COMP3702 Artificial Intelligence
Semester 2, 2025
Tutorial 1

Before you begin, please note:

- Tutorial exercises are provided to help you understand the materials discussed in class, and to improve your skills in solving AI problems.
- Tutorial exercises will not be graded. However, you are highly encouraged to do them for your own learning. Moreover, we hope you get the satisfaction from solving these problems.
- The skills you acquire in completing the tutorial exercises will help you complete the assignments.
- You'll get the best learning outcome when you try to solve these exercises on your own first (before your tutorial session), and use your tutorial session to ask about the difficulties you face when trying to solve this set of exercises.

This tutorial covers material from Module 0 and the start of Module 1 of COMP3702.

To solve an **agent design problem**, the following components are required:

- **Action Space** (A): The set of all possible actions the agent can perform (sometimes called the *action set* in the discrete case). An action is denoted $a \in A$.
- **Percept Space** (P): The set of all possible things the agent can perceive or sense. This is also sometimes called the *observation space*.
- **State Space** (S): The set of all possible configurations of the world the agent is operating in (sometimes called the *set* of states in discrete state systems). A state is denoted $s \in S$.
- **World Dynamics/Transition Function** ($T : S \times A \to S'$): A function that specifies how the world changes when the agent performs actions in it; a system model. We sometimes write $T(s, a) = s'$. When the transition function is stochastic, it can be denoted as a conditional probability distribution, e.g. $P(s'|s, a)$, which we will see later in the course.
- **Perception Function** ($Z : S \to P$): A function that maps a world state to a perception.
- **Utility Function** ($U : S \to \mathbb{R}$): A function that maps a state (or a sequence of states) to a real number, indicating how desirable it is for the agent to occupy that state/sequence of states. We sometimes write $U(s) =$ some reward value, which we want to maximise. In many situations, we utilise a cost function instead, and aim to minimise its value.

In addition, P&M provide more granular discussion of the dimensions of complexity in the agent design space (see P&M Ch 1.5). Draw on both of these agent design frameworks when completing the exercises below.

A **state graph representation** is a way to represent a search problem concretely in a program. It is also a way of thinking about the problem, and sometimes we may or may not explicitly represent the state graph. Moreover, in problems with continuous or very large state space, a state graph is often used as a compact representation of the state space (more on this later in the course). Formally, a state graph $G = (V, E)$ comprises:

- **Vertices** (V) representing states, and
- **Edges** (E) representing world dynamics

Each edge $\overline{ss'} \in E$ is labelled with the cost to move from s to s'. It may also be labelled by the action to move from state s to s'. The initial and goal states are mapped to the initial and goal vertices of the graph. A solution is a path from initial to goal vertices in the state graph. A solution's *cost* is the sum of the cost associated with each edge in the path. Given this, the optimal solution is the shortest path through the state graph. In the next lectures, we will explore several different ways to **search** over a problem's state graph representation to find a good solution.

Exercises

Exercise 1.1. Design a **tic-tac-toe** or *noughts-and-crosses* playing agent, using the agent design components listed above. Assume that a single time step includes a single move by the agent and the immediate move by the opponent. The goal is to win with as few steps as possible.

O	X	O
X	O	
X		

Example tic-tac-toe board

Exercise 1.2. Determine the number of valid game states in tic-tac-toe.
If you wish, you may use ChatGPT or similar.

- Before running ChatGPT, think about how you would solve the task and draft the pseudocode
- Craft a suitable prompt
- Before executing the code, check that the code does what you expect and has interpreted the task correctly (if not, adjust your prompt or guide the response)
- Verify your answer (e.g. consider upper bounds on the problem, compare to other sources)

Now, compare your answer to the following output from ChatGPT (July 20, 2023 Version).

To determine the total number of valid states in the game of Tic-Tac-Toe, we need to consider all possible combinations of X (player 1), O (player 2), and empty spaces (blank cells) on the board. Since Tic-Tac-Toe is played on a 3x3 grid, there are 9 cells in total, each of which can be either X, O, or empty.

We can use a recursive function to generate all possible board configurations and then count the valid states. A valid state is a board configuration that follows the rules of Tic-Tac-Toe, where there is no winner yet and no more moves can be made. In other words, it's a board with no three consecutive X's or O's in a row, column, or diagonal.

Here's the Python code to calculate the total number of valid states:

```python
def is_winner(board, player):
    # Check if the player has won horizontally, vertically, or diagonally
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_board_full(board):
    return all(board[i][j] != '' for i in range(3) for j in range(3))

def count_valid_states(board, player):
    if is_winner(board, 'X') or is_winner(board, 'O') or is_board_full(board):
        return 1

    count = 0
    next_player = 'X' if player == 'O' else 'O'
    for i in range(3):
        for j in range(3):
            if board[i][j] == '':
                board[i][j] = player
                count += count_valid_states(board, next_player)
                board[i][j] = ''  # Reset the cell to its empty state after checking

    return count
```

```python
if __name__ == "__main__":
    # Initialize an empty 3x3 board
    board = [['' for _ in range(3)] for _ in range(3)]

    # Start counting valid states with the first player as 'X'
    total_valid_states = count_valid_states(board, 'X')
    print("Total number of valid states in Tic-Tac-Toe:", total_valid_states)
```

Keep in mind that calculating the total number of valid states for Tic-Tac-Toe can take some time since it involves exploring all possible combinations. The code above efficiently uses recursion to explore the game tree and determine the count of valid states.

Note that running the above code outputs the following:
`Total number of valid states in Tic-Tac-Toe:  255168`

Exercise 1.3. Consider the **components of an intelligent robot vacuum cleaner**.
Define its components:

- **abilities:**
- **goals:**
- **prior knowledge:**
- **stimuli/percepts:**
- **past experiences:**

Exercise 1.4. Percept space vs State space
How would the state space and percept space differ for a robot vacuum cleaner?

Exercise 1.5. Consider a **navigation app**, like an app on your smart phone or car that you use to find your way around UQ or other places. This program is essentially a rational agent. Assume that:

- Its goal is to find the shortest path to a goal location,
- The map used by the agent is 100% up to date,
- The location provided by the GPS is correct.

Suppose that you want to develop this navigation agent,

(a) How will you design it? Specify the design components of this problem, listed earlier.

(b) Specify the type of environment this agent operates in (i.e., discrete/continuous, deterministic/non-deterministic, fully/partially observable, static/dynamic)? Explain your selections, and think of the effect of each assumption above on the agent and its environment.

(c) Define the search problem and its corresponding state graph representation for this query.

Exercise 1.6. A **poker bot** is a program that automatically plays poker on the internet. Poker bots are software agents that typically use AI techniques to attempt to beat human poker players. Think about how to design a poker bot for the version of poker called *Texas hold 'em*, with rules:

- Every player is dealt two cards, for their eyes only.
- The dealer spreads five cards face up for all to see in three *stages*: (i) three at once, (ii) then another, (iii) then the last. All five face-up cards can be used by all players to make their best possible five-card hand.
- Before and after the card/s in each *stage* are revealed, players take turns to bet.
- The best poker hand wins the pot (all the the bets).

What complications arise when a poker bot tries to play against more than one other poker player?