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

Project Part A:
Searching

Last updated March 3, 2020

Overview

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

The single-player variant of *Expendibots* we will analyse works as follows. You will always play as White, and you will be the only player that acts. The Black player is static and does not take any turns. As the White player, you will start with at most 3 tokens in some configuration. You will repeatedly take turns until all of the Black tokens have been eliminated, at which point you win. If on your last turn you eliminate all enemy tokens but lose your last token, you still win.

The tasks

Firstly, your team will design and implement a program that 'plays' a game of single-player *Expendibots* — given a board configuration, your program will find a sequence of actions for the White player to take to win (to eliminate all enemy tokens). Your program's performance will be judged based upon its ability to find winning sequences of actions and handle cases involving multiple tokens (See the Assessment section for details). There is *no* requirement that your sequences be optimal.

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 folder named `search` containing a Python file called `__main__.py` as the program entry-point[1] 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.

[1] To run such a Python 'module' from the command-line, begin outside the `search` folder and use the following command: '`python -m search`' (followed by command-line arguments). If `python` is your Python 3.6 interpreter, it will execute the code inside `search/__main__.py`.

Coordinate system

The coordinate system in use is a two element pair (x, y) representing the x (horizontal) and y (vertical) position on the board. This is shown in Figure 1. Note that the coordinates are zero-indexed.

$(0,7)$	$(1,7)$	$(2,7)$	$(3,7)$	$(4,7)$	$(5,7)$	$(6,7)$	$(7,7)$
$(0,6)$	$(1,6)$	$(2,6)$	$(3,6)$	$(4,6)$	$(5,6)$	$(6,6)$	$(7,6)$
$(0,5)$	$(1,5)$	$(2,5)$	$(3,5)$	$(4,5)$	$(5,5)$	$(6,5)$	$(7,5)$
$(0,4)$	$(1,4)$	$(2,4)$	$(3,4)$	$(4,4)$	$(5,4)$	$(6,4)$	$(7,4)$
$(0,3)$	$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)$	$(5,3)$	$(6,3)$	$(7,3)$
$(0,2)$	$(1,2)$	$(2,2)$	$(3,2)$	$(4,2)$	$(5,2)$	$(6,2)$	$(7,2)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$	$(5,1)$	$(6,1)$	$(7,1)$
$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$	$(4,0)$	$(5,0)$	$(6,0)$	$(7,0)$

Figure 1: Coordinate system for *Expendibots*.

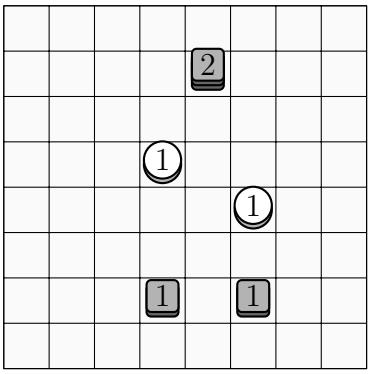
Input format

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 2 entries (name/value pairs):[2]

- An entry with key `"white"` specifying the starting position of your player's token(s). The value will be a list (JSON array) of *stacks*, with each stack represented in the form `[`n`,`x`,`y`]`. This represents a stack of n tokens at the coordinates (x, y) (as described in Figure 1). For example, the entry `"white":[[2,7,0]]` indicates a single stack of two White tokens at $(7, 0)$ (which is the bottom right corner of the board).
- An entry with key `"black"` specifying the position of your opponent's token(s) in the same format.

```
{
    "white": [[1,5,3],[1,3,4]],
    "black": [[2,4,6],[1,3,1],[1,5,1]]
}
```

(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 stack coordinates represent real and otherwise unoccupied squares 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).

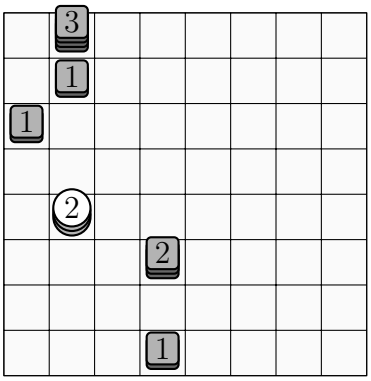
[2]We recommended using Python's Standard Library module '`json`' for loading JSON-formatted input data. In this setting, an appropriate call would be '`with open(sys.argv[1]) as file:  data = json.load(file)`' (creating `data`, a dictionary with the input).

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` n `from (`x_a`,` y_a`) to (`x_b`,` y_b`).`' where n is the number of tokens in the stack to move, (x_a, y_a) are the coordinates of the moving tokens before the move, and (x_b, y_b) are the coordinates after the move. Note: If you want to output the action of moving a whole stack, n would just be equal to the number of tokens in the stack.
- To output a boom action, print a line in the format '`BOOM at (`x`,` y`).`' where (x, y) are the coordinates of the stack initiating the explosion.

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.

```
MOVE 1 from (1, 3) to (1, 1).
MOVE 1 from (1, 1) to (2, 1).
BOOM at (2, 1).
MOVE 1 from (1, 3) to (1, 4).
BOOM at (1, 4).
# lines beginning with # are ignored
# don't forget the full stops!
```

(b) One possible solution 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, and must not be longer than 2 pages.

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. The test cases will range in difficulty as described below, with 1 mark earned by passing the test cases at each 'level' of difficulty.[3]

  1. The player controls 1 White token, and there is 1 Black token.
  2. The player controls 1 White token, and there are 2–12 Black tokens.
  3. The player controls 2 or 3 White tokens (initially at separate squares and with no need to form larger stacks to find a solution), and there are 2–12 Black tokens.
  4. The player controls 2 or 3 White tokens (possibly in stacks of multiple tokens, or with a need to form such stacks), and there are 2–12 Black tokens.

  The 5$^{\text{th}}$ mark is earned by correctly adhering to the input and output format requirements.

- **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.

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.

Academic integrity

Your submission should be **entirely the work of your team.** We automatically check all submissions for originality. **Submitting work that is not entirely your own is against the university's academic integrity policy, and may lead to formal disciplinary action.** For example, please note the following:

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 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.

Please refer to the 'Academic Integrity' section of the LMS and to the university's academic integrity website (`academicintegrity.unimelb.edu.au`), or ask the teaching team, if you need further clarification.

[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.

[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).

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 TODO (`<<TODO>>`) 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.