

COMP3702 Artificial Intelligence (Semester 2, 2025)

Assignment 1: Search in CHEESEHUNTER

Key information:

- **Due Date: 1pm, Friday 29 August 2025**
- 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://github.com/comp3702/A1-Support-Code-2025>.
- 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 CHEESEHUNTER AI Environment

CHEESEHUNTER is a 2.5D platformer game in which the mouse player must activate all the levers in a level before collecting a wedge of cheese. The player must choose which actions to perform, including walking, sprinting, climbing, jumping and dropping movements, to navigate a maze including drawbridges and trapdoors (i.e., “traps”). The traps are toggled between locked and unlocked by activating their corresponding levers using the activate action. Each action incurs a different cost, which requires clever problem solving to not only complete a level, but also solve it with the minimum cost!

Your task is to design an AI agent to optimally solve levels through Search, finding a sequence of actions for the player to activate all levers and then reach the cheese, while minimising the total action cost.

Levels in CHEESEHUNTER are composed of a 2D grid of tiles, where each tile contains a character representing the tile type as well as a schematic indicating which levers correspond to which traps based on matching numbers (i.e., lever 1 toggles trap 1). In the graphical visualiser, the blue levers correspond to Trapdoors, while red levers correspond to Drawbridges, and are numbered sequentially from 0 for the two trap types. An example game level is shown in Figure 1.

Grid data Lever-Trap Schematic

```

XXXXXXXXXXXXXXXXXX
X                    LX
X=XXXX=XDXXXX=XX
X=XXXX=L  XG  = X
XP  LXXXX=XXXX=X
XXXTXXXX=X    =X
XL   X   =X=X= =X
XXXXTX   = =X= =X
X       =XDXXXX=X
XL      =   L  =X
XXXX=XXXXXXTXXXX
X   =           X
XXXXXXXXXXXXXXXXXX

```

```

                    5
                1
                3
                2
                3
            1
                2
                    4
            4        6
                5 6

```

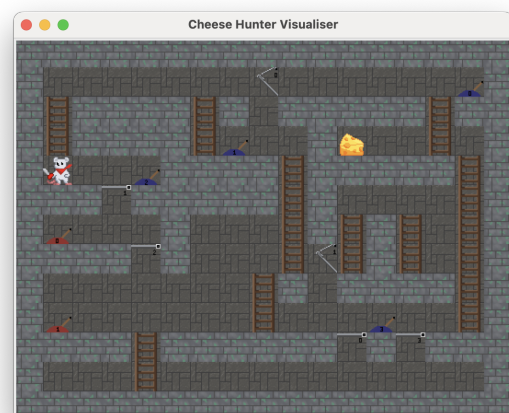


Figure 1: Example level of CHEESEHUNTER, showing character-based and GUI visualiser representations.




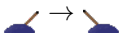
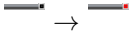
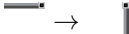
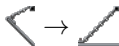


Game state representation

Each game state is represented by a character array representing the tile types and their position on the board, as well as the state of each trap (whether it has been locked or not, by activating its corresponding lever).

In the visualizer, 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 CHEESEHUNTER, 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, jump and sprint actions are valid when the player is directly above a Solid tile.
Ladder	'='		The player can move through Ladder tiles. Walk, sprint and drop actions are all valid when the player is directly above a Ladder tile, while climb is valid when on a Ladder tile.
Air	' '		The player can move through Air tiles. The drop action is valid when the player is directly above an Air tile.
Lever	'L'		<i>Unactivated → Activated</i> The player can move through Lever tiles. Blue levers correspond to Trapdoors, while red levers correspond to Drawbridges, and are numbered sequentially from 0 for the two trap types. When the player performs the activate action on a Lever tile it will toggle the lever from unactivated/activated and alter the status of the corresponding trap from unlocked/locked. The player must activate all levers in order to complete the level.
Trapdoor	'T'		<i>Closed (unlocked) → Closed (locked)</i> The player can move over Trapdoor tiles with walk, jump and sprint actions only when the Trapdoor is locked (marked with red). Trapdoors are locked by activating their corresponding lever.
			<i>Closed (unlocked) → Open (unlocked)</i> When the player stands on an unlocked Trapdoor, it will open and the only valid actions are drop or activate. The Trapdoor closes again after the player moves out of the way.
Drawbridge	'D'		<i>Closed (unlocked) → Open (locked)</i> The player can only move onto or through a Drawbridge tile when it is Open, which is performed by activating its corresponding lever.
Goal	'G'		Moving to the Goal tile (the cheese) after activating all levers completes the level. Goal tiles behave as 'Air' tiles.
Player	'P'		The player (mouse) 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	Player must be above a Solid or Ladder tile or locked Trapdoor/Drawbridge, 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	
Climb	c	2.0	Move up by 1 position	Player must be on a Ladder Tile, and new player position must not be a Solid tile.
Sprint Left	sl	1.9	Move left by 2	Player must be above a Solid or Ladder or locked Trapdoor/Drawbridge tile, and new player position must not be a Solid tile. In addition, the tile that the player is sprinting through must also be above a Solid or Ladder or locked Trapdoor/Drawbridge tile.
Sprint Right	sr	1.9	Move right by 2	
Drop	d	0.5	Move down by 1	Player must be above a Ladder or Air or unlocked Trapdoor tile, and new player position must not be a Solid tile.
Activate	a	1.0	Activate a lever	This action is valid in all tiles, however, it will only toggle the lever and corresponding trap when performed on a Lever tile. The lever then changes status from unactivated/activated, and locks/unlocks the corresponding trap (Trapdoor/Drawbridge).

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/level_1.txt`

Depending on your python installation, you should run the code using `python`, `python3` or `py`.

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

CHEESEHUNTER as a search problem

In this assignment, you will write the components of a program to play CHEESEHUNTER, 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 CHEESEHUNTER environment and a number of helper functions (in `game_env.py`)
2. A class representing the CHEESEHUNTER 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://github.com/comp3702/A1-Support-Code-2025>. 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 Mouse) 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 CHEESEHUNTER into a search problem, you should first study the supplied code and documentation and identify where and how the following agent design components are defined:

- 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 (the opposite of the utility function)

Note that a goal-state test function is provided in the support code (in the `is_solved()` method of the environment). Once you have identified 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://github.com/comp3702/A1-Support-Code-2025>.

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 Winston Niogret for the game design and 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/level_1.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/level_1.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 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 CheeseHunter implementation is provided in the Assignment 1 Support Code *README.md*, while a high-level description is provided in the CHEESEHUNTER AI Environment description document.

Autograder limits: Gradescope has memory and time limits. While these are not part of your agent design, they may result in your code failing to run properly. To restrict which test cases to evaluate on Gradescope (e.g. disable large testcases which have issues with RAM usage), you can modify the `get_testcases()` method in `solution.py`.

```
@staticmethod
def get_testcases():
    """
    Select which testcases you wish the autograder to test you on.
    The autograder will not run any excluded testcases.
    e.g. [1, 4, 6] will only run testcases 1, 4, and 6, excluding, 2, 3, and 5.
    :return: a list containing which testcase numbers to run (testcases in 1-6).
    """
    return [1, 2, 3, 4, 5, 6]
```

This method is executed on Gradescope in the autograder, and tests will only be run for the test cases returned by this method. For example, if `get_testcases()` returns `[1, 2]`, only test cases for level 1 and 2 will be evaluated on Gradescope, and your score will be capped accordingly based on the number of levels tested. This feature helps address potential time-out issues caused by inefficient implementations that fail to solve within the allowed time. If you encounter such problems, this method enables you to select and submit test cases you're confident will pass.

In addition, you can select which search you wish the autograder to run (`ucs`, `a_star`, or `both`) via the `get_search()` method ("`ucs`" to only run UCS, "`a_star`" to only run A*, and "`both`" to run both).

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. activate all levers and get to the cheese), 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, for a total of 60 code marks:

- 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.

Note: If your A* implementation is found to be exploiting the autograder by submitting Uniform Cost Search for your A* search, or using a trivial heuristic such as a constant value, your A* search code will receive a score of 0. Similarly, you will score 0 if you have not implemented UCS, or made illegal modifications to the environment. For the programming component of the assignments, the teaching staff will conduct interviews with a subset of students about their submissions for the purpose of establishing genuine authorship, as per the Course Profile.

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, 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)

Define and justify your selection of the following eight dimensions of complexity of CHEESEHUNTER: planning horizon, representation, computational limits, learning, sensing uncertainty, effect uncertainty, number of agents, and interactivity. Refer to the P&M textbook <https://artint.info/3e/html/ArtInt3e.Ch1.S5.html> (and tabular summary in Figure 1.8) for the possible values and description of each dimension.

Rubric:

0.25 marks for each correct value x8
0.25 marks for correct justification x8
1 mark for neatly formatted table

Question 2. (5 marks)

Describe the components of your Agent Design for CHEESEHUNTER. Specifically, define the Action Space, State Space, Transition Function and Utility/Cost Function generally, and what these components are for the CHEESEHUNTER agent design problem. Refer to the methods and definitions in the support code to support your answer.

Rubric:

1.25 marks for each component's general definition and definition in CHEESEHUNTER, incl. reference to code x4

Question 3. (15 marks)

Compare the performance of Uniform Cost Search and A* search in terms of the following statistics for each testcase in a table:

- The number of nodes visited/reached when the search terminates
- The number of nodes explored/expanded when the search terminates
- The number of nodes on the frontier when the search terminates
- The run time of the algorithm. Note that you can report run-times from your own machine, and do not need to use the Gradescope servers.

You must report the numerical results for (a)-(d) in a neatly formatted table (not screenshots from your computer or Gradescope). Then, discuss and interpret these results. In your discussion, you must first describe the difference between the UCS and A* algorithms, including the purpose of the heuristic function in A* search, and the expected difference in number of explored states and run time of the algorithms. Then discuss whether A* achieved the intended benefits of A* over UCS and if not, what trade-offs or flaws may exist in your heuristic. In this question you do not need to describe the details of your heuristic.

Rubric:

5 marks: Complete numerical results presented for all testcases and algorithms, neatly formatted in a table
5 marks: Insightful explanation and excellent understanding of intended difference between algorithms, identifying purpose of heuristic function and expected difference in number of explored states and run time between the algorithms
5 marks: Discussion and interpretation of results: Comparison of numerical performance between algorithms and testcases, and discussion of how this compares to expectation, including trade-offs in heuristic design.

Question 4.

(15 marks)

Some challenging aspects of designing a CHEESEHUNTER agent are the large number and differing cost of available actions, the asymmetric movement dynamics (moving up behaves differently to moving down), sprinting is more efficient than walking, and the problem of choosing the order in which to activate the levers.

Describe the heuristics or components of a combined heuristic function that you have developed in the CHEESEHUNTER search task that account for these aspects or any other challenging aspects you have identified of the problem. Your documentation must provide an explanation of the rationale for your chosen heuristics, describing factors including admissibility, computational complexity, and accuracy of the heuristic.

Rubric:

5 marks: Choice and justification of heuristic(s) demonstrate insightful consideration about the structure of the problem , clearly written and formatted, demonstrating expert heuristic design for the environment.
5 marks: Choice and justification of heuristic(s) demonstrate high level of understanding and consideration of admissibility
5 marks: Choice and justification of heuristic(s) demonstrate high level of understanding and consideration of computational complexity and accuracy

Approximate mark breakdown:

1. Design

- What are the components of your heuristic (e.g. distance computation, factoring in different costs of actions, incorporating the position and order of levers/traps) [2]
- Use them as **subheadings** [-2 for no subheadings]
 - Provide the formula or image explaining each [1]
 - The rationale behind choosing each [2]
- If you didn't get a working heuristic
 - You may include those that you explored but didn't end up using and reasons you left them out to demonstrate understanding
 - Were there any ideas that you were unable to implement? Justify how and why they would improve your heuristic

2. Admissibility

- State the definition of admissibility [1]
- Identify whether your heuristic is admissible [1]
- Argue why your heuristic is admissible, or reason about why your heuristic is not if you believe that's the case. You would need to give out a concrete counterexample for a non-admissible heuristic, and explain which component(s) caused it. [3]

3. Computational Complexity and Accuracy

- What are the steps you did? [1.5] (Caching / Pre-computation / Cost Estimation etc.)
- Use them as **subheadings** [-2 for no subheadings]
 - Give out time and space complexity for each [0.5]
 - Justify doing each [1]
- How accurate/ inaccurate is it and why? Any tradeoffs between computational complexity and accuracy? [2 marks]

4. Page limit: your answer must fit within a maximum limit of 2 pages [-2 for over page limit]

References and Generative AI

At the end of your report, ensure you reference any sources (e.g. using IEEE or APA referencing style). If you utilised Generative AI to assist in producing your solution or report, write a GenAI declaration, briefly describing what tool you used and how you used it, citing the version and date. See the link in the Academic Misconduct section.

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.

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, in comments next to the referenced code, and include a list of references you have drawn on in your `solution.py` docstring.

If you have utilised **Generative AI** tools such as ChatGPT, you must clearly cite any use of generative AI in each instance. To reference your use of AI see:

- <https://guides.library.uq.edu.au/referencing/chatgpt-and-generative-ai-tools>

Failure to reference use of generative AI tools constitutes student misconduct under the Student Code of Conduct.

It is 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. 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 late penalty of 10% per 24 hours of the maximum possible mark for the assessment item.

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>, and submit the UQ Application for Extension of Assessment form.