Semester 2, 2019
COMP3702/7702 ARTIFICIAL INTELLIGENCE
ASSIGNMENT 1: Sokoban – Discrete Search
Due: 5pm, Thursday 22 August

Note:

- This assignment consists of two parts: Programming and Report.
- You can do this assignment in a group of **at most** 3 students. This means you can also do the assignment individually.
- For those who choose to work in a group:
  - Please register your group name at the following link before **8am on Monday, 12 August 2019**. If you have not registered your group by then, you will need to work on the assignment individually: https://docs.google.com/spreadsheets/d/1uHLPhOFiNNHX3gijRSm0nXVux5NgKwMkL4YbAcaqlqQ/edit?usp=sharing
  - All students in the group must be enrolled in the same course code, i.e., all COMP3702 students or all COMP7702 students.
  - All group members are expected to work in both programming and report. The demo will involve Q&A to each group member, individually.
- **Submission Instructions:**
  - Your program should either run directly from command prompt (python):

    ```
    > python ProgramName.py inputFileName outputFileName
    ```

    or must be compiled to generate an executable that can be run from command prompt as e.g.:

    ```
    > java ProgramName inputFileName outputFileName
    ```

  - You should submit **only the source code** required to compile the program, i.e., remove all object files and executables before submission.
  - The report should be in .pdf format and named a1-[courseCode]-[ID].pdf.
    If you work individually, ID is your student number.
    If you work in a group, ID is the student number of all group members separated by a dash. For instance, if you work in a group of two, and the student number is 12345 and 45678, then
    [ID] should be replaced with `12345-45678`
  - The report and all the source code necessary to compile your program should be placed inside a folder named a1-[courseCode]-[ID].
  - The folder should be zipped under the same name, i.e., a1-[courseCode]-[ID].zip, and the zip file should be submitted via Turnitin before **5pm on Thursday, 22 August 2019**.
- **Demo Instructions:**
  - Demos will be scheduled the week following the submission deadline.
  - If you work in a group, all group members must be present for the demo.
  - You must use the lab PC for the demo, and therefore you must make sure that your code compiles (if in Java or C/C++) and runs correctly on the lab PCs.

Background

Sokoban is a puzzle video game introduced in Japan in 1981. The word *Sokoban* means "warehouse keeper" in Japanese which refers to the main character of the game. In this game, the player controls the warehouse keeper who is in a closed warehouse with a number of boxes that need to be put on some predefined target locations. The keeper can move around the warehouse and the goal of the game is to push all boxes onto the target positions in the fewest number of moves. An example state of the game is shown in Figure 1.

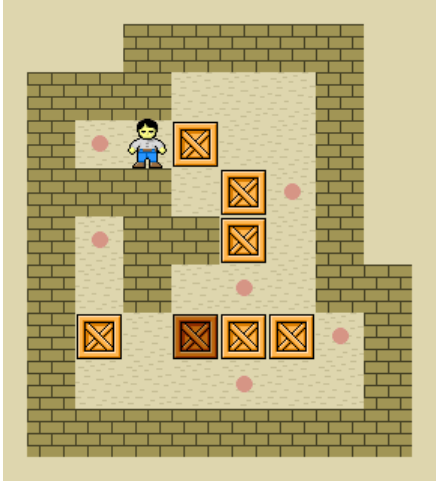

Figure 1. An example Sokoban game – the boxes need to be moved onto the pink dots

How to play

The warehouse is a grid where each square is a floor or wall tile. In each step, the keeper can move in any of the four basic directions (up, down, left, right). The keeper cannot walk into a wall or into a box. The keeper can however push a box if the square next to the box (in the pushing direction) is either a target or an empty square. Only one box can be pushed in any given step. If there are multiple targets, each box can go in any of the target positions and can be placed on targets in any order. A box can also be pushed off a target if needed. The game ends as soon as every box is in a target position.

Sokoban as a search problem

In this assignment, you will write components of a program to play Sokoban, with the objective to find a solution to the problem using various search algorithms. This assignment will test your experience in defining a search space for a practical problem and developing good heuristics to make your program more efficient.

To turn this game into a search problem, the following components need to be defined:

1. A state representation
2. A successor function that indicates which states can be reached from a given state
3. A goal-test
4. A cost function

Once these components are defined, the problem can be solved using search algorithms such as Breadth-First-Search, Depth-First-Search, A*, etc. The efficiency of the program depends on the type of search algorithm used. We assume that each step has a uniform cost of 1.

Game state representation

Each game state will be represented as a character array, with the row and column representing the (x,y) position on the board.

An example game state can be represented as below, where each grid space is represented by the mapping in Table 1.

```
#########
#####   #
#   # # #
# BT B P#
#### T###
#########
```

Table 1: Mapping of grid contents

Grid contents	Symbol
Free space	' '
Wall (obstacle)	'#'
Box	'B'
Target	'T'
Player	'P'
Box on top of a target	'b'
Player on top of a target	'p'

What is provided to you:
We will provide supporting code in Python only:

1. A parser to take an input file and convert it into a Sokoban map
2. A state visualiser
3. A tester

We will also supply test cases to test and evaluate your solution.

The support code can be found at:
https://gitlab.com/3702-2019/assignment-1-support-code

Some tips:

1. Write a **goal_test** function that takes a state as the input argument and determines if the given state is a goal state of the game. (see "how to play" section for a definition of the goal)
2. Write a **next_states** (or "get_successors") function to return a list of all states that can be reached from the given state in one move.

Your task

Your task it to develop a program that outputs a path (series of moves/actions) for the agent (i.e. the Sokoban) to push the boxes to the goal locations. Implement the Uniform Cost and A* search algorithms to output a path (series of moves/actions) for the agent (i.e. the Sokoban) to push the boxes to the goal locations.

Output format

- Your program should output a solution in the form of a comma separated list of directions: {u, d, l, r}, which indicates each of the possible directions (up, down, left, right)
- For example, a solution could be: r, r, d, l, d, l, u, u, u
- Your program should also record the performance statistics as outlined in question 4 of the report questions. These can either be output on the following line, with each of the 4 fields separated by commas, or alternatively, the statistics can be printed to stdout (i.e. standard out).

Note: Only your most efficient solution will be graded during the demo. You can manually set which search algorithm runs in your program, or if you wish you can have additional optional input arguments to set which algorithm is run.

You will be graded on your **programming** (60%) and a **report** (40%).

Grading for the Programming component (total points: 60/100)

For marking, we will use 12 different maps to evaluate your solution. COMP3702 students can get full marks by solving at least 9 of the 12 maps. However, COMP7702 students must solve all maps to get full marks. Solving a map means finding the optimal path within the given time limit. The time limit is numberOfBoxes*30 seconds.

For your information, our reference solution runs with the following times:

Test Case	UCS	A*
1box_m1	0.004s	0.003s
1box_m2	0.005s	0.004s
1box_m3	0.03s	0.01s
2box_m1	0.001s	0.001s
2box_m2	0.3s	0.3s
2box_m3	0.9s	0.9s
3box_m1	0.2s	0.06s
3box_m2	25s	7s
4box_m1	3.5s	1.2s
4box_m2	N/A	114s
4box_m3	95s	31s
6box_m2	54s	12.4s

The details of the grading scheme are as follows:

COMP3702:

- >= 1 & < 10: The program does not run (staff discretion).
- >= 10 & < 20: The program runs but fails to solve any test map within 1-2X the given time limit (staff discretion).
- 30: The program solves at least one of the test maps within 1-2X the given time limit.
- 40: The program solves all test maps involving 1 box.
- 50: The program solves all test maps involving 2 boxes.
- 60: The program solves at least 9 of the 12 test maps.

COMP7702:

- >= 1 & < 5: The program does not run (staff discretion).
- >= 5 & < 10: The program runs but fails to solve any test map within 1-2X the given time limit (staff discretion).
- 20: The program solves at least one of the test maps within 1-2X the given time limit.
- 30: The program solves all test maps involving 1 box.
- 40: The program solves all test maps involving 2 boxes.
- 50: The program solves at least 9 of the 12 test maps.
- 60: The program solves all test maps.

Grading for the Report component (total points: 40/100)

Please answer the following questions as part of the report:

1. [2.5 points] Define the agent design problem
2. [2.5 points] What type of agent is it? (i.e. discrete /continuous, fully / partially observable, deterministic / non-deterministic, static / dynamic)? Please explain your selection.
3. [10 points] What is the heuristic you use for A* search? Please explain why you think it is a good heuristic. Is your heuristic admissible?
4. [15 points] Compare the performance of Uniform Cost and A* search in terms the following statistics:
    a. The number of nodes generated
    b. The number of nodes on the fringe when the search terminates
    c. The number of nodes on the explored list (if there is one) when the search terminates
    d. The run time of the algorithm (e.g. in units such as mins:secs)
5. [10 points] A challenging aspect of designing a Sokoban puzzle solving agent are "deadlock" states where there is no possible path to a solution. Implement a function that detects the "deadlock" states. Your documentation should provide a thorough explanation of the criteria you used to find deadlocks.

Please format the report with each question under its own heading.