COMP3702 Artificial Intelligence (Semester 2, 2021)
Assignment 2: DragonGame MDP

Key information:

- **Due: 4pm, Friday 24 September**
- This assignment will assess your skills in developing algorithms for solving MDPs.
- Assignment 2 contributes 15% to your final grade.
- This assignment consists of two parts: (1) programming and (2) a report.
- This is an individual assignment.
- Both code and report are to be submitted via Gradescope (`https://www.gradescope.com/`). You can find instructions on how to register for the COMP3702 Gradescope site on Blackboard.
- Your program (Part 1, 60/100) will be graded using the Gradescope code autograder, using testcases similar to those in the support code provided at `https://gitlab.com/3702-2021/a2-support`.
- Your report (Part 2, 40/100) should fit the template provided, be in .pdf format and named according to the format `a2-COMP3702-[SID].pdf`, where SID is your student ID. Reports will be graded by the teaching team.

The DragonGame AI Environment

"Untitled Dragon Game"[1] or simply DragonGame, is a 2.5D Platformer game in which the player must collect all of the gems in each level and reach the exit portal, making use of a jump-and-glide movement mechanic, and avoiding landing on lava tiles. DragonGame is inspired by the "Spyro the Dragon" game series from the original PlayStation. For this assignment, there are some changes to the game environment indicated in pink font. In Assignment 2, actions may have non-deterministic outcomes!

To optimally solve a level, your AI agent must find a policy (mapping from states to actions) which collects all gems and reaches the exit while incurring the minimum possible expected action cost.

Levels in DragonGame are composed of a 2D grid of tiles, where each tile contains a character representing the tile type. An example game level is shown in Figure 1.

```
XXXXXXXXXXXXXXXXXXXX
XX            XXXXXX
X G             XXXX
XXX=X             XX
X  =        G      X
X  =       XX=     X
X  =  G      =     X
X=XXXXXX     =     X
X=    X      =   G X
X=    X      =  XXXX
X=        =X = XX  X
X=XX G   =XX =    EX
X=PXXX  XXXXXXX   =X
XXXXXX**XXXXXXX**XXX
```

Figure 1: Example game level of DragonGame

[1] The full game title was inspired by Untitled Goose Game, an Indie game developed by some Australians in 2019

Game state representation

Each game state is represented as a character array, representing the tiles and their position on the board. In the visualizer and interactive sessions, the tile descriptions are triples of characters, whereas in the input file these are single characters.

Levels can contain the tile types described in Table 1.

Table 1: Table of tiles in DragonGame, their corresponding symbol and effect

Tile	Symbol in Input File	Symbol in Visualiser	Effect
Solid	`'X'`	`'XXX'`	The player cannot move into a Solid tile. Walk and jump actions are valid when the player is directly above a Solid tile.
Ladder	`'='`	`'==='`	The player can move through Ladder tiles. Walk, jump, glide and drop actions are all valid when the player is directly above a Ladder tile. When performing a walk action on top of a ladder tile, there is a small chance to fall downwards by 2 tiles (when the position 2 tiles below is non-solid).
Air	`' '`	`'   '`	The player can move through Air tiles. Glide and drop actions are all valid when the player is directly above an Air tile.
Lava	`'*'`	`'***'`	Moving into a Lava tile or a tile directly above a Lava tile results in Game Over.
Super Jump	`'J'`	`'[J]'`	When the player performs a '`Jump`' action while on top of a Super Jump tile, the player will move upwards by between 2 and 5 tiles (random outcome - probabilities given in input file). If blocked by a solid tile, probabilities for moving to each of the blocked tiles roll over to the furthest non-blocked tile. For all other actions, Super Jump tiles behave as 'Solid' tiles.
Super Charge	`'C'`	`'[C]'`	When the player performs a '`Walk Left`' or '`Walk Right`' action while on top of a Super Charge tile, the player will move (left or right) until on top of the last adjoining Super Charge tile (see example in Table 2), then move in the same direction by between 2 and 5 tiles (random outcome - probabilities given in input file). If blocked by a solid tile, probabilities for moving to each of the blocked tiles roll over to the furthest non-blocked tile. For all other actions, Super Charge tiles behave as 'Solid' tiles.
Gem	`'G'`	`' G '`	Gems are collected (disappearing from view) when the player moves onto the tile containing the gem. The player must collect all gems in order to complete the level. Gem tiles behave as 'Air' tiles, and become 'Air' tiles after the gem is collected.
Exit	`'E'`	`' E '`	Moving to the Exit tile after collecting all gems completes the level. Exit tiles behave as 'Air' tiles.
Player	`'P'`	`' P '`	The player starts at the position in the input file where this tile occurs. The player always starts on an 'Air' tile. In the visualiser, the type of tile behind the player is shown in place of the spaces.

An example of performing the Walk Right action on a Super Charge tile is shown in Table 2:

Table 2: Example `Walk Right` action on a Super Charge tile

		P	>>>	>>>	>>>	(P)	(P)	(P)	(P)		
XXX	[C]	[C]	[C]	[C]	XXX	XXX		XXX	XXX	XXX	

'P' represents the current player position, '>>>' represents tiles which are skipped over, and each (P) represents a possible new position of the player.

Actions

At each time step, the player is prompted to select an action. Each action has an associated cost, representing the amount of energy used by performing that action. Each action also has requirements which must be satisfied by the current state in order for the action to be valid. The set of available actions, costs and requirements for each action are shown in Table 3.

Table 3: Table of available actions, costs and requirements

Action	**Symbol**	**Cost**	**Description**	**Validity Requirements**
Walk Left	wl	1.0	Move left by 1 position; if above a Ladder tile, random chance to move down by 2; if above a Super Charge tile, move a variable amount (see example in Table 2)	Current player must be above a Solid or Ladder tile, and new player position must not be a Solid tile.
Walk Right	wr	1.0	Move right by 1 position; if above a Ladder tile, random chance to move down by 2; if above a Super Charge tile, move a variable amount (see example in Table 2)	
Jump	j	2.0	Move up by 1 position; if above a Super Jump tile, move up by between 2 and 5 (random outcome)	
Glide Left 1	gl1	0.7	Move left by between 0 and 2 (random outcome) and down by 1.	Current player must be above a Ladder or Air tile, and all tiles in the axis aligned rectangle enclosing both the current position and new position must be non-solid (i.e. Air or Ladder tile). See example below.
Glide Left 2	gl2	1.0	Move left by between 1 and 3 (random outcome) and down by 1	
Glide Left 3	gl3	1.2	Move left by between 2 and 4 (random outcome) and down by 1	
Glide Right 1	gr1	0.7	Move right by between 0 and 2 (random outcome) and down by 1	
Glide Right 2	gr2	1.0	Move right by between 1 and 3 (random outcome) and down by 1	
Glide Right 3	gr3	1.2	Move right by between 2 and 4 (random outcome) and down by 1	
Drop 1	d1	0.3	Move down by 1	Current player must be above a Ladder or Air tile, and all cells in the line between the current position and new position must be non-solid (i.e. Air or Ladder tile).
Drop 2	d2	0.4	Move down by 2	
Drop 3	d3	0.5	Move down by 3	

Example of glide action validity requirements for GLIDE_RIGHT_2 (‘gr2’):

Current Position	Must be Non-Solid	Must be Non-Solid
Must be Non-Solid	Must be Non-Solid	New Position

Interactive mode

A good way to gain an understanding of the game is to play it. You can play the game to get a feel for how it works by launching an interactive game session from the terminal with the following command:

```
$ python play_game.py <input_file>.txt
```

where `<input_file>.txt` is a valid testcase file (from the support code).

In interactive mode, type the symbol for your chosen action and press enter to perform the action. Type ‘q’ and press enter to quit the game.

DragonGame as an MDP

In this assignment, you will write the components of a program to play DragonGame, with the objective of finding a high-quality solution to the problem using various sequential decision-making algorithms based on the Markov decision process (MDP) framework. This assignment will test your skills in defining a MDP for a practical problem and developing effective algorithms for large MDPs.

What is provided to you

We will provide supporting code in Python only, in the form of:

1. A class representing DragonGame game map and a number of helper functions
2. A parser method to take an input file (testcase) and convert it into a DragonGame map
3. A policy visualiser
4. A simulator script to evaluate the performance of your solution
5. Testcases to test and evaluate your solution
6. A solution file template

The support code can be found at: `https://gitlab.com/3702-2021/a2-support`. Autograding of code will be done through Gradescope, so that you can test your submission and continue to improve it based on this feedback — you are strongly encouraged to make use of this feedback.

Your assignment task

Your task is to develop algorithms for computing paths (series of actions) for the agent (i.e. the Dragon), and to write a report on your algorithms' performance. You will be graded on both your submitted **program (Part 1, 60%)** and the **report (Part 2, 40%)**. These percentages will be scaled to the 15% course weighting for this assessment item.

The provided support code formulates DragonGame as an MDP, and your task is to submit code implementing the following MDP algorithms:

1. Value Iteration (VI)
2. Policy Iteration (PI)
3. Monte Carlo Tree Search (MCTS).

Individual testcases will not impose the use of a particular strategy (value iteration/policy iteration/MCTS), but the difficulty of higher level testcases will be designed to require a more advanced solution (e.g. linear algebra policy iteration or MCTS). Each testcase will specify different amounts of time available for use for offline planning and for online planning, so these can be used to encourage your choice of solution e.g. towards choosing offline or online solving for a particular testcase. You can choose which search approach to use in your code at runtime if desired (e.g. by checking the given offline_time and online_time, and choosing offline planning if offline time $\gg$ online time, online planning otherwise).

Once you have implemented and tested the algorithms above, you are to complete the questions listed in the section "Part 2 - The Report" and submit the report to Gradescope.

More detail of what is required for the programming and report parts are given below.

Part 1 — The programming task

Your program will be graded using the Gradescope autograder, using testcases similar to those in the support code provided at `https://gitlab.com/3702-2021/a2-support`

Interaction with the testcases and autograder

We now provide you with some details explaining how your code will interact with the testcases and the autograder (with special thanks to Nick Collins for his efforts making this work seamlessly).

Implement your solution using the supplied `solution.py` Template file. You are required to fill in the following method stubs:

- `__init__(game_env)`
- `plan_offline()`
- `select_action()`

You can add additional helper methods and classes (either in `solution.py` or in files you create) if you wish. To ensure your code is handled correctly by the autograder, you should avoid using any try-except blocks in your implementation of the above methods (as this can interfere with our time-out handling). Refer to the documentation in `solution.py` for more details.

Grading rubric for the programming component (total marks: 60/100)

For marking, we will use 8 different testcases of ascending level of difficulty to evaluate your solution.

There will be a total of 60 code marks, consisting of: