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

Project Part A:
Searching

Last updated March 17, 2019

Overview

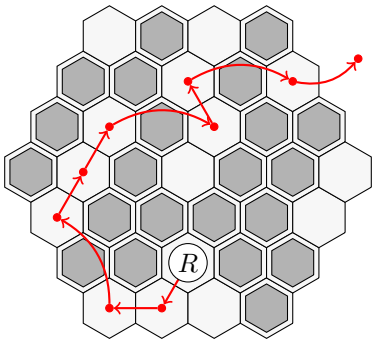
In this first part of the project, we will play a single-player variant of the game of *Chexers*. Before you read this specification, please make sure you have carefully read the entire 'Rules for the Game of *Chexers*' document.

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 new algorithms we have met in lectures so far, and (3) become more familiar with some core mechanics of *Chexers*. This is also a chance for you to invest some time developing 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 game-playing program for Project Part B).

Single-player *Chexers*

The single-player variant of *Chexers* we will analyse has the following differences from regular *Chexers*:

1. There is only one player (it may be Red, Green, or Blue). This player starts the game with at most 4 pieces in some configuration. Then, they repeatedly take turns until they have taken an exit action with all of their pieces (according to the usual rules for taking exit actions). At this point, the player wins the game.
2. There are no pieces of other colours, however some hexes may be occupied by colourless **blocks**. These blocks never move. They can be jumped-over like regular pieces, but they cannot be converted (unlike regular pieces).



The tasks

Firstly, your team will design and implement a program that 'plays' a game of single-player *Chexers*—given a board configuration, your program will find a sequence of actions for the player to take to win (to exit with all of their pieces). Ideally (for full marks) this will be a *shortest possible* sequence (as measured by the number of actions), but most of the marks for this task are available for programs that find longer sequences too. (Complete information on the project marking scheme will be given below, in the Assessment section.)

Secondly, your team will 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.

The program

You must create a program to play this game in the form of a Python 3.6 module called `search` (for example, a Python file called `search.py` would be sufficient—see the provided skeleton code for a starting point).

When executed, your program must read a JSON-formatted board configuration from a file, calculate a winning sequence of actions, and print out this sequence of actions. The input and output format are specified below, along with the coordinate system we will use for both input and output.

Coordinate system

For input and output, we'll index the board's hexes with an *axial coordinate system*[1]. In this system, each hex is addressed by a **column** (q) and **row** (r) pair, as shown in Figure 1.

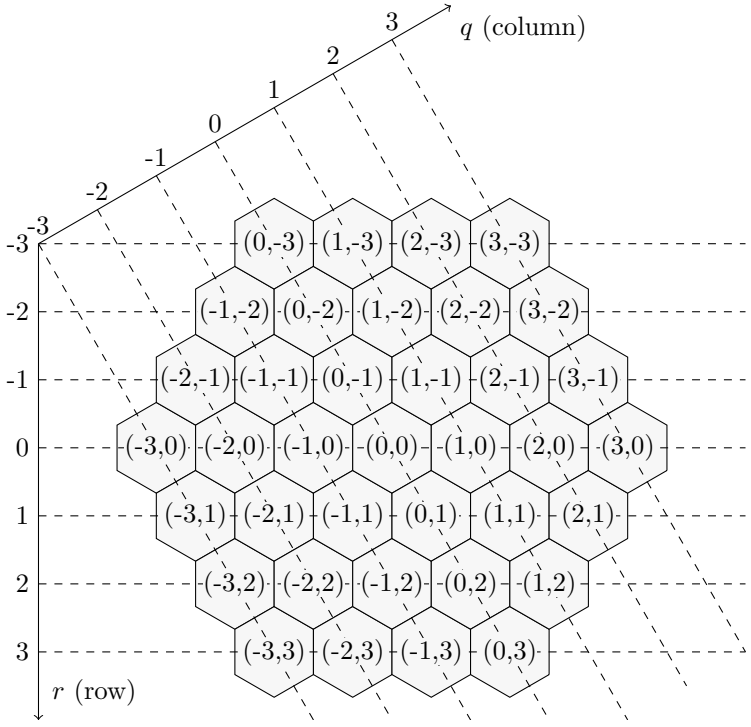


Figure 1: The q and r axes, with (q, r) indices for all hexes.

Input format

Your program must read a board configuration from a JSON-formatted file. The name of the file will be given to your program as its first (and only) command-line argument. The JSON file will contain a single dictionary (JSON object) with the following 3 entries (name/value pairs):[2]

- An entry with key `"colour"` declaring which player is playing the game (Red, Green, or Blue). The value will be `"red"`, `"green"`, or `"blue"` correspondingly. Note that the player's colour will determine from which hexes exiting actions are available.
- An entry with key `"pieces"` specifying the starting position of the player's piece(s). The value will be a list (JSON array) of 1-4 coordinate pairs, with each pair represented by a two-element list of integers (JSON numbers) in the format `[q, r]` (q is the column and r is the row of the hex the piece is occupying—see Figure 1). For example, the entry `"pieces": [[-1,2]]` indicates a single piece on the hex indexed $(-1, 2)$.

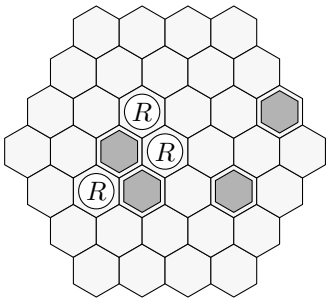
[1] The following guide to hexagonal grids may prove useful: redblobgames.com/grids/hexagons/. In particular, see the 'coordinates' section for tips on using our axial coordinates system. **Don't forget to acknowledge** any algorithms or code you use in your program.

[2] We recommended using Python's Standard Library module '`json`' for loading JSON-formatted input data. In this setting, an appropriate call would be '`with open(sys.argv[1]) as file: data = json.load(file)`' (creating `data`, a dictionary with the input).

- An entry with key `"blocks"` specifying the position of any blocks on the board. The value will be a list of 0-36 coordinate pairs, with each pair represented by a two-element list of integers in the format `[q, r]` (q is the column and r is the row of the hex the block is occupying—see Figure 1). For example, the entry `"blocks": [[0,1], [2,-1]]` indicates blocks on the hexes indexed $(0, 1)$ and $(2, -1)$.

```
{
    "colour": "red",
    "pieces": [[0,0],[0,-1],[-2,1]],
    "blocks": [[-1,0],[-1,1],[1,1],[3,-1]]
}
```

(a) A complete example input file.



(b) The corresponding board configuration.

Your program may assume that its input will match this format specification exactly, and that all column and row values represent real and otherwise unoccupied hexes on the board (that is, there will be no invalid or duplicated coordinates and no improperly formatted JSON in the input files we provide to your program).

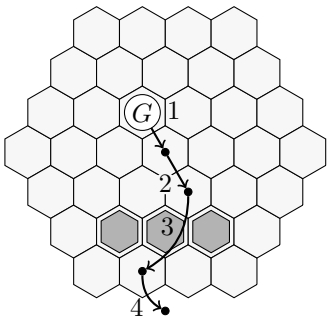
Furthermore, your program may assume that at least one winning sequence of actions exists for the board configuration described in its input (that is, your program will not be tested on configurations with no solution).

Output format

Your program must print out a winning sequence of actions. Each action must be allowed according to the rules of the game when played in the order of output. The actions must be printed to *standard output*, one per line, in the following format:

- To output a move action, print a line in the format '`MOVE from (`q_a`, `r_a`) to (`q_b`, `r_b`).`' 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.
- To output a jump action, print a line in the format '`JUMP from (`q_a`, `r_a`) to (`q_b`, `r_b`).`' 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.
- To output an exit action, print a line in the format '`EXIT from (`q`, `r`).`' where (q, r) are the coordinates of the exiting piece before the exit.

Your program *should not print any other lines to standard output.* If you must print additional output, consider printing to *standard error* instead. Alternatively, we will ignore lines in standard output beginning with the character '`#`', so you may use this character to begin any lines of output that are not part of your action sequence.



(a) An example board configuration and action sequence.

action no.	standard output
1	`MOVE from (0, -1) to (0, 0).`
2	`MOVE from (0, 0) to (0, 1).`
-	`# lines beginning with # are ignored`
3	`JUMP from (0, 1) to (-2, 3).`
4	`EXIT from (-2, 3).`
-	`# dont forget the full stops!`

(b) The corresponding output.

The report

Finally, you must briefly discuss your approach to solving this problem in a separate file called `report.pdf` (to be submitted alongside your program). Your discussion should be structured around the following 3 questions:

- How have you formulated the game as a search problem? (You could discuss how you view the problem in terms of states, actions, goal tests, and path costs, for example.)
- What search algorithm does your program use to solve this problem, and why did you choose this algorithm? (You could comment on the algorithm's efficiency, completeness, and optimality. You could explain any heuristics you may have developed to inform your search, including commenting on their admissibility.)
- What features of the problem and your program's input impact your program's time and space requirements? (You might discuss the branching factor and depth of your search tree, and explain any other features of the input which affect the time and space complexity of your algorithm.)

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.

Assessment

Project Part A will be marked out of 8 marks, and contribute 8% to your final score for the subject. Of these 8 marks:

- 2 marks will be for the quality of your program's code (its structure and readability, including use of code comments) and its adherence to input and output format requirements.
- 4 marks will be for the correctness of your program, based on running your program through a collection of automated test cases of increasing complexity. The test cases will range in difficulty as described below, with 1 mark allocated to passing the test cases at each 'level' of difficulty.[3]

  1. The player is always Red, and the game always starts with single piece.
  2. The player is Green or Blue, and the game always starts with a single piece.
  3. The player may be any colour, and the game starts with 2-4 pieces.

  The 4^{th} mark is allocated for programs which pass all 3 of these levels and, in addition, always find a *shortest possible* solution, as measured by the number of actions in the output sequence.
- 2 marks will be for the clarity and accuracy of the discussion in your `report.pdf` file. A mark will be deducted if the report is longer than 2 pages or not a PDF document.

Additional notes

- All tests will be conducted on the **Melbourne School of Engineering's student unix machines** (for example, `dimefox`). There, your program will be given **up to 30 seconds** of computation time per test case. Programs that do not run in this environment or compute a solution within this time limit will be considered incorrect. Therefore, we strongly recommended that you test your program in this environment[4]before submission. If your team has trouble accessing the student unix machines, please seek help *well before* submission.
- Your program will be run with **Python 3.6**. For this part of the project, your program should **not require any tools from outside the Python Standard Library**.

  As one exception to the above, you may optionally make use of the library of algorithms provided by the AIMA textbook website (provided you make appropriate acknowledgements that you have used this library). This library will be made available to your program in the testing environment if required.

[3]It may help to break this task down into stages. For example, begin by implementing a program that can pass tests of 'difficulty level 1' described above, then move on to the higher levels. Furthermore, even if you don't have a program that works correctly by the time of the deadline, you should submit anyway. You may be awarded some marks for making a reasonable attempt at the project.

[4]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).

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.

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 A Submission' item in the 'Assessments' section of the LMS. This compressed file should contain all Python files required to run your program, along with your report.

The submission deadline for Project Part A is 11:00PM on Friday the 12th of April.

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`) using the subject 'COMP30024 Extension Request' 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.