Computer Science 384
St. George Campus

Monday, May 20, 2019
University of Toronto

Homework Assignment #1: Search
Due: Tuesday, June 4, 2019 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 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 `solution.py`, `lunarlockout.py` and `search.py` from `http://www.teach.cs.toronto.edu/~csc384h/summer/Assignments/A1/a1_faq.html`. Modify `solution.py` so that it solves the Lunar Lockout problem as specified in this document. Then, submit your modified `solution.py` along with Astar.txt (answering the questions specified in section 4) 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.

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.

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`.
You are responsible for monitoring the A1 Clarification page.

Help Sessions: There will be two help sessions for this assignment. Dates and times for these sessions will be posted to the course website and to Piazza ASAP.

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

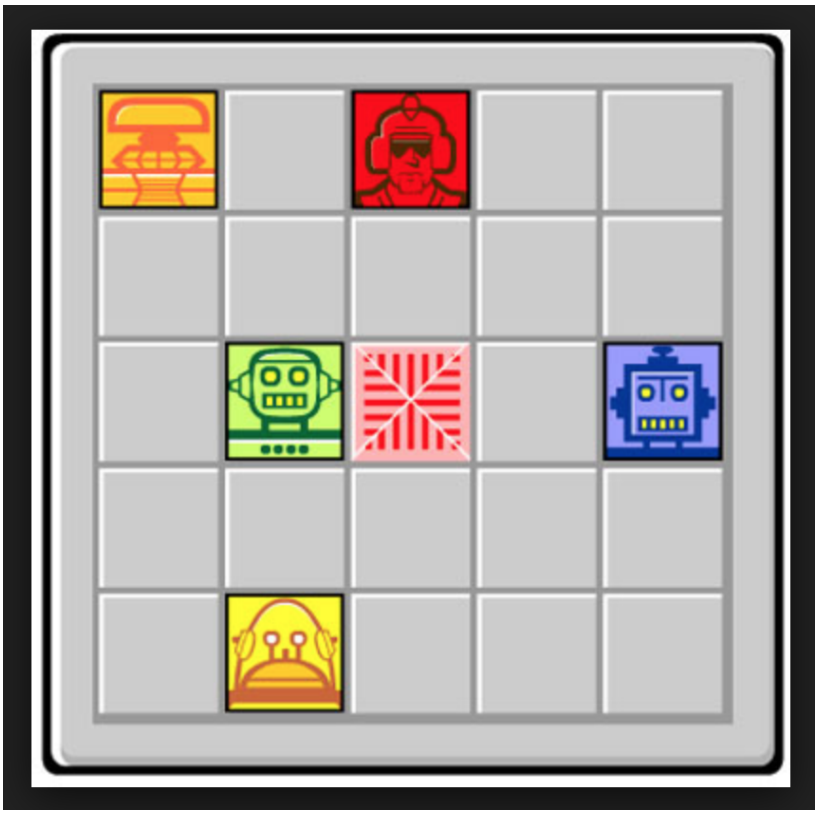

Figure 1: A state of the Lunar Lockout puzzle. The red piece is a 'rover' (or 'xanadu'), and the other pieces are 'helper robots'. The goal is to guide the 'rover' pieces into the escape hatch at the center of the board.

`https://piazza.com/utoronto.ca/summer2019/csc384/home`.
If you have a question of a personal nature, please email the A1 TA, Randy, at rhickey@cs.toronto.edu or a course instructor. Make sure to place [CSC384] and A1 in the subject line of your message.

1 Introduction

The goal of this assignment will be to implement a working solver for the puzzle game Lunar Lockout shown in Figure 1. Lunar Lockout is a puzzle game that is played on an *NxN* board. The game requires 'helper robots' to guide 'rovers' (also called 'xanadus') into an escape hatch that is located at the center of the board. The rules hold that pieces (i.e. helper bots and rovers) must move one at a time in a straight line; they may not move diagonally. Pieces also cannot move through one another. Moreover, each piece may only move in the direction of a second piece, and it must move until it collides with the second piece and comes to a stop. Moves may sometimes take pieces directly across the escape hatch or result in a robot blocking the escape hatch (i.e. occupying the same square as the escape hatch). A rover cannot exit the escape hatch unless in lands directly atop it.
The game is over when all rovers have successfully made it through the escape hatch.

You can watch a video of Lunar Lockout game-play on YouTube, at
`https://www.youtube.com/watch?v=2BxPr55buhM`. Our version of Lunar Lockout is slightly more complicated, however, as there may be more than one rover that needs to be guided to the escape hatch,

there can be an arbitrary number of helper robots, and the size of the board may be larger than 5*x*5.

2 Description of Lunar Lockout

Lunar Lockout has the following formal description. Read the description carefully.

- The puzzle is played on a square board that is a grid *board* with N squares in the x-dimension and N squares in the y-dimension. The dimension N is always odd.
- Each state in the game contains the x and y coordinates for each robot as well as the x and y coordinates for each rover (or xanadu).
- From each state, each robot and each rover can move North, South, East, or West, but *only* if there is a second robot or rover that lies in that direction. When a robot or rover moves, it must move all the way to the second piece until the pieces collide. For example, if a robot located at position $(4,0)$ moves South toward a rover located at $(4,3)$, the robot will end at the location $(4,2)$. No two pieces (robots or rovers) can move simultaneously or diagonally and pieces cannot pass through walls or one another.
- The escape hatch is always located in the center of the board (i.e. at the grid location $((N-1)/2,(N-1)/2)$.
- Once a rover arrives at the escape hatch, it exits the board through the escape hatch. It then disappears from subsequent play.
- Each movement is of equal cost.
- The goal is achieved when all rovers have exited the board via the escape hatch.

Ideally, we will want our rovers to exit the escape hatch before they deplete our internal oxygen supplies. This means that with each problem instance, you will be given a computation time constraint. You must attempt to provide some legal solution to the problem (i.e. a plan) within this time constraint. Better plans will be plans that are shorter, i.e. that require fewer operations to complete.

Your goal is to implement an anytime algorithm for this problem: one that generates better solutions (i.e. shorter plans) the more computation time it is given.

3 Code You Have Been Provided

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 Lunar Lockout 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 Lunar Lockout 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. This 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* which 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 tied to search nodes during the execution of a search, where applicable.
  - An object of class `Open` is used to represent the search frontier. The search frontier will be organized in the way that is appropriate for a given search strategy.

  When a `SearchEngine`'s search strategy 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.
  - $heuristic\_fn(s)$ 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*).
  - $fval\_fn(sNode, weight)$ 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 also takes in a float *weight*. It will use the variables that are provided to arrive at an f value calculation for the state contained in the *sNode*.
  - $timebound$ is a bound on the amount of time your code will be allowed to execute the search. Once the run time exceeds the time bound, the search must stop; if no solution has been found, the search will return *False*.

- *costbound* is an optional parameter that is used to set boundaries on the cost of nodes that are explored. This *costbound* is defined as a list of three values. $costbound[0]$ is used to prune states based on their g-values; any state with a g-value higher than $costbound[0]$ will not be expanded. $costbound[1]$ is used to prune states based on their h-values; any state with an h-value higher than $costbound[1]$ will not be expanded. Finally, $costbound[2]$ is used to prune states based on their f-values; any state with an f-value higher than $costbound[2]$ will not be expanded.

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

The file lunarlockout.py contains:

- An object of class `LunarLockoutState`, which is a `StateSpace` with these additional key attributes:
  - $s.width$: the width (and height) of the Lunar Lockout board
  - $s.robots$: positions for each robot that is on the board. Each robot position is a tuple (x,y), that denotes the robot's x and y position. $s.robots$ is therefore a tuple of tuples.
  - $s.xanadus$: positions for each rover that is on the board. Each rover position is also an (x,y) tuple. Like $s.robots$, $s.xanadus$ is a tuple of tuples.
- `LunarLockoutState` also contains the following key functions:
  - $successors()$: This function generates a list of LunarLockoutStates that are successors to a given LunarLockoutState. Each state will be annotated by the action that was used to arrive at the LunarLockoutState. These actions are (r,d) tuples wherein r denotes the index of the robot d denotes the direction of movement of the robot.
  - $hashable\_state()$: This is a function that calculates a unique index to represents a particular LunarLockoutState. It is used to facilitate path and cycle checking.
  - $print\_state()$: This function prints a LunarLockoutState to stdout.

  Note that LunarLockoutState depends on one auxiliary class:
  - An object of class `Direction`, which is used to define the directions that each robot can move and the effect of this movement.

Also note that lunarlockout.py contains a set of 20 initial states for Lunar Lockout problems, which are stored in the tuple *PROBLEMS*. You can use these states to test your implementations.

The file `solution.py` contains the methods that need to be implemented.

The file `autograder.py` runs some tests on your code to give you an indication of how well your methods perform.

4 Assignment Specifics

To complete this assignment you must modify `solution.py` to:

- Implement an L distance heuristic (*heur_L_distance*(*state*)). This heuristic will be used to estimate how many moves a current state is from a goal state. To calculate the L distance of a xanadu, assume it can move any number of squares horizontally as a first move followed by any number of squares vertically as a second move. Your implementation should calculate the sum of the L distances between each xanadu and the escape hatch. Ignore the positions of any intervening pieces in your calculations (i.e., for the purposes of this calculation, the xanadu does not need a piece to slide into and does not stop sliding when it hits another piece).

- Implement a non-trivial heuristic for Lunar Lockout that improves on the L distance heuristic (*heur_alternate*(*sta*
  Explain your heuristic in your comments in under 250 words.

- Implement Anytime Weighted A* (*weighted_astar*(*initail_state*, *timebound*)). Details about this algorithm are provided in the next section.

  Note that your implementation will require you to instantiate a `SearchEngine` object with a search strategy that is '*custom*'. You must therefore create an f-value function ($fval\_fn(sNode, weight)$) and remember to provide this, and the weight that your function requires, when you execute *search*.

- Briefly type an answer to the following questions and submit it in a text file named Astar.txt.

  1. Suppose you want to use A* and you have a heuristic that may overestimate or underestimate cost to the goal by as much as 20%. Can you do anything to guarantee the algorithm will find the optimal solution (if one exists)?
  2. (True or False) Assume you you are playing Lunar Lockout on a board that has three xanadus. The sum of the Manhattan distances between each xanadu and the exit is an admissible heuristic for this problem, should you use A-star to solve it.

Note that when we are testing your code, we will limit each run of your algorithm on teach.cs to 2 seconds. Instances that are not solved within this limit will provide an interesting evaluation metric: failure rate.

5 Anytime Weighted A*

Instead of A*'s regular node-valuation formula ($f = g(node) + h(node)$), Weighted-A* introduces a weighted formula:

$$f = g(node) + w * h(node)$$

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 real number. Theoretically, the smaller w is, the better the solution will be (i.e. the closer to the optimal solution it will be ... *why??*). However, different values of w will require different computation times.

An anytime version of this algorithm begins with a fast-running weight for w, and iteratively reduces the weight each time a solution is found. When time is up, the best solution (so far) is returned.

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

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

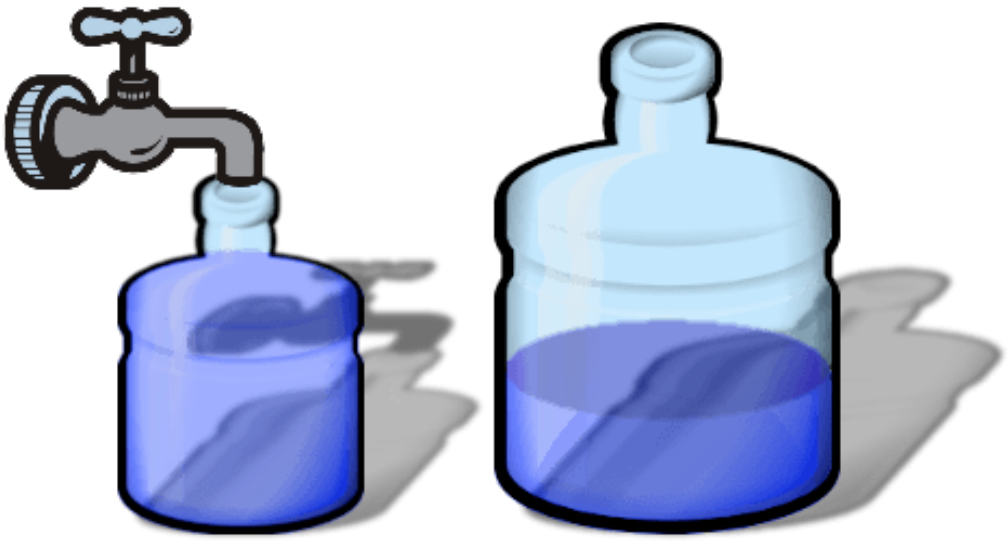

Figure 2: A state of the Water Jugs puzzle.

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:
  - 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!