Computer Science 384
St. George Campus

Monday, May 25, 2020
University of Toronto

Homework Assignment #1: Search
Due: Tuesday, June 9, 2020 by 10:00 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 3 grace days.

Total Marks: This part of the assignment represents 13% 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 the assignment files from http://www.teach.cs.toronto.edu/~csc384h/summer/Assignments/A1/. Modify `solution.py` and `tips.txt` as described in this document and submit your modified versions using MarkUs. Your login to MarkUs is your teach.cs 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. More detailed instructions for using Markus are available at:
http://www.teach.cs.toronto.edu/~csc384h/summer/markus.html.

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 3.7), 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 TEACH.CS using python3 (version 3.7) using only standard imports.* This version is installed as "python3" on TEACH.CS. 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. See Section 3 for a description of the starter code. For the purposes of this assignment, we consider the standard imports to be what is included with Python 3.7 plus NumPy 1.11.3 and SciPy 0.16.1 (which are installed on the teach.cs machines). If there is another package you would like to us, please ask and we will consider adding it to the list.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 Clarification page:
`http://www.teach.cs.toronto.edu/~csc384h/summer/Assignments/A1/a1_faq.html`.

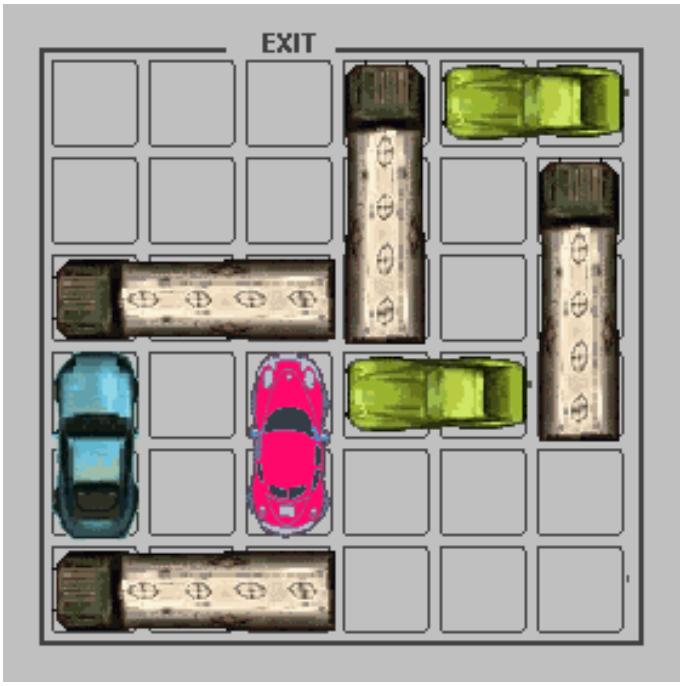

Figure 1: A state of the Rush Hour puzzle.

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 and a code walk through for this assignment; dates will be announced on the A1 web page.

Questions: Questions about the assignment should be asked on Piazza:

`https://piazza.com/utoronto.ca/summer2020/csc384/home`.

If you have a question of a personal nature, please email the A1 TA, Sam Motamed, at:

`sam.motamed@mail.utoronto.ca`

or the instructor, placing [CSC384] and A1 in the subject line of your message. Assignment solution specific questions will not be answered via email.

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 at: `https://www.crazygames.com/game/rush-hour-online`. 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.

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 N x-dimension and M y-dimension.
- 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 $(N-1,M-1)$ 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. $v.length$ must be positive.
  - $v.is\_goal$ indicates whether moving v to the goal completes the puzzle.

  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 $(5,0)$ 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.*

2 Starter Files

You have been provided with the following files:

1. `search.py`
2. `rushhour.py`
3. `rushhour_tests.pkl`
4. `autograder.py`
5. `solution.py`
6. `tips.txt`

The only files you will submit are `solution.py` and `tips.txt`. We consider the other files to be starter code, and we will test your code using the original versions of those files. In order for your `solution.py` to be compatible with our starter code, you should not modify the starter code. In addition, you should not modify the functions defined in the starter code files from within `solution.py`

The file `search.py`, which is available from the website, provides a generic search engine framework and code to perform several different search routines. This code will serve as a base for your Rush Hour Puzzle solver. A brief description of the functionality of `search.py` follows. The code itself is 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 the Rush Hour Puzzle problem, we will define a concrete sub-class that inherits from `StateSpace`. This concrete sub-class will inherit some of the "utility" methods that are implemented in the base class.

  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.

- 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'*, *'a_star'* or *'custom'*) and a cycle checking level (*'none'*, *'path'*, or *'full'*).

  Note that `SearchEngine` depends on two auxiliary classes:

  - An object of class `sNode` *sn* represents a node in the search space. Each object *sn* contains a `StateSpace` object and additional details: *hval*, i.e., the heuristic function value of that state and *gval*, i.e. the cost to arrive at that node from the initial state. An $fval\_fn$ and *weight* are also tied to search nodes during the execution of a search, where applicable.
  - 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.

When a `SearchEngine` has a search strategy that is is set to 'custom', you will have to specify the way that f-values of nodes are calculated; these values will structure the order of the nodes that are expanded during your search.

Once a `SearchEngine` object has been instantiated, you can set up a specific search with:

$init\_search(initial\_state, goal\_fn, heur\_fn, fval\_fn)$

and execute that search with

$search(timebound, costbound)$

The arguments are as follows:

- *initial_state* will be an object of type `StateSpace`; it is your start state.
- $goal\_fn(s)$ is a function which returns *True* if a given state s is a goal state and *False* otherwise.
- $heur\_fn(s)$ is a function that returns a heuristic value for state s. This function will only be used if your search engine has been instantiated to be a heuristic search (e.g., $best\_first$).
- *timebound* is a bound on the amount of time your code will execute the search. Once the run time exceeds the time bound, the search will stop; if no solution has been found, the search will return *False*.
- $fval\_fn(sNode)$ defines f-values for states. This function will only be used by your search engine if it has been instantiated to execute a 'custom' search. Note that this function takes in an *sNode* and that an *sNode* contains not only a state but additional measures of the state (e.g., a *gval*). The function will use the variables that are provided in order to arrive at an f-value calculation for the state contained in the *sNode*.
- *costbound* is an optional bound on the cost of each state s that is explored. *costbound* should be a 3-tuple $(g\_bound, h\_bound, g\_plus\_h\_bound)$. If a node's $g\_val > g\_bound$, $h\_val > h\_bound$, or $g\_val + h\_val > g\_plus\_h\_bound$, that node will not be expanded. You can use *costbound* to implement pruning in the anytime searches described below.

For this assignment we have also provided `rushhour.py`, which specializes StateSpace for the Rush Hour problem. You will therefore not need to encode representations of Rush Hour Puzzle states or the successor function for Rush Hour Puzzle states! These have been provided to you so that you can focus on implementing good search heuristics and anytime algorithms.

The file `rushhour.py` contains:

- An object of class `Rushhour`, which is a `StateSpace` with these additional key attributes:
  - $s.board\_properties$:
  - $s.vehicle\_list$:
- `Rushhour` also contains the following key functions:
  - $successors()$: This function generates a list of `Rushhour` states that are successors to a given `Rushhour` state. Each state will be annotated by the action that was used to arrive at the `Rushhour` state.
  - $hashable\_state()$: This is a function that calculates a unique index to represents a particular `Rushhour` state. It is used to facilitate path and cycle checking.
  - $print\_state()$: This function prints a `Rushhour` state to stdout.

  Note that `Rushhour` states depend on one auxiliary class:
  - An object of class Vehicle ...

The file `rushhour_tests.pkl` contains a set of initial states for Rush Hour Puzzle problems. The file `autograder.py` will load these states and use them in the context of a series of tests. You can use these files to test your implementations.

The file `solution.py` contains the methods that need to be implemented and the file `tips.txt` will contain a description of your custom search heuristic (see below).

3 Assignment Specifics – Your Tasks

To complete this assignment you must modify `solution.py` and `rushhour.py`.

In `solution.py`, you will:

- **(Worth 15 points)** Implement a goal test for the rush hour puzzle ($rushhour\_goal\_fn(state)$). Your function must return $True$ if the goal vehicle gv is touching the goal entrance and in the correct orientation to enter the goal. The function must return $False$ otherwise.
- **(Worth 15 points)** Implement a "min moves" distance heuristic ($heur\_min\_moves(state)$). Since the board wraps around, there are two different directions that can lead to the goal.
  - If we move in one direction, let MOVES1 = the number of moves required to get to the goal if the goal were unobstructed.
  - If we move in the other direction, let MOVES2 = number of moves required to get to the goal if it were unobstructed.

  The "min moves" distance heuristic value will be the minimum of MOVES1 and MOVES2 over all goal vehicles. You should implement this heuristic function exactly even if you think modifications would improve the heuristic.
- **(Worth 15 points)** implement a heuristic for the Rush Hour Puzzle that attempts to improve on the "min moves" distance heuristic ($heur\_alternate(state)$)
- **(Worth 20 points)** Implement Anytime Greedy Best-First Search ($anytime\_gbfs(initial\_state, heur\_fn, timebound)$). Details regarding this algorithm are provided in the next section.
- **(Worth 20 points)** Implement Anytime Weighted A* ($anytime\_weighted\_astar(initial\_state, heur\_fn, weight, timebound)$). Details regarding this algorithm are provided in the next section. Note that your implementation will require you to instantiate a `SearchEngine` object with a custom search strategy. To do this you must create an f-value function ($fval\_function(sNode, weight)$) and remember to provide this when you execute $init\_search$. Tests of the $fval\_function(sNode, weight)$ will account for 10 of the 20 points for this problem.
- **(Worth 15 points)** In your `tips.txt` file:
  1. You may notice that results from your anytime weighted a-star algorithm don't significantly improve with time. In less than two sentences, explain why this might be the case.
  2. Give a scenario where the min-moves heuristic falls short (in less than two sentences). Is this heuristic admissible?
  3. Explain a key improvement in your alternative heuristic (in less than two sentences). Is your revised heuristic admissible?

Note that when we are testing your code, we will limit the time for each run of your algorithm on teach.cs.

4 Anytime Greedy Best-First Search

Greedy best-first search expands nodes with lowest $h(node)$ first. The solution found by this algorithm may not be optimal. Anytime greedy-best first search (which is called *anytime_gbfs* in the code) continues searching after a solution is found in order to improve solution quality. Since we have found a path to the goal after the first iteration, we can introduce a cost bound for pruning: if node has $g(node)$ greater than the best path to the goal found so far, we can prune it. The algorithm returns either when we have expanded all non-pruned nodes, in which case the best solution found by the algorithm is the optimal solution, or when it runs out of time. We prune based on the *g_value* of the node only because greedy best-first search is not necessarily run with an admissible heuristic.

Record the time when *anytime_gbfs* is called with $os.times()[0]$. Each time you call search, you should update the time bound with the remaining allowed time. The automarking script will confirm that your algorithm obeys the specified time bound.

5 Anytime Weighted A*

Instead of A*'s regular node-valuation formula $f(node) = g(node) + h(node)$, Weighted A* (which is called *anytime_weighted_astar* in the code) introduces a weighted formula:

$$f(node) = g(node) + w \times h(node)$$

where $g(node)$ is the cost of the path to node, $h(node)$ the estimated cost of getting from node to the goal, and $w \geq 1$ is a bias towards states that are closer to the goal. Theoretically, the smaller w is, the better the first solution found will be (i.e., the closer to the optimal solution it will be ... why??). However, different values of w will require different computation times.

Since the solution that is found by Weighted A* may not be optimal when $w > 1$, we can keep searching after we have found a solution. Anytime Weighted A* continues to search until either there are no nodes left to expand (and our best solution is the optimal one) or it runs out of time. Since we have found a path to the goal after the first search iteration, we can introduce a cost bound for pruning: if node has a $g(node) + h(node)$ value greater than the best path to the goal found so far, we can prune it.

When you are passing in a *f_val* function to *init_search* for this problem, you will need to have specified the *weight* for the *f_val* function. You can do this by wrapping the $fval\_function(sN, weight)$ you have written in an anonymous function, i.e.,

$$wrapped\_fval\_function = (lambdasN : fval\_function(sN, weight))$$

6 `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:

Figure 2: A state of the Water Jugs puzzle.

  - 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.

GOOD LUCK!