Blue player (constructing three `Player` classes including running their `.__init__()` methods).
2. Repeat the following until the game ends (starting with Red as the current player, then alternating):
   (a) Ask the current player for their next action (calling their `.action()` method).
   (b) Validate this action (or pass) and apply it to the game if it is allowed (otherwise, end the game with an error message). Display the resulting game state to the user.
   (c) Notify all three players (including the current player) of the action (or pass) (using their `.update()` methods).
3. Display the final result of the game to the user.

To play a game using the referee, invoke it as follows, ensuring that the referee package (the directory `referee`) and your player package (the directory named with your team name) are both within your current directory:

```
python -m referee <red_package> <green_package> <blue_package>
```

where `python` is the name of a Python 3.6 interpreter[3] and `<red_package>`, `<green_package>`, and `<blue_package>` are the names of packages containing the class `Player` to be used for Red, Green, and Blue, respectively.

The referee offers many additional command-line options, including for inserting a delay between turns (to make games easier to watch); controlling output verbosity; creating an action log; enforcing time constraints and (on linux) space constraints; and using other player classes (not named `Player`) from a package (useful for testing your main player against other player classes). You can find information about these additional options by running:

```
python -m referee --help
```

[3]Note that Python 3.6 is not available on `dimefox` by default. However, it can be used after running the command '`enable-python3`' (once per login).

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:

- **4 marks** will be for the ability of your program to play a legal game of *Chexers*. Partial marks will be deducted for programs that crash with syntax or runtime errors (including from failing to use absolute imports), return invalid actions (or passes), or consistently run out of time or space.
- **4 marks** will be for the quality of your program's code: its structure (including good use of modules, classes, and functions), readability (including good use of code comments and meaningful variable names), and documentation (including use of docstrings for modules, classes, and functions).
- **7 marks** will be for the performance of your program, based on playing your program against a suite of hidden 'benchmark' opponents of increasing difficulty. The opponents will range in difficulty as described below, with the maximum marks available for winning games against each type of player shown in parentheses:
  - Players who choose uniformly at random from the set of allowed actions each turn (**3 marks available**).
  - 'Greedy' players who select the most immediately promising action available without considering opponent responses (**2 marks available**).
  - Players using basic adversarial search techniques discussed in lectures and basic evaluation functions to look a small number of turns ahead when selecting actions (**1 mark available**).
  - Players using enhanced adversarial search techniques discussed in lectures and improved evaluation functions to look further ahead when selecting actions (**1 mark available**).

  We will test your player as Red, Green, and Blue against different combinations of the above opponents and derive your mark out of 7 based on the proportion of games won against each type of opponent.
- **7 marks** will be for the overall 'creativity' demonstrated by your work. As discussed in lectures, this component of your mark is a measurement of how far beyond the basic techniques discussed in lectures you have taken your implementation. Possible examples of 'creativity' include (but are not limited to) improving the choice of search algorithm, designing a clever and effective evaluation function, or optimising representation and implementation efficiency. The following marking criteria will be applied:
  - **0–1 marks:** Work that does not demonstrate a successful application of the important techniques discussed in lectures for playing adversarial games.
  - **1–3 marks:** Work that demonstrates a successful application of some or many of the techniques discussed in lectures for playing adversarial games, and possibly some strategic or algorithmic enhancements to these techniques.
  - **3–5 marks:** Work that demonstrates a successful application of many of the techniques discussed in lectures for playing adversarial games (or additional/alternative techniques beyond those discussed) and may include some significant theoretical or algorithmic enhancements to these approaches, original strategic insights specific to the game being played, or improvements to the process of designing a game-playing program.
  - **5–7 marks:** Work that demonstrates a highly successful application of many of the techniques discussed in lectures for playing adversarial games (or additional/alternative techniques beyond those discussed) and includes many significant theoretical or algorithmic enhancements to these approaches, original strategic insights specific to the game being played, and improvements to the process of designing a game-playing program, leading to a program with excellent performance.

  Note that your report will be an important part of how we assess this component of the project, so please use it as an opportunity to highlight any creative aspects of your work. Furthermore, note that other creative aspects of your work not matching those outlined in this criteria may also be taken into account when deciding your mark for this component.

Additional notes: Once again, all testing games will be run with **Python 3.6** on the **Melbourne School of Engineering's student unix machines**. Therefore, we strongly recommended that you test your program in this environment before submission. If your team still has trouble accessing the student unix machines, please seek help *well before* submission.

Academic integrity

Your submission should be entirely the work of your team. You are encouraged to discuss ideas with your fellow students, but it is not acceptable to share code between teams, nor to use the code of someone else. You should not show your code to another team, nor ask another team to look at their code. **This includes sharing player programs for the purposes of playing *Chexers* games**, which is *not* allowed.

You are encouraged to use code-sharing and collaboration services (e.g. GitHub) **within** your team. However, you should take care to ensure that your code is **never visible to students outside your team** (for example by setting your online repository to 'private' mode, so that only your team members can access it.).

If your program is found to be suspiciously similar to someone else's or a third party's software, or if your submission is found to be the work of a third party, you may be subject to investigation and, if necessary, formal disciplinary action.

Please refer to the 'Academic Integrity' section of the LMS and to the university's academic integrity website (`academicintegrity.unimelb.edu.au`) if you need any further clarification on these points.

Submission

One submission is required from each team. That is, one team member is responsible for submitting all of the necessary files that make up your team's solution.

You must submit a single compressed archive file (e.g. a `.zip` or `.tar.gz` file) containing all files making up your solution via the 'Project Part B Submission' item in the 'Assessments' section of the LMS. This compressed file should contain all Python files required to run your program (with the correct directory structure), along with your report. In addition, if you have created any extra files to assist you while working on this project[4]then all of these files are worth including when you submit your solution.

The submission deadline for Project Part B is 11:00PM on Tuesday the 21st of May.

Late submissions are allowed. A late submission will incur a penalty of two marks per working day (or part thereof) late. You may submit multiple times. We will mark the latest submission made by a member of your team, unless we are advised otherwise.

Extensions

If you require an extension, please email Matt (`matt.farrugia@unimelb.edu.au`) at the earliest possible opportunity. We will then assess whether an extension is appropriate. If you have a medical reason for your request, you will be asked to provide a medical certificate. Requests for extensions on medical grounds received after the deadline may be declined.

Note that computer systems are often heavily loaded near project deadlines, and unexpected network or system downtime can occur. Generally, system downtime or failure will *not* be considered as grounds for an extension. You should plan ahead to avoid leaving things to the last minute, when unexpected problems may occur.

Teamwork issues

In the unfortunate event that an issue arises between you and your partner (such as a breakdown in communication, a dispute about responsibilities or individual contributions to the project, or any other such issue), and you are unable to resolve this issue within your team, then please email Matt (`matt.farrugia@unimelb.edu.au`). Note that it is in your best interest that any teamwork issues are identified as early as possible so that we have a chance to mitigate their effect on your completion of the project.

In addition, while completing this project, it is a good idea to keep brief notes, minutes, or chat logs recording topics discussed at each meeting, and other such documents related to your collaboration. In the event of a teamwork-related issue these documents (along with your project plan) will help us to reach a fair resolution.

[4]For example you may have created alternative player classes, a modified referee, additional programs to test your player or its strategy, programs to create training data for machine learning, or programs for any other purpose not directly related to implementing your player class. As long as these files are not too large, you are encouraged to include them with your submission.