Project Part A

Searching

COMP30024 Artificial Intelligence

2021

Overview

In this first part of the project, we will play a simplified, single-player (non-adversarial), search-based variant of the game of *RoPaSci 360*. Before you read this specification, please make sure you have carefully read the entire 'Rules for the Game of *RoPaSci 360*' 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 algorithms you have met in lectures, and (3) become more familiar with some core mechanics of *RoPaSci 360*. 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).

Single-player *RoPaSci 360*

The single-player game we will analyse is based on *RoPaSci 360*, with the following modifications:

1. There is *no throw action*. Instead, the game begins with tokens already on the board.
2. Lower's tokens *never move.* On each turn, *all* of Upper's tokens move (each using a *slide* or *swing* action), all at the same time. If Upper has more than one token, they all move at the same time.
3. The player must repeatedly move the Upper tokens to *defeat all of Lower's tokens* to win. If the last Upper token is defeated in the same turn as the last Lower token, the player still wins.
4. There is a new type of token: **Block tokens** prevent other tokens from occupying their hexes. Block tokens never move. No token can slide or swing onto or over a Block token.

Most of the other rules (for example, the rules for slide and swing actions, and the rules for token battles on shared hexes and defeating tokens) are the same as in the original *RoPaSci 360* game.

The tasks

Firstly, you will **design and implement a program** that 'plays' a game of single-player *RoPaSci 360*. That is, given a starting board configuration, the program must find a sequence of actions to win the game (to defeat all Lower tokens). Secondly, you 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**[1] called `search`.

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*[2]. In this system each hex is addressed by a **row** (r) and **column** (q) pair, as shown in Figure 1.

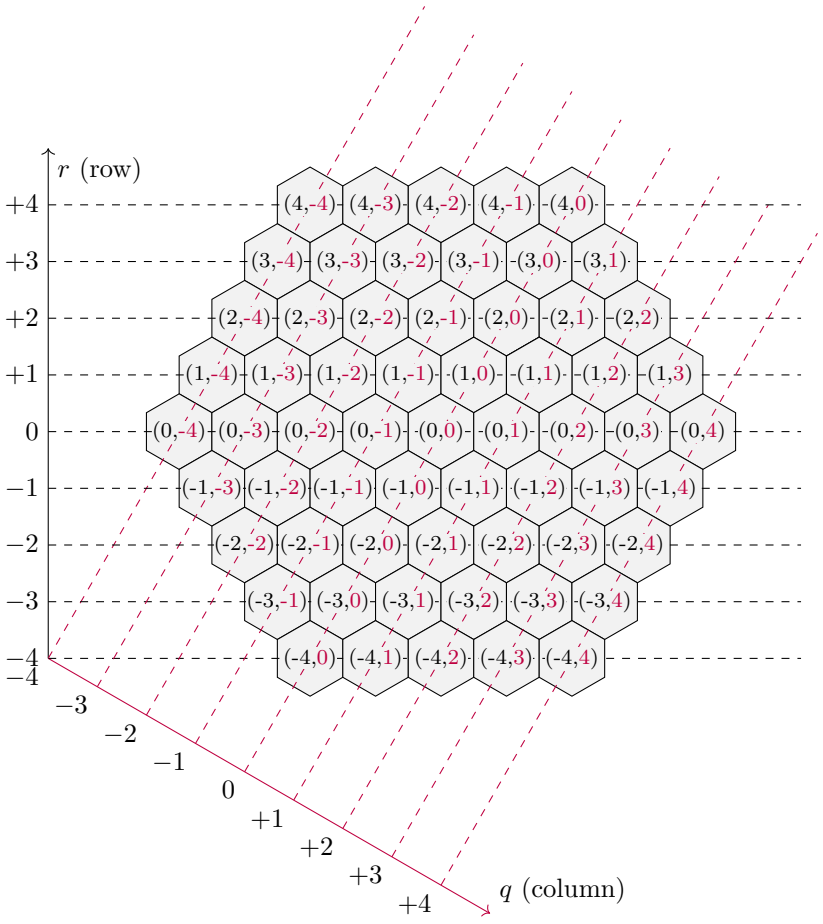


Figure 1: The r and q axes, with (r, q) indices for all hexes.

[1] This module might be a *single file* called `search.py`, or it might include multiple files in a directory called `search/`, with an entry-point in `search/__main__.py`. Either way, you should be able to run the program using the command `python -m search` (followed by command-line arguments). See the skeleton code for an example of the correct structure.

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

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.[3] The JSON file will contain a single dictionary (JSON object) with the following three entries:

- An entry with key `"upper"` specifying the starting position of Upper's tokens as a list of token descriptions (as described below).
- An entry with key `"lower"` specifying the position of Lower's tokens as a list of token descriptions.
- An entry with key `"block"` specifying the position of any blocking tokens as a list of token descriptions.

Each token description is a list of the form `[`s`,`r`,`q`]` where s is a string representing the token's symbol (`"r"` for Rock, `"p"` for Paper, `"s"` for Scissors; blocking tokens have no symbol and use the string `""`), and (r, q) are the axial coordinates of the token's hex (see Figure 1).

You may assume that (1) the input matches this format description exactly (we will not give you invalid JSON), (2) all token descriptions contain valid coordinates (we will not give you coordinates outside of the board), (3) there are no overlapping tokens (we will not give you a configuration with two tokens initially on the same hex), and (4) there is at least one winning sequence of actions (we will not give you configurations from which it is impossible to win).

Output format

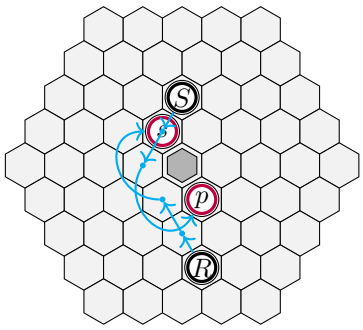
Your program must print out the actions Upper should take on each turn to win the game. The actions must be printed to *standard output*[4], with one line of output per action, in the following format:

- To output a slide action on turn t for a token on hex (r_a, q_a) sliding to hex (r_b, q_b), print the line: '`Turn` t`: SLIDE from (`r_a`,`q_a`) to (`r_b`,`q_b`)`'.
- To output a swing action on turn t for a token on hex (r_a, q_a) swinging to hex (r_b, q_b), print the line: '`Turn` t`: SWING from (`r_a`,`q_a`) to (`r_b`,`q_b`)`'.

The first turn is numbered 1, and note that in this single-player variant of the game, Upper will take *one action with each token on each turn*, and these actions happen at the same time. For example, if Upper controls three tokens, the first three lines of output should all begin with '`Turn 1:`'.

```
{
  "upper": [["s", 2,-1],
            ["r",-3, 2]],
  "lower": [["s", 1,-1],
            ["p",-1, 1]],
  "block": [["",  0, 0]]
}
```

(a) An example input file.



(b) Starting board and a solution.

```
# Lines like this are ignored
Turn 1: SLIDE from (2,-1) to (1,-1)
Turn 1: SLIDE from (-3,2) to (-2,1)
# Both Scissors now share a hex
Turn 2: SLIDE from (1,-1) to (0,-1)
Turn 2: SLIDE from (-2,1) to (-1,0)
# Note, turn happens all at once:
Turn 3: SWING from (0,-1) to (-1,1)
Turn 3: SWING from (-1,0) to (1,-1)
```

(c) Standard output for this solution.

[3] We recommended using Python's Standard Library module '`json`' for loading this input. An appropriate call would be '`with open(sys.argv[1]) as file: data = json.load(file)`' (creating `data`, a dictionary version of the input).

[4] Your program *should not print any other lines to standard output.* However, we will ignore lines beginning with the character '`#`', so you may use this character to begin any lines of output that are not part of your action sequence.

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 answer these three questions:

1. How did you formulate this game as a search problem? Explain your view of the problem in terms of states, actions, goal tests, and path costs, as relevant.
2. What search algorithm does your program use to solve this problem, and why did you choose this algorithm? Comment on the algorithm's efficiency, completeness, and optimality. If you have developed heuristics to inform your search, explain them and comment on their admissibility.
3. How do the features of the starting configuration (the position and number of tokens) impact your program's time and space requirements? For example, discuss their connection with the branching factor and search tree depth and how these 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, and must not be longer than 2 pages (excluding references, if any).

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 of increasing complexity, as described below. The first mark is earned by correctly following the input and output format requirements. The remaining four marks are available for passing the tests at each of the following four difficulty 'levels':[5]

  1. Upper and Lower each begin with a single token. There are no blocking tokens.
  2. Upper begins with 1 token and Lower begins with 2–9 tokens. There may be blocking tokens.
  3. Upper begins with 2 tokens and Lower begins with 1–9 tokens. There may be blocking tokens.
  4. Upper begins with 3 tokens and Lower begins with 1–9 tokens. There may be blocking tokens.

  The tests will run with **Python 3.6** on the **student Unix machines** (for example, `dimefox`[6]). There will be a **30 seconds time limit** per test case. Programs that do not run in this environment or do not complete within this time will be considered incorrect.

- **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 questions 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 libraries **NumPy and SciPy** (these are the only libraries installed on `dimefox`). With acknowledgement, you may also include code from the AIMA textbook's Python library, where it is compatible with Python 3.6 and the above limited dependencies.

[5] 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', 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 may be awarded some marks for making a reasonable attempt.

[6] We strongly recommended that you test your program on `dimefox` before submission. Seek our help early if you cannot access `dimefox`. Note that Python 3.6 is not available on `dimefox` by default, but it can be used after running the command '`enable-python3`' (once per login).

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 (`academicintegrity.unimelb.edu.au`) 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.

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 is **11:00PM on Wednesday the 31st March, Melbourne time (AEDT).**

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

Extensions

If you require an extension, please email the lecturers 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 may be declined.