COMP3702 Artificial Intelligence (Semester 2, 2024)
Assignment 1: Search in BeeBot

Key information:

- **Due: 1pm, Friday 23 August 2024**
- 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 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/comp3702/2024-Assignment-1-Support-Code`.
- 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 BeeBot AI Environment

You have been tasked with developing a search algorithm for automatically controlling BeeBot, a Bee which operates in a hexagonal environment, and has the capability to push, pull and rotate honey 'Widgets' in order to reposition them to target honeycomb locations. To aid you in this task, we have provided support code for the BeeBot environment which you will interface with to develop your solution. To optimally solve a level, your AI agent must efficiently find a sequence of actions so that every Target cell is occupied by part of a Widget, while incurring the minimum possible action cost.

Levels in BeeBot are composed of a Hexagonal grid of cells, where each cell contains a character representing the cell type. An example game level is shown in Figure 1.

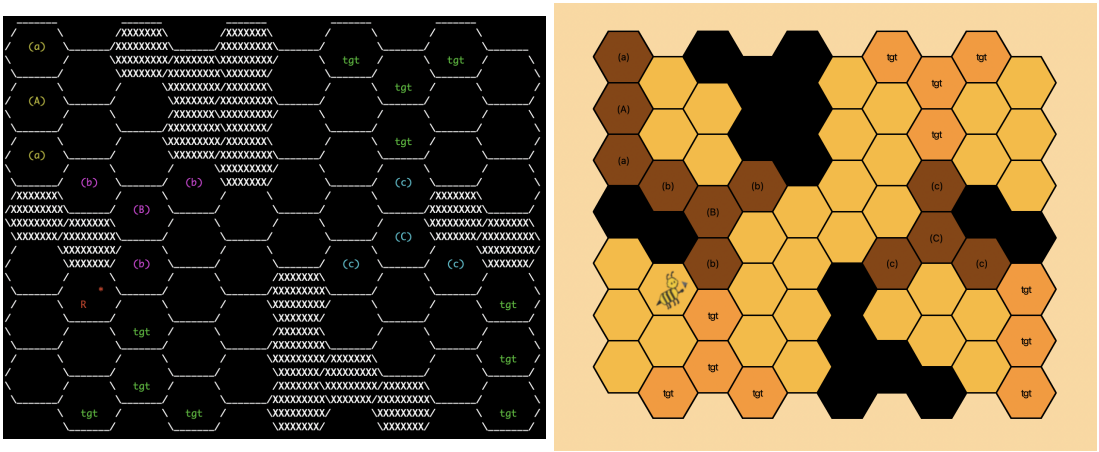

Figure 1: Example game level of BeeBot, showing character-based and GUI visualiser representations

Environment representation

Hexagonal Grid

The environment is represented by a hexagonal grid. Each cell of the hex grid is indexed by (row, column) coordinates. The hex grid is indexed top to bottom, left to right. That is, the top left corner has coordinates $(0, 0)$ and the bottom right corner has coordinates $(n_{rows} - 1, n_{cols} - 1)$. Even numbered columns (starting from zero) are in the top half of the row, and odd numbered columns are in the bottom half of the row. An example is shown in Figure 2.

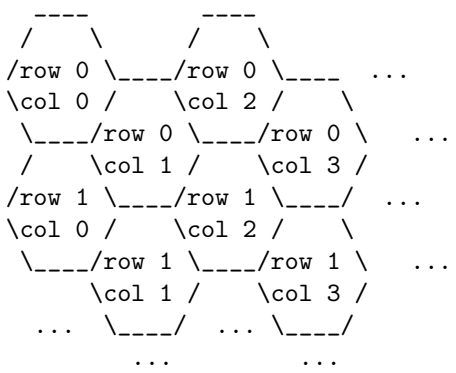

Figure 2: Example hexagonal grid showing the order that rows and columns are indexed

Two cells in the hex grid are considered adjacent if they share an edge. For each non-border cell, there are 6 adjacent cells.

The BeeBot Agent and its Actions

The BeeBot Bee occupies a single cell in the hex grid. In the visualisation, the Bee is represented by the cell marked with the character 'R' (or the Bee in the graphical visualiser). The side of the cell marked with '*' (or the arrow in the graphical visualiser) represents the front of the Bee. The state of the Bee is defined by its (row, column) coordinates and its orientation (i.e. the direction its front side is pointing towards).

At each time step, the agent is prompted to select an action. The Bee has 4 available actions:

- Forward → move to the adjacent cell in the direction of the front of the Bee (keeping the same orientation)
- Reverse → move to the adjacent cell in the opposite direction to the front of the Bee (keeping the same orientation)
- Spin Left → rotate left (relative to the Bee's front, i.e. counterclockwise) by 60 degrees (staying in the same cell)
- Spin Right → rotate right (i.e. clockwise) by 60 degrees (staying in the same cell)

The Bee is equipped with a gripper on its front side which allows it to manipulate honey Widgets. When the Bee is positioned with its front side adjacent to a honey Widget, performing the 'Forward' action will result in the Widget being pushed, while performing the 'Reverse' action will result in the honey Widget being pulled.

Action Costs

Each action has an associated cost, representing the amount of energy used by performing that action.

If the Bee moves without pushing or pulling a honey widget, the cost of the action is given by a base action cost, `ACTION_BASE_COST[a]` where 'a' is the action that was performed.

If the Bee pushes or pulls a honey widget, an additional cost of `ACTION_PUSH_COST[a]` is added on top, so the total cost is `ACTION_BASE_COST[a] + ACTION_PUSH_COST[a]`.

The costs are detailed in the `constants.py` file of the support code:

```
ACTION_BASE_COST = {FORWARD: 1.0, REVERSE: 1.0, SPIN_LEFT: 0.1, SPIN_RIGHT: 0.1}
ACTION_PUSH_COST = {FORWARD: 0.8, REVERSE: 0.5, SPIN_LEFT: 0.0, SPIN_RIGHT: 0.0}
```

Obstacles
Some cells in the hex grid are obstacles. In the visualisation, these cells are filled with the character 'X' (represented as black cells in the graphical visualiser). Any action which causes the Bee or any part of a honey Widget to enter an obstacle cell is invalid (i.e. results in collision). The outside boundary of the hex grid behaves in the same way as an obstacle.

Widgets
Honey Widgets are objects which occupy multiple cells of the hexagonal grid, and can be rotated and translated by the BeeBot Bee. The state of each honey widget is defined by its centre position (row, column) coordinates and its orientation. Honey Widgets have rotational symmetries - orientations which are rotationally symmetric are considered to be the same.

In the visualisation, each honey Widget in the environment is assigned a unique letter 'a', 'b', 'c', etc. Cells which are occupied by a honey Widget are marked with the letter assigned to that Widget (surrounded by round brackets). The centre position of the honey Widget is marked by the uppercase version of the letter, while all other cells occupied by the widget are marked with the lowercase.

Three honey widget types are possible, called Widget3, Widget4 and Widget5, where the trailing number denotes the number of cells occupied by the honey Widget. The shapes of these three honey Widget types and each of their possible orientations are shown in Figures 3 to 5 below.

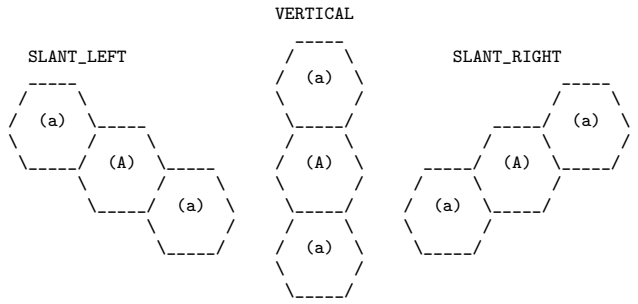

Figure 3: Widget3

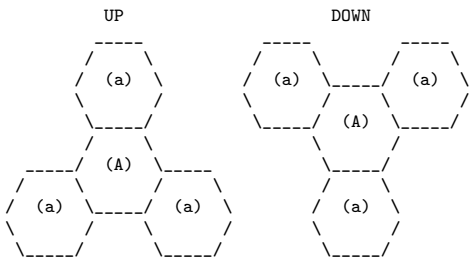

Figure 4: Widget4

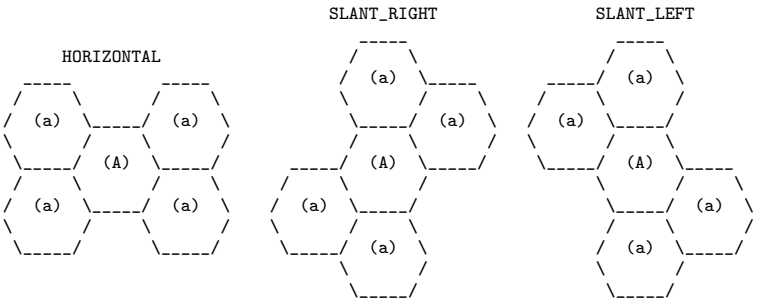

Figure 5: Widget5

Two types of widget movement are possible – translation (change in centre position) and rotation (change in orientation).

Translation occurs when the Bee is positioned with its front side adjacent to one of the honey widgets' cells such that the Bee's orientation is in line with the honey widget's centre position. Translation results in the centre position of the widget moving in the same direction as the Bee. The orientation of the honey Widget does not change when translation occurs. Translation can occur when either 'Forward' or 'Reverse' actions are performed. For an action which results in translation to be valid, the new position of all cells of the moved widget must not intersect with the environment boundary, obstacles, the cells of any other honey Widgets or the Bee's new position.

Rotation occurs when the Bee's current position is adjacent to the centre of the widget but the Bee's orientation does not point towards the centre of the widget. Rotation results in the honey widget spinning around its centre point, causing the widget to change orientation. The position of the centre point does not change when rotation occurs. Rotation can only occur for the 'Forward' action - performing 'Reverse' in a situation where 'Forward' would result in a widget rotation is considered invalid.

The following diagrams show which moves result in translation or rotation for each honey Widget type, with the arrows indicating directions from which the Bee can push or pull a widget in order to cause a translation or rotation of the widget. Pushing in a direction which is not marked with an arrow is considered invalid.

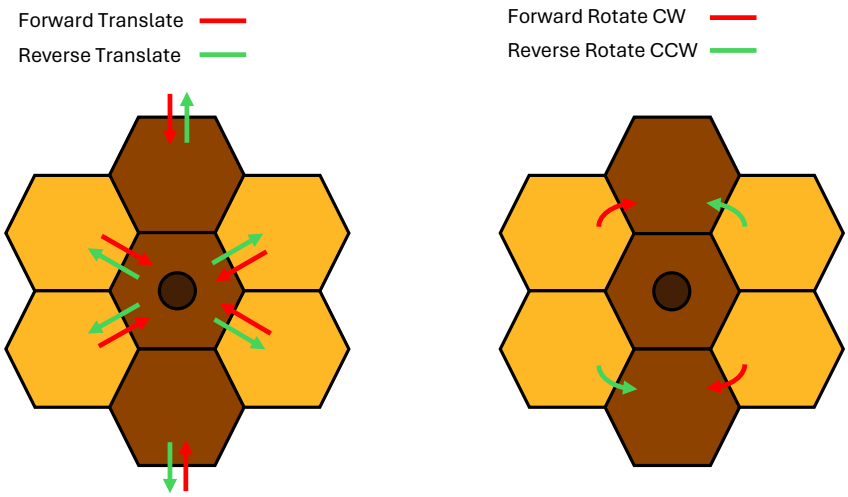

Figure 6: Widget3 translations and rotations

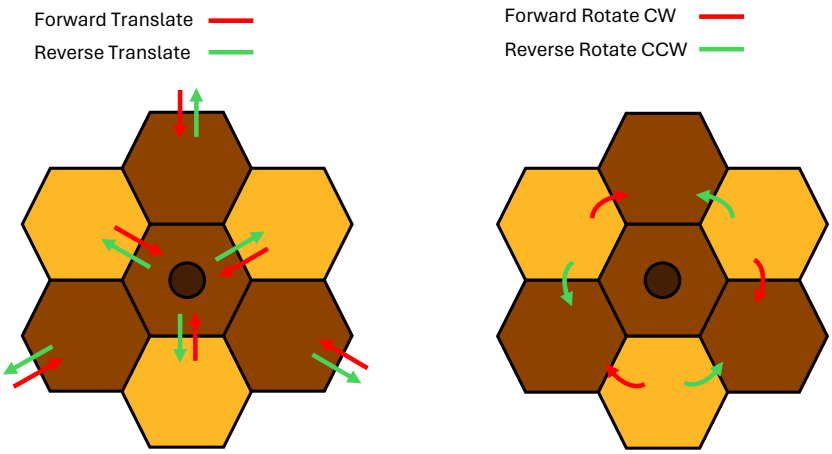

Figure 7: Widget4 translations and rotations

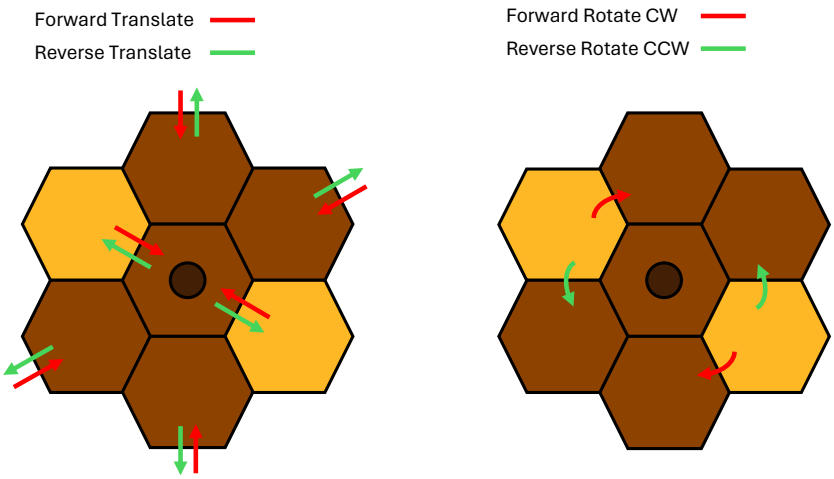

Figure 8: Widget5 translations and rotations

Targets
The hex grid contains a number of 'target' honeycomb cells which must be filled with honey. In the visualisation, these cells are marked with 'tgt' (cells coloured in orange in the graphical visualiser). For a BeeBot environment to be considered solved, each target cell must be occupied by part of a honey Widget. The number of targets in an environment is always less than or equal to the total number of cells occupied by all honey Widgets.

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 for the graphical visualiser:

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

or the following command for the command-line ASCII-character based visualiser:

```
$ python play.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/ex1.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 (and press enter in the command line version) to perform the action: press 'W' to move the Bee forward, 'S' to move the Bee in reverse, 'A' to turn the Bee left (counterclockwise) and 'D' to turn the Bee right (clockwise). Use 'Q' to quit the simulation, and 'R' to reset the environment to the initial configuration.

BeeBot as a search problem

In this assignment, you will write the components of a program to play BeeBot, 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 BeeBot game map and a number of helper functions
2. A parser method to take an input file (testcase) and convert it into a BeeBot map
3. A state visualiser
4. A tester
5. Testcases to test and evaluate your solution
6. A solution file template

The support code can be found at: `https://github.com/comp3702/2024-Assignment-1-Support-Code`. Autograding of code will be done through Gradescope, so that you can evaluate your submission and continue to improve it based on this feedback — you are strongly encouraged to make use of this feedback. You can also test locally using the supplied `tester.py` file.

Your assignment task

Your task is to develop a program that implements search algorithms and outputs the series of actions for the agent (i.e. the BeeBot) to solve game levels. In addition, you will 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 BeeBot 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:

- The problem state representation (state space),
- The successor function that returns which states can be reached from a given state (action space and transition function), and
- The 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 `environment.py`). 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, 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.

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/comp3702/2024-Assignment-1-Support-Code`.

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 and Aryaman Sharma). 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 `solver.solve_ucs()` and `solver.solve_a_star()` (and associated heuristic methods `compute_heuristic` and optionally `preprocess_heuristic`) and should return the path found to the goal (i.e. the list of actions, where each action is an element of BEE_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] [testcases] [-v (optional)]
  ```

  - search_type = 'ucs', 'a_star' or 'both'
  - testcases = a comma separated list of numbers of which test cases to run (can be 1 up to 5) (e.g. '1,3,4')
  - if -v is specified, the solver's trajectory will be visualised

  For example, to test your UCS implementaiton, you can type the following in the command prompt:

  ```
  $ python tester.py ucs 1 -v
  ```

- The *autograder* on Gradescope (hidden to students) handles running your python program with all of the testcases. It will run the tester python program on your solution code and assign a mark for each testcase based on the return code of tester.
- You can inspect the testcases in the support code, which each include information on their optimal solution cost (target cost), target times and target number of nodes expanded (for UCS and A*). 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 `environment.py` or `state.py`.

More detailed information on the BeeBot implementation is provided in the Assignment 1 Support Code *README.md* and code comments.

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

For marking, we will use 5 test cases each run twice (for UCS and A*) to evaluate your solution.

Each test case is scored out of 6.0 marks, divided evenly into 1.5 marks for each category for UCS and 1.25 marks for each category for A* plus 5 marks for the evaluation of your heuristic function.

- **For UCS**, 5 testcases, each worth 6 marks:
  - Completion: 1.5 marks
  - Path Cost: 1.5 marks
  - Time Elapsed: 1.5 marks
  - Nodes Expanded: 1.5 marks
- **For A***, 5 testcases, each worth 5 marks:
  - Same criteria as UCS but each testing point is worth 1.25 marks
  - An additional hidden test is added for evaluating your heuristic, worth 5 marks overall
- Each test case has a single target for Path Cost (applied to both UCS and A*)
- Each test case has separate targets for Time Elapsed and #Nodes expanded for UCS and A* (where the targets for A* are harder to achieve)
- Maximum score is achieved when your program matches or beats the target in each category
- Partial marks are available for up to 2x cost, 2x time elapsed and 2x number of nodes expanded
- Total mark for the test case is a weighted sum of the scores for each category
- Total code mark is the sum of the marks for each test case

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, 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 Electronic 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, in the report template provided.

Question 1. **(5 marks)**
Define the following eight dimensions of complexity of BeeBot: planning horizon, representation, computational limits, learning, sensing uncertainty, effect uncertainty, number of agents, and interactivity. Justify your selection, referring to the P&M textbook `https://artint.info/3e/html/ArtInt3e.Ch1.S5.html` 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 BeeBot. Specifically, define the Action Space, State Space, Transition Function and Utility/Cost Function generally, and what these components are for the BeeBot 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 BeeBot x4

Question 3. **(15 marks)**
Compare the performance of Uniform Cost Search and A* search in terms of the following statistics for each testcase:

a) The number of nodes visited/reached when the search terminates
b) The number of nodes explored/expanded when the search terminates
c) The number of nodes on the frontier when the search terminates
d) 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 should report the numerical results for (a)-(d) in a neatly formatted table (not screenshots from your computer or Gradescope). Discuss and interpret these results. In your discussion, start by describing the difference between the UCS and A* algorithms, including the purpose of the heuristic function, $h(n)$, 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 the 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, how this compares to expectation, and discussion of differences to expectation and trade-offs.

Question 4. **(15 marks)**
Some challenging aspects of designing a BeeBot agent are the hexagonal grid, the asymmetric cost of actions (pushing is more expensive than pulling a widget), rotation of widgets to avoid obstacles, estimating the order in which to manoeuvre each widget, and which target squares to cover.

Describe your heuristic or components of a combined heuristic function that you have developed in the BeeBot search task that account for these aspects or any other challenging aspects you have identified of the problem. Your documentation needs to 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 (e.g. with headings for each component of the heuristic)
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**

References and Generative AI
At the end of your report, you must cite any references (e.g. using IEEE or APA referencing style). If you utilised Generative AI to assist in producing your code or report, briefly describe what tool you used and how you used it, citing the version and date. e.g. "This work was corrected using Copilot (Microsoft, `https://copilot.microsoft.com/`) on 30 July 2024." 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.