Computer Science 384 Monday, June 12, 2017
St. George Campus University of Toronto

Homework Assignment #1: Search
Due: Monday, June 12, 2017 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 3 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 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 3.5.2), 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 3.5.2) 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 3.5.2 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.

Evaluation Details: The details of the evaluation will be released as a separate document in approximately one week.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 Clarification page:

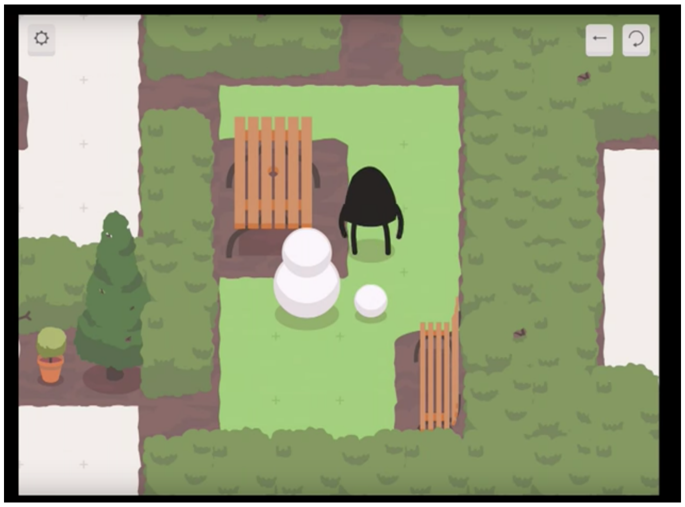

Figure 1: A state of the Snowman Puzzle.

`http://www.teach.cs.toronto.edu/~csc384h/summer/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; dates for these are TBD and will be announced on the A1 web page.

Questions: Questions about the assignment should be asked on Piazza:
`https://piazza.com/utoronto.ca/summer2017/csc384/home`.
If you have a question of a personal nature, please email the A1 TA, Joe Wu, at joe.wu.ca at gmail.com or the instructor, placing [CSC384] and A1 in the subject line of your message.

1 Introduction

The goal of this assignment will be to program a robot to successfully build a snowman in a given spot using snowballs that are located in an obstacle course. The rules hold that only one snowball can be moved by the robot at a time, that snowballs can only be pushed by the robot and not pulled, and that neither the robot nor the snowballs can pass through obstacles (i.e. walls or other snowballs). While snowballs cannot pass through obstacles, smaller snowballs can be stacked atop larger snowballs. Small snowballs can be stacked on large and medium snowballs; medium snowballs can be stacked on large snowballs. In addition, a robot cannot push more than one snowball at a time, i.e., if there are two snowballs in a row, the robot cannot push the the both of them simultaneously. The robot also cannot push a stack of snowballs; they must be pushed one at a time.

The game is over when a snowman exists on the game board in a pre-defined goal spot. A snowman is a stack of three snowballs: a large snowball on the bottom, a medium sized snowball in the middle and a small snowball at the top.

```
#######  #######  #######  #######
##   ##  ##   ##  ##   ##  ##   ##
#   m #  #   m #  # s m #  # s m #
# sbX #  # sbX #  # ?bX #  #  ?b #
#? ####  # ?####  #  ####  #  ####
#######  #######  #######  #######
   1        2        3        4
```

Figure 2: ASCII visualizer of a robot moving about a maze, and moving snowballs. This visualizer is provided with starter code for A1. Note that 'b' represents a big snowball, 'm' a medium one, and 's' a small one. The robot is a '?' and the desired destination for the snowman is marked with an 'X'.

Note that this game is a variant of a classic puzzle called Sokoban. Sokoban can be played online at https://www.sokobanonline.com/play. The variation we are asking you to encode is different as there are snowballs to push instead of boxes, and snowballs can be stacked. It is therefore related, but not identical, to a game called "a good snowman is hard to build"; a video walkthrough of this game can be found at https://www.youtube.com/watch?v=1HozNkueh4U. We will give a formal description of our version of the puzzle in the next section.

2 Description of the Snowman Puzzle

The Snowman Puzzle has the following formal description. Read the description carefully.

- The puzzle is played on a board that is a grid board with N squares in the x-dimension and M squares in the y-dimension.
- Each state contains x and y coordinates for the robot, the snowballs, the destination point for the snowman, and the obstacles.
- Each board initially will contains three snowballs: a small, medium and large snowball.
- From each state, the robot can move Up, Down, Left, or Right. If a robot moves to the location of an unobstructed snowball, the snowball will move one square in the same direction. Snowballs and the robot cannot pass through walls or obstacles, however.
- The robot cannot push more than one snowball at a time. If two snowballs are in succession and the robot is adjacent to the smaller of the two, the robot may push the smaller snowball atop the larger one. However, the robot cannot push a large snowball atop a smaller one, nor can the robot move a stack of snowballs. Movements that cause a snowball to travel more than.one unit of the grid are also illegal.
- Each movement is of equal cost. Whether or not the robot is pushing an snowball does not change the cost.
- The goal is achieved when there is a stack of three snowballs on the game board and in the destination spot. This stack must have a large snowball on the bottom, a medium sized snowball in the middle, and a small snowball at the top.

Ideally, we will want the robot to complete the snowman before the temperature rises and the snowman starts to melt. 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 to build the snowman) within this constraint. Better plans will be plans that are shorter, i.e. that require fewer operators 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

You have been provided with the following files:

1. `search.py`
2. `snowman.py`
3. `test_script.py`
4. `test_problems.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 Snowman Puzzle solver. You may also use this code later in the course for your project assignment if you like. 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 Snowman 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 nodes $g\_val > g\_bound$, $h\_val > h\_bound$, or $g\_val + h\_val > g\_plus\_h\_bound$, that node will not be expanded. You will use *costbound* to implement pruning in both of the anytime searches described below.

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

The file `snowman.py` contains:

- An object of class `SnowmanState`, which is a `StateSpace` with these additional key attributes:
  - *s.width*: the width of the Snowman Puzzle board
  - *s.height*: the height of the Snowman Puzzle board
  - *s.robot*: position for the robot: a tuple (x, y), that denotes the robots x and y position.
  - *s.snowballs*: positions for each snowball (or stack of snowballs) as keys of a dictionary. Each position is an (x, y) tuple. The value of each key is the index for that snowballs size (see below). Some values denote stacks of snowballs at a given location as well.
  - *s.obstacles*: locations of all of the obstacles (i.e. walls) on the board. Obstacles, like robots and snowballs, are also tuples of (x, y) coordinates.
  - *s.destination*: the target destination for the snowman: a tuple (x, y), that denotes the desired position for the completed snowman.
  - *s.sizes*: contains key, value pairs that indicate snowball sizes or the presence of a snowball stack. The possible values are: 'b' for a big snowball, 'm' for a medium snowball and 's' for a small one. A 'G' denotes a completed snowman. In addition, note that there are values to indicate stacks of snowballs on the board: 'A' represents a medium snowball atop big one, 'B' represents a small snowball atop big one and 'C' represents a small snowball atop medium one. See Figure 2 for snowballs as they are represented by the ASCII visualizer you have been provided.
- `SnowmanState` also contains the following key functions:
  - *successors*(): This function generates a list of `SnowmanStates` that are successors to a given `SnowmanState`. Each state will be annotated by the action that was used to arrive at the `SnowmanState` *up*, *down*, *left*, *right*.
  - *hashable_state*(): This is a function that calculates a unique index to represents a particular `SnowmanState`. It is used to facilitate path and cycle checking.
  - *print_state*(): This function prints a `SnowmanState` to stdout.

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

Also note that `test_problems.py` contains a set of 20 initial states for Snowball Puzzle problems, which are stored in the tuple PROBLEMS. You can use these states to test your implementations. Additional testing instances will be provided with the evaluation details.

`snowman.py` comes with an ASCII visualizer for Snowball Puzzle problems (see Figure 2).

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

`tips.txt` will contain a description of your original heuristic (see below).

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

4 Assignment Specifics Your Tasks

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

- Implement a Manhattan distance heuristic ($heur\_manhattan\_distance(state)$). This heuristic will be used to estimate how many moves a current state is from a goal state. The Manhattan distance between coordinates $(x0, y0)$ and $(x1, y1)$ is $|x0 - x1| + |y0 - y1|$. Your implementation should calculate the sum of Manhattan distances between each snowball (or stack of snowballs) and the target destination. Ignore the positions of obstacles in your calculations.
- Implement Anytime Greedy Best-First Search ($anytime\_gbfs(initial\_state, heur\_fn, timebound)$). Details regarding this algorithm are provided in the next section.
- 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 therefore an f-value function ($fval\_function(sNode, weight)$) and remember to provide this when you execute *init_search*.
- Implement a non-trivial heuristic for the Snowman Puzzle that improves on the Manhattan distance heuristic ($heur\_alternate(state)$). We will provide a separate evaluation document that specifies the performance we expect from your heuristic.
- You should give five tips (2 sentences each) as if you were advising someone who was attempting this problem for this first time on what to do. Write these tips in `tips.txt`. Note that when we are testing your code, we will limit each run of your algorithm on teach.cs to 5 seconds. Instances that are not solved within this limit will provide an interesting evaluation metric: failure rate.

5 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.

6 Anytime Weighted A*

Instead of A*s regular node-valuation formula $f(node) = g(node) + h(node)$, Weighted A* 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.,

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

GOOD LUCK!