Programming Assignment 1: The Search for a Snowperson

Due Feb 11 by 10pm **Points** 100

Released: January 25, 2021

Table of Contents.

Fig 1. An image of a robot building a snowperson in an obstacle course.

Warning (Please read this)

We are aware that solutions to this problem, or related ones, may exist on the internet. **Do not use these solutions as this would be plagiarism.** To earn marks on this assignment you must develop your own solutions. Also please consider the following points.

- **Do not add any non-standard imports in the python files you submit** (all imports already in the starter code must remain). All imports that are available on teach.cs are considered to be standard.
- **Do not change any of the supplied files except for** `solution.py`
- **Make certain that your code runs on teach.cs using python3**. You should all have an account on teach.cs and you can log in, download all of your code (including all of the supplied code) to a subdirectory of your home directory, and use the command `python3 autograder.py` and test it

there before you submit. Your code will be graded by running it on teach.cs, so the fact that it runs on your own system but not on teach.cs is not a legitimate reason for a regrade.
- The test cases used in the autograder reflect the test cases we will use during marking but are not identical. Tests performed by the autograder will help you gauge the final grade you can expect on the assignment. We will also look for certain things in the assignments (e.g., running them through code plagiarism checkers, looking at assignments that fail all tests, etc.). If we have **good reasons** we will change your grade from that given by the autograder either up or down.

Introduction

The goal of this assignment will be to program a robot to successfully build a snowperson in a specific spot using snowballs on 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, **small** snowballs can be stacked on **larger** snowballs. More specifically, **small** snowballs can be stacked on **large** or **medium** snowballs and **medium** snowballs can be stacked on **large** ones. 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 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 snowperson exists on the game board at the designated goal location. A snowperson is a stack of three snowballs: a **large** snowball on the bottom, a **medium** snowball in the middle, and a **small** snowball at the top.

You may note that this game is a variant of a classic puzzle called Sokoban. Sokoban can be played online at **this link (https://www.sokobanonline.com/play)**. The variation we are asking you to encode is slightly different as it involves snowballs instead of boxes, and snowballs can be stacked. A related game is called a good snowperson is hard to build; a video walkthrough of this game can be found at **this link (https://www.youtube.com/watch?v=1HozNkueh4U)**.

Formal Description

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

- The puzzle is played on a grid board with N squares in the x-dimension and M squares in the y-dimension.
- Each state contains the x and y coordinates for the robot, the snowballs, the destination point for the snowperson, and the obstacles.

- Each board initially 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 more than one snowball at a time. 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 a 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 snowball in the middle, and a small snowball at the top.

Ideally, we will want the robot to complete the snowperson before the temperature rises and the snow 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 snowperson) 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.

Starter Code

The code for this assignment consists of several Python files, some of which you will need to read and understand in order to complete the assignment and some of which you can ignore. You can download all the code and supporting files as a **Zipped file archive**. In that archive, you will find the following files.

Files you'll edit and submit on Markus:

`solution.py` Where all of your heuristics and anytime algorithms will reside.

`tips.txt` Where your heuristic engineering 'tips' will reside.

Files you might want to look at (look but don't modify):

`search.py` This contains default implementations of search algorithms discussed in class

`search.py` This contains default implementations of search algorithms discussed in class.

`snowman.py` This specifies the search to the snowperson domain, specifically.

`test_problems.py` This contains some example problems.

`autograder.py` This is an autograder for you to check your solutions as you develop them. The autograder can be run with the command:

```
python3 autograder.py
```

Note: the autograder and snowperson environment use python3. Note that on teach.cs the default is python2.7, so you must use the prefix python3.

See the autograder tutorial in Assignment 0 for more information about using the autograder.

Details of Starter Code

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 Snowperson Puzzle solver. A brief description of the functionality of `search.py` follows. The code itself is documented and worth reading. The file `search.py` contains:

- 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 a search in that state space.
- For the Snowperson Puzzle problem, we will define a concrete subclass 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 the 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` and with the name `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').

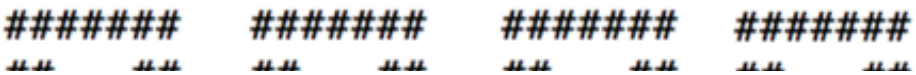
- Note that `SearchEngine` depends on two auxiliary classes:
  - An object `sn` of class `sNode` represents a node in the search space. Each object `sn` contains a `StateSpace` object and additional details: `hval`, i.e., the heuristic function value of that state and `gval`, i.e. the cost to arrive 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 a way that is appropriate for a given search strategy.
- When a `SearchEngine` has a search strategy that 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 the command `init search(initial state,goal fn,heur fn, fval fn)` and execute that search with the command `search(timebound,costbound)`. The arguments are as follows:
  - `initial_state`; this will be an object of type `StateSpace` and it is your start state.
  - `goal_fn(s)` is a function that 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 the 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. The parameter `costbound` should be a 3-tuple (g bound,h bound,g + h bound). If a node's g val is greater than g bound, h val is greater than h bound, or g val + h val is greater than g + 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 Snowperson Puzzle problem. You will therefore not need to encode representations of Snowperson

Puzzle states or the successor function for Snowperson 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 robot's 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 snowball's 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 snowperson: a tuple (x, y), that denotes the desired position for the completed snowperson.
  - `s.sizes`: contains the 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 snowperson. 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 called `Direction`, which is used to define the directions that the robot can move and the effect of this movement.

The file `test_problems.py` contains a set of 20 initial states for Snowperson 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.



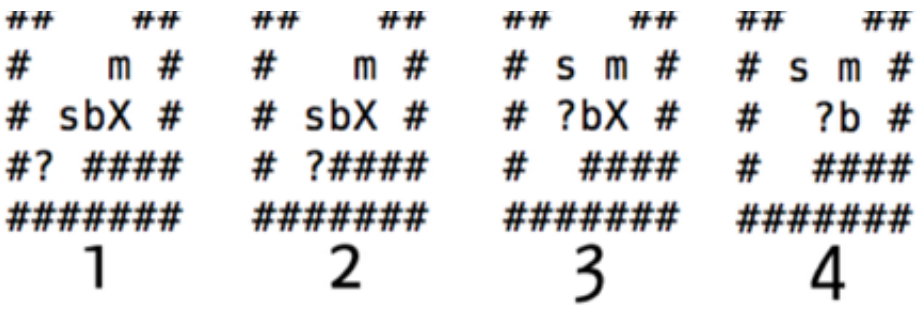


Fig 2. ASCII printouts of snowballs, obstacles, and robots. There are three snowballs on each board: Small (s), Medium (m) and Big (b). The target location is indicated with an X.

The file `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.

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

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

What to Submit

You will be using MarkUs to submit your assignment. You will submit two files:

1. Your modified `solution.py`
2. Your modified `tips.txt`

Your Job

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 (even if they are stacked) 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. 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.
- 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 Snowperson 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.
- Give five tips (NOT MORE than one sentence 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.

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 a 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 gb fs 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.

Anytime Weighted A* Search

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 * 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 $w > 1$ is a bias towards states that are closer to the goal. Theoretically, the smaller w is, the

goal, and w ≥ 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))
```

Marking Criteria

```
1) Test of the heuristic function in the context of best first search: 5 seconds (based on # solv
ed; 25% max)
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 20% awarded based on # solved relative to “better” benchmark
- additional points (up to 5) pro-rated based on relative performance (i.e. top quartile)

- points here are based on # solved

(2) Test of anytime best first search: 5 seconds (based on # solved; 25% max)
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 20% awarded based on # solved relative to “better” benchmark

- additional points (up to 5) pro-rated based on relative performance (i.e. top quartile)

- points here are based on # solved

- deductions if any anytime best first search solutions are longer than regular best first search
solutions

(3) Test of the weighted a star function: 5 seconds (based on # solved and length; %25 max)
```

```
– 5% for solving more than 2 problems
– >= 10% awarded based on # solved relative to Manhattan distance
– >= 20% awarded based on # solved relative to “better” benchmark
-  if length of each match or improve on the benchmark in terms of length, full 25% awarded

-  points here are based on # solved and length of solution

(4) Test of f-value computation (based on 10 test cases similar to the test code provided to stud
ents; 10% max)

(5) Test of Manhattan distance based heuristic (based on test cases similar to the test code prov
ided to students; 10% max)

(6) Read of tips.txt (5 sensible tips for heuristics in given domain; 5% max).
```

GOOD LUCK!