Computer Science 384
St. George Campus

Tuesday, January 26, 2016
University of Toronto

Homework Assignment #1: Search
Due: Tuesday, February 9, 2016 by 11:59 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e. no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 2 grace days.

Total Marks: This part of the assignment represents 10% of the course grade.

Handing in this Assignment

What to hand in on paper: Nothing.

What to hand in electronically: You must submit your assignment electronically. Download `rushhour.py` and `search.py` from the website. Modify `rushhour.py` so that it solves the Rush Hour problem as specified in this document. Then, submit your modified `rushhour.py` using MarkUs. Your login to MarkUs is your CDF username and password. You can submit a new version of any file at any time, though the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission.

We will test your code electronically. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using Python version 3.4.3), you will receive a failing grade on the assignment.

When your code is submitted, we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness; it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the tests in the provided testing script.

- *make certain that your code runs on CDF using python3 (version 3.4.3) using only standard imports.* This version is installed as "python3" on CDF. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 Clarification page:
`http://www.cdf.toronto.edu/~csc384h/winter/Assignments/A1/a1_faq.html`.
It is also linked to the A1 webpage. *You are responsible for monitoring the A1 Clarification page.*

Help Sessions: There will be two help sessions for this assignment: Wednesday, Feb. 3, 5-6 pm and Friday, Feb. 5, 12-1 pm. Both will be held in Pratt 378 (6 King's College Rd., 3rd Floor).

Questions: Questions about the assignment should be asked on Piazza:
`https://piazza.com/utoronto.ca/winter2016/csc384/home`.
If you have a question of a personal nature, please email the A1 TA, Andrew Perrault, at t1perrau at cdf dot utoronto dot ca or the instructor, placing 384 and A1 in the subject line of your message.

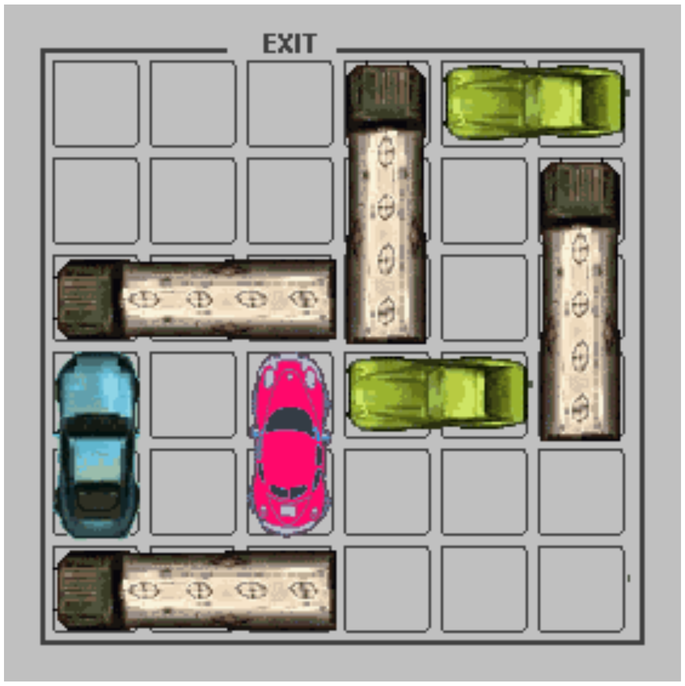

Figure 1: A state of the Rush Hour puzzle.

1 Introduction

The goal of this assignment will be to implement a working solver for the puzzle game Rush Hour shown in Figure 1. The puzzle involves moving a goal car, that may be blocked by additional cars and trucks, to an exit. Cars and trucks can only move in straight lines. The game can be played online here: `http://www.thinkfun.com/play-online/rush-hour/`. We recommend that you familiarize yourself with the rules and objective of the game before proceeding, although it is worth noting that the version that is presented online is only an example. We will give a formal description of the puzzle in the next section.

`search.py`, which is available from the website, provides a generic search engine framework and code to perform several different search routines. You will use it as a base for your Rush Hour solver. A brief description of the functionality of `search.py` follows. The code itself is heavily documented and worth reading.

- An object of class `StateSpace` represents a node in the state space of a generic search problem. The base class defines a fixed interface that is used by the `SearchEngine` class to perform search in that state space.

  For a specific problem, one defines a concrete sub-class that inherits from `StateSpace`. This concrete sub-class inherits some of the "utility" methods already implemented in the base class and overrides some other core methods to provide functionality specific to the problem being represented.

  Each `StateSpace` object *s* has the following key attributes:

  – *s.gval*: the *g* value of that node, i.e., the cost of getting to that state.

  – *s.parent*: the parent `StateSpace` object of *s*, i.e., the `StateSpace` object that has *s* as a successor. Will be *None* if *s* is the initial state.

- $s.action$: a string that contains that name of the action that was applied to $s.parent$ to generate s. Will be *"START"* if s is the initial state.

The three methods of `StateSpace` you must override (in `rushhour.py`) to implement the Rush Hour problem are:

- $s.successors()$: returns a list of the successor `StateSpace` objects of s that can be generated by applying the available actions to s.
- $s.hashable\_state()$: returns an immutable and unique (i.e., over the set of possible `StateSpace` objects) representation of s. This method is used by the search functions to check if s has been generated before, e.g., for cycle checking.
- $s.print\_state()$: returns a string representation of the state. This is used for viewing the paths the search engine takes.

- An object of class `SearchEngine` *se* runs the search procedure. A `SearchEngine` object is initialized with a search strategy (*'depth_first'*, *'breadth_first'*, *'best_first'* or *'a_star'*) and a cycle checking level (*'none'*, *'path'*, or *'full'*). Tracing can be turned on by setting $se.trace\_on(level)$ to 1 or 2. The main method of `SearchEngine` is $se.search(initial\_state, goal\_function, heuristic\_function)$, which runs a search starting at initial state until the goal is found or the search space is exhausted.

  - $initial\_state$ is an object of type `StateSpace`.
  - $goal\_function(s)$ is a function which returns `True` if `StateSpace` s is a goal and `False` otherwise.
  - $heuristic\_function(s)$ is a function that returns a heuristic value of `StateSpace` s.

  `SearchEngine` uses two auxiliary classes:

  - An object of class `sNode` *sn* represents a node in the search space. *sn* is simply a `StateSpace` object with an associated *hval*, i.e., the heuristic function value of that state.
  - An object of class `Open` is used to represent the search frontier. An `Open` object organizes the search frontier in the way that is appropriate for a given search strategy.

Figure 2: A state of the Water Jugs puzzle.

2 `StateSpace` Example: `WaterJugs.py`

`WaterJugs.py` contains an example implementation of the search engine for the Water Jugs problem shown in Figure 2.

- You have two containers that can be used to store water. One has a three-gallon capacity, and the other has a four-gallon capacity. Each has an initial, known, amount of water in it.
- You have the following actions available:
  - You can fill either container from the tap until it is full.
  - You can dump the water from either container.
  - You can pour the water from one container into the other, until either the source container is empty or the destination container is full.
- You are given a goal amount of water to have in each container. You are trying to achieve that goal in the minimum number of actions, assuming the actions have uniform cost.

`WaterJugs.py` has an implementation of the Water Jugs puzzle that is suitable for using with `search.py`. Note that in addition to implementing the three key methods of `StateSpace`, the author has created a set of tests that show how to operate the search engine. You should study these to see how the search engine works.

3 Description of the Rush Hour Puzzle

The Rush Hour puzzle has the following formal description. Note that our version differs from the standard one. Read the description carefully.

- The puzzle is played on a grid *board* with dimensions M rows by N columns.
- The board has V *vehicles* on it. Each vehicle v has several properties:
  - $v.loc$ is the vehicle's initial position $(x.y)$, counted from the upper left corner of the grid, e.g., $(0,0)$ would be the upper left corner and $(M-1,N-1)$ would be the bottom right corner.
  - $v.is\_horizontal$ indicates whether v is horizontal, i.e., parallel to the x-axis. If v is not horizontal, it is parallel to the y-axis.
  - $v.length$ is the length of v, i.e., the number of units that it extends in the positive direction of its orientation.
  - $v.is\_goal$ indicates whether v is the vehicle that you must move to the goal (i.e. the "goal vehicle").

  For example, the red vehicle in Figure 1 would have $v.loc = (2,3)$, $v.is\_horizontal =$ `False`, $v.length =$ 2, $v.is\_goal =$ `True` (traditionally, red indicates the vehicle to move to the exit in a Rush Hour puzzle).
- A *move* consists of moving any single vehicle one unit in either direction along its axis of movement: a horizontal vehicle can only move parallel to the x-axis and a vertical vehicle can only move parallel to the y-axis. A vehicle can only move into a space that is not occupied by another vehicle. *Note that in in our version of the puzzle, the board has no walls at the edges; a vehicle that drives off the board on one side will reappear on the other.* For example, in Figure 1, the green car in the upper right could move one tile to the right and end up occupying the tiles $(0,5)$ and $(0,0)$.
- The *goal* g is represented by a location $g.loc = (x,y)$ and an orientation $g.orientation \in \{'N','E','S','W'\}$. The objective of the puzzle is to move a goal vehicle gv to the goal location such that the goal vehicle is touching the goal entrance and in the correct orientation to enter the goal. For example, consider the board of Figure 1, disregarding the walls at the edges. The goal has location $(2,0)$ and orientation $'N'$. Having the goal vehicle at location $(2,0)$ would be the unique goal state. If the goal was at $(2,0)$ and oriented $'S'$, then $(2,5)$ would be the unique goal state. In this state, the goal vehicle would be wrapped around the board, and occupying both $(2,5)$ and $(2,0)$. *Note that in our version of the puzzle, a goal can be located anywhere on the board, not just at board boundaries.*

4 To Do

1. Implement a state representation for this problem and a successor state function. You will need to keep track of at least the following items in your state object:
   (a) The properties of each vehicle, including its location.
   (b) The board properties, including size, goal entrance and direction.

2. Implement the successor state function for class `rushhour`. In particular, your successor state function should implement the action `move_vehicle(<vehicle_name>, <direction>)`, where direction is one of $\{'N','E','S','W'\}$. States produced by your function will have to record the actions that preceded them; i.e. they should incorporate action names of the form `move_vehicle(<vehicle_name>, <direction>)`.

3. Implement a `make_init_state` function that creates an initial state object from a puzzle specification. Details are in `rushhour.py`.

4. Implement a set of data access methods for your `rushhour` state objects. Details are in `rushhour.py`.

5. Implement a heuristic for this domain so that you can use A* search. Details are in `rushhour.py`.

GOOD LUCK!