COMP3702 Artificial Intelligence (Semester 2, 2023)
Assignment 1: Search in DragonGame

Key information:

- **Due: 3pm, Wednesday 23 August 2023**
- This assignment assesses your skills in developing discrete search techniques for challenging problems.
- Assignment 1 contributes 20% to your final grade.
- This assignment consists of two parts: (1) programming and (2) a report.
- This is an individual assignment.
- Both the code and report are to be submitted via Gradescope (`https://www.gradescope.com/`). You can find a link to the COMP3702 Gradescope site on Blackboard.
- Your code (Part 1) will be graded using the Gradescope code autograder, using the testcases in the support code provided at `https://gitlab.com/3702-2023/a1-support`.
- Your report (Part 2) should fit the template provided, be in .pdf format and named according to the format `a1-COMP3702-[SID].pdf`. 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.

To optimally solve a level, your AI agent must find a sequence of actions which collects all gems and reaches the exit while incurring the minimum possible 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
```

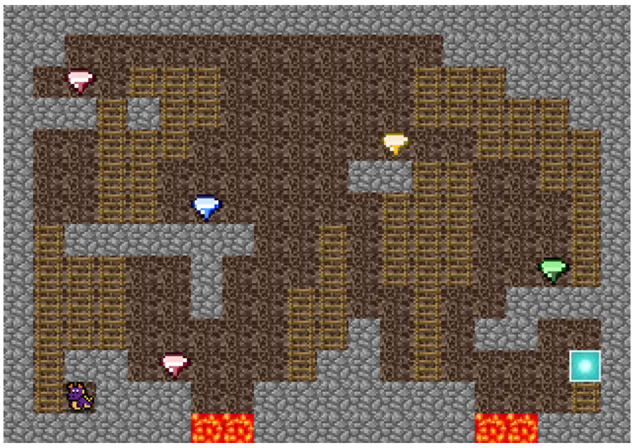

Figure 1: Example game level of DragonGame, showing character-based and GUI visualiser representations

[1]This full game title was inspired by Untitled Goose Game, an Indie game developed by Australian developers in 2019

Game state representation

Each game state is represented as a character array, representing the tile types and their position on the board. In the visualizer and interactive sessions, the tile descriptions are graphical assets, 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	Image in Visualiser	Effect
Solid	`'X'`		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.
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	`'*'`		The player cannot move into a Lava tile. Moving into a tile directly above a Lava tile results in Game Over.
Gem	`'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'`		Moving to the Exit tile after collecting all gems completes the level. Exit tiles behave as 'Air' tiles.
Player	`'P'`		The player starts at the position in the input file where this tile occurs. The player always starts on an 'Air' tile.

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 2.

Table 2: Table of available actions, costs and requirements

Action	**Symbol**	**Cost**	**Description**	**Validity Requirements**
Walk Left	wl	1.0	Move left by 1 position	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	
Jump	j	2.0	Move up by 1 position.	
Glide Left 1	gl1	0.7	Move left by 1 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 2 and down by 1	
Glide Left 3	gl3	1.2	Move left by 3 and down by 1	
Glide Right 1	gr1	0.7	Move right by 1 and down by 1	
Glide Right 2	gr2	1.0	Move right by 2 and down by 1	
Glide Right 3	gr3	1.2	Move right by 3 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 with path relative to the current directory, e.g. `testcases/L1.txt`

In interactive mode, type the symbol for your chosen action (e.g. '`wl`') and press enter to perform the action. Type '`q`' and press enter to quit the game.

DragonGame as a search problem

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 search algorithms. This assignment will test your skills in implementing search algorithms for a practical problem and developing good heuristics to make your program more efficient.

What is provided to you

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

1. A class representing the DragonGame environment and a number of helper functions (in `game_env.py`)
   - The constructor takes an input file (testcase) and converts it into a DragonGame map
2. A class representing the DragonGame game state (in `game_state.py`)
3. A graphical user interface for visualising the game state (in `gui.py`)
4. A solution file template (`solution.py`)
5. A tester (in `tester.py`)
6. Testcases to test and evaluate your solution (in `./testcases`)

The support code can be found at: `https://gitlab.com/3702-2023/a1-support`. Please read the README.md file which provides a description of the provided files. 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 a program that implements search algorithms and outputs the series of actions the agent (i.e. the Dragon) performed to solve the game, and to provide a written report explaining your design decisions and analysing your algorithms' performance. You will be graded on both your submitted **code (Part 1, 60%)** and the **report (Part 2, 40%)**. These percentages will be scaled to the 20% course weighting for this assessment item.

To turn DragonGame into a search problem, you will have to first define the following agent design components:

- A problem state representation (state space),

- A successor function that indicates which states can be reached from a given state (action space and transition function), and
- A cost function (utility function); **the cost of each movement is given in Table 2**

Note that a goal-state test function is provided in the support code. Once you have defined the components above, you are to develop and submit code implementing two discrete search algorithms in the indicated locations of `solution.py`:

1. Uniform-Cost Search (UCS), and
2. A* Search

Note that **your heuristic function used in A* search must be implemented in the** `compute_heuristic` **method and called from your A* method**, and any pre-processing-based heuristics should be implemented in `preprocess_heuristic` (optional). This enables consistent evaluation of your heuristic functions, independent of your A* implementation.

The provided tester can assess your submitted UCS or A* search based on the '`search_type`' argument. Both UCS and A* will be run separately by the autograder, and the heuristic function will also be assessed independently.

Finally, after you have implemented and tested the algorithms above, you are to complete the questions listed in the section "Part 2 - The Report" and submit them as a written report.

More detail of what is required for the programming and report parts are given below. *Hint: Start by implementing a working version of UCS, and then build your A* search algorithm out of UCS using your own heuristics.*

Part 1 — The programming task

Your program will be graded using the Gradescope autograder, using the testcases in the support code provided at `https://gitlab.com/3702-2023/a1-support`.

Interaction with the testcases and autograder

We now provide 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). Your solution code only needs to interact with the autograder via your implementation of the methods in the Solver class of `solution.py`. Your search algorithms, implemented in `search_ucs` and `search_a_star`, should return the path found to the goal (i.e. the list of actions, where each action is an element of GameEnv.ACTIONS).

This is handled as follows:

- The file `solution.py`, supplied in the support code, is a template for you to write your solution. All of the code you write can go inside this file, or if you create your own additional python files they must be invoked from this file.
- The script `tester.py` can be used to test your code on testcases.
  After you have implemented UCS (uniform cost search) and/or A* search in `solution.py` you can test them by going to your command prompt, navigating to your folder and running `tester.py`:
  Usage:

  ```
  $ python tester.py [search_type] [testcase_file] [-v (optional)]
  ```

  - search_type = 'ucs', 'a_star'
  - testcase_file = a filename of a valid testcase file with path relative to the `tester.py` script (e.g. testcases/L1.txt)
  - if -v is specified, the solver's trajectory will be visualised

For example, to test UCS after you have written the code for it, you can type the following in the command prompt:

```
$ python tester.py ucs testcases/L1.txt -v
```

Note that the GameEnv class constructor (`__init__(self, filename)`) handles reading the input file and is called from `tester.py`

- The *autograder* (hidden to students) handles running your python program with all of the testcases. It will run your submitted `solution.py` code and assign a mark for each testcase.
- You can inspect the testcases in the support code, which each include information on their optimal solution cost and test time limits. Looking at the testcases might also help you develop heuristics using your human intelligence and intuition.
- To ensure your submission is graded correctly, write your solution code in `solution.py`, and do not rename any of the provided files or alter the methods in `game_env.py` or `game_state.py`.

More detailed information on the DragonGame implementation is provided in the Assignment 1 Support Code *README.md*, while a high-level description is provided in the DragonGame AI Environment description document.

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

For marking, we will use 6 testcases of approximately ascending level of difficulty to evaluate your solution.

Note that a *valid solution* means returning a list of actions that complete a testcase (i.e. collect all gems and get to the exit), while the *optimal path cost* refers to returning a list of actions which provide the minimum cost solution.

There are 10 criteria for each testcase based on the accuracy and efficiency of your solution, each worth 1 mark:

- UCS valid solution
- UCS path cost (vs optimal cost)
- UCS run time (vs benchmark)
- A* heuristic admissible
- A* heuristic path cost when used with reference A* (vs optimal cost)
- A* heuristic run time when used with reference A* (vs benchmark)
- A* heuristic nodes expanded when used with reference A* (vs benchmark)
- A* search valid solution
- A* search path cost (vs optimal cost)
- A* search run time (vs benchmark)

Partial marks are available for the path cost, run time and nodes expanded criteria. For each case, a minimum target (where scores worse than the minimum receive 0 marks) and a maximum target (where scores better than the maximum receive full marks) are provided; scores between the minimum and maximum targets are interpolated linearly.

There will be a total of 60 code marks.

Part 2 — The report

The report tests your understanding of the methods you have used in your code, and contributes 40/100 of your assignment mark. **Please make use of the report templates provided on Blackboard**, because Gradescope makes use of a predefined assignment template. Submit your report via Gradescope, in .pdf format (i.e. use "save as pdf" or "print-to-file" functionality), and named according to the format `a1-COMP3702-[SID].pdf`. Reports will be graded by the teaching team.

Your report task is to answer the questions below:

Question 1. **(5 marks)**
State the ten dimensions of complexity in DragonGame, and briefly explain your selection.

Refer to the P&M textbook `https://artint.info/3e/html/ArtInt3e.Ch1.S5.html` (and tabular summary in Figure 1.8) for a description of each dimension (modularity, planning horizon, representation, computational limits, learning, sensing uncertainty, effect uncertainty, preference, number of agents, and interactivity).

Question 2. **(5 marks)**
Describe the components of your agent design for DragonGame. Specifically, the Action Space, State Space, Transition Function and Utility Function.

Question 3. **(15 marks)**
Compare the performance of Uniform Cost Search and A* search in terms of the following statistics, presenting the numerical results in tabular format:

a) The number of nodes on the fringe/frontier when the search terminates

b) The number of nodes explored/expanded when the search terminates

c) The run time of the algorithm. Note that you can report run-times from your own machine, not the Gradescope servers.

Discuss and interpret these results. In your discussion, you may describe the purpose of the heuristic function $h(n)$ in A* search, whether it achieved its intended benefits and any trade-offs.

Question 4. **(15 marks)**
Some challenging aspects of designing a DragonGame agent are the asymmetric movement dynamics (moving up behaves differently to moving down), the problem of choosing the order in which to visit and collect each gem, and the large number and differing cost of available actions.

Describe the heuristics (or components of a combined heuristic function) that you have developed in the DragonGame search task that account for these aspects or any other challenging aspects you have identified of the problem. Your documentation should provide a thorough explanation of the rationale for using your chosen heuristic(s) considering factors such as admissibility and computational complexity.

References and Generative AI
At the end of your report, please cite any references, and if you utilised Generative AI to assist in producing your solution or report, please describe how you utilised such tools and how you verified the answers, citing the version and date.

Academic Misconduct

The University defines Academic Misconduct as involving "a range of unethical behaviours that are designed to give a student an unfair and unearned advantage over their peers." UQ takes Academic Misconduct very seriously and any suspected cases will be investigated through the University's standard policy (`https://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct`). If you are found guilty, you may be expelled from the University with no award.

It is the responsibility of the student to ensure that you understand what constitutes Academic Misconduct and to ensure that you do not break the rules. If you are unclear about what is required, please ask.

It is also the responsibility of the student to take reasonable precautions to guard against unauthorised access by others to his/her work, however stored in whatever format, both before and after assessment.

In the coding part of COMP3702 assignments, you are allowed to draw on publicly-accessible resources and provided tutorial solutions, but you must make reference or attribution to its source, by doing the following:

- All blocks of code that you take from public sources must be referenced in adjacent comments in your code or at the top of your code.
- Please also include a list of references indicating code you have drawn on in your `solution.py` docstring.

If you have utilised Generative AI tools such as ChatGPT, you must cite the tool and version in your code as well as describe in the report. A failure to reference generative AI use may constitute student misconduct under the Student Code of Conduct.

However, you must not show your code to, or share your code with, any other student under any circumstances. You must not post your code to public discussion forums (including Ed Discussion) or save your code in publicly accessible repositories (check your security settings). You must not look at or copy code from any other student.

All submitted files (code and report) will be subject to electronic plagiarism detection and misconduct proceedings will be instituted against students where plagiarism or collusion is suspected. The electronic plagiarism detection can detect similarities in code structure even if comments, variable names, formatting etc. are modified. If you collude to develop your code or answer your report questions, you will be caught.

For more information, please consult the following University web pages:

- Information regarding Academic Integrity and Misconduct:
  - https://my.uq.edu.au/information-and-services/manage-my-program/student-integrity-and-conduct/academic-integrity-and-student-conduct
  - http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct
- Information on Student Services:
  - https://www.uq.edu.au/student-services/

Late submission

Students should not leave assignment preparation until the last minute and must plan their workloads to meet advertised or notified deadlines. It is your responsibility to manage your time effectively.

It may take the autograder up to an hour to grade your submission. It is your responsibility to ensure you are uploading your code early enough and often enough that you are able to resolve any issues that may be revealed by the autograder *before the deadline*. Submitting non-functional code just before the deadline, and not allowing enough time to update your code in response to autograder feedback is not considered a valid reason to submit late without penalty.

Assessment submissions received after the due time (or any approved extended deadline) will be subject to a 100% late penalty. A one-hour grace period will be applied to the due time after which time the 100% late penalty will be imposed. This grace period is designed to deal with issues that might arise during submission (e.g. delays with Blackboard or Turnitin) and should not be considered a shift of the due time. Please keep a record of your submission time.

In the event of exceptional circumstances, you may submit a request for an extension. You can find guidelines on acceptable reasons for an extension here https://my.uq.edu.au/information-and-services/manage-my-program/exams-and-assessment/applying-extension. All requests for extension must be submitted on the UQ Application for Extension of Assessment form at least 48 hours prior to the submission deadline.