Computer Science 384 Friday, January 20, 2017
St. George Campus University of Toronto

Homework Assignment #1: Search

Due: Tuesday, February 7, 2017 by 11:59 PM
version updated: January 20, 7 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 the assignment files from `http://www.cdf.toronto.edu/~csc384h/winter/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.cdf.toronto.edu/~csc384h/winter/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 (which is installed on the CDF machines).

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:

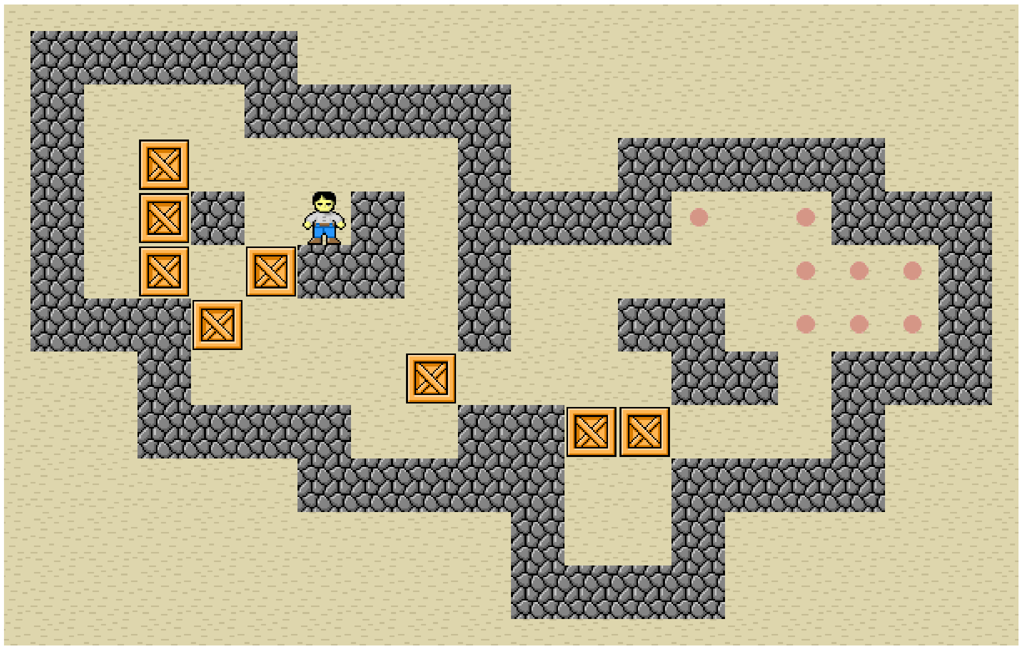

Figure 1: A state of the Sokoban puzzle.

`http://www.cdf.toronto.edu/~csc384h/winter/Assignments/A1/a1_faq.html`.
You are responsible for monitoring the A1 Clarification page.

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

Questions: Questions about the assignment should be posed on Piazza:
`https://piazza.com/utoronto.ca/winter2017/csc384`.
If you have a question of a personal nature, please email the A1 TA, Andrew Perrault, at perrault at cs dot toronto dot edu or the instructor, Sheila McIlraith, at csc384prof at cs dot toronto dot edu placing 384 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 Sokoban shown in Figure 1. Sokoban is a puzzle game in which a warehouse robot must push boxes into storage spaces. The rules hold that only one box can be moved at a time, that boxes can only be pushed by the robot and not pulled, and that neither robots nor boxes can pass through obstacles (walls or other boxes). In addition, the robot cannot push more than one box, i.e., if there are two boxes in a row, the robot cannot push them. The game is over when all the boxes are in their storage spots.

In our version of Sokoban the rules are slightly more complicated, as there may be restrictions on which storage spaces are allowed for each box.

Sokoban can be played online at `https://www.sokobanonline.com/play`. We recommend that you familiarize yourself with the rules and objective of the game before proceeding. In particular, lessons #2-1 through #2-15 deal with the restriction variant that we study. It is worth noting that the version that is presented online is only an example, and that we will give a formal description of the puzzle in the next section.

2 Description of Sokoban

Sokoban has the following formal description. Note that our version differs from the standard one. 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 the x and y coordinates for the robot, the boxes, the storage points, and the obstacles.
- From each state, the robot can move Up, Down, Left, or Right. If a robot moves to the location of a box, the box will move one square in the same direction. Boxes and robots cannot pass through walls or obstacles, however. The robot cannot push more than one box at a time; if two boxes are in succession the robot will not be able to move them. Movements that cause a box to move more than one unit of the grid are also illegal.
- Each movement is of equal cost. Whether or not the robot is pushing an object does not change the cost.
- The goal is achieved when each box is located in a storage area on the grid.

Ideally, we will want the robot to organize everything before the supervisor arrives. 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 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:

1. `search.py`
2. `sokoban.py`
3. `solution.py`
4. `test_script.py`
5. `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 Sokoban 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 Sokoban 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.
- $heur\_fn(s)$ 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). It will use the variables that are provided 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 will use $costbound$ to implement pruning in both of the anytime searches described below.

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

The file `sokoban.py` contains:

- An object of class `SokobanState`, which is a `StateSpace` with these additional key attributes:
  - $s.width$: the width of the Sokoban board
  - $s.height$: the height of the Sokoban board
  - $s.robot$: positions for the robot: a tuple (x, y), that denotes the robot's x and y position.
  - $s.boxes$: positions for each box as keys of a dictionary. Each position is an (x, y) tuple. The value of each key in the index for that box's restrictions (see below).
  - $s.storage$: positions for each storage bin that is on the board (also (x, y) tuples).
  - $s.obstacles$: locations of all of the obstacles (i.e. walls) on the board. Obstacles, like robots and boxes, are also tuples of (x, y) coordinates.
  - $s.restrictions$: which storage space coordinates are allowed for each box. $s.restrictions[i]$ is a list of (x, y) coordinates that the box with value i is allowed to be stored in.
- `SokobanState` also contains the following key functions:
  - $successors()$: This function generates a list of SokobanStates that are successors to a given SokobanState. Each state will be annotated by the action that was used to arrive at the SokobanState $\in \{$"*up*", "*down*", "*left*", "*right*"$\}$.
  - $hashable\_state()$: This is a function that calculates a unique index to represents a particular SokobanState. It is used to facilitate path and cycle checking.

  - $print\_state()$: This function prints a SokobanState to stdout.

  Note that SokobanState 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 `sokoban.py` contains a set of 10 initial states for Sokoban 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.

`sokoban.py` comes with an ASCII visualizer for Sokoban problems with restrictions. The visualizer uses colour terminal printing in order to display the states in an intuitive manner. This requires a terminal that supports basic ANSI colour codes, which should be the case for all standard Unix- or Linux-based terminals (include Mac OS X), as well as many others. On Windows (e.g., when using PowerShell), installing the Python library `colorama` and importing it will convert the ANSI codes into appropriate system calls. The primary colour scheme paints boxes and compatible storage spaces the same colour. In addition, uncoloured boxes can go into any storage location and uncoloured storage locations can accept any box. A more complex colour scheme may be used in the additional evaluation problems (with details to follow).

There is also a non-colour version of the visualizer that can be used by setting `disable_terminal_colouring = True` at the beginning of the `state_string` function.

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:

1. 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 (x_0, y_0) and (x_1, y_1) is $|x_0 - x_1| + |y_0 - y_1|$. Your implementation should calculate the sum of Manhattan distances between each box that has yet to be stored and the storage point nearest to it that is permitted by the storage restrictions.

   Ignore the positions of obstacles in your calculations and assume that many boxes can be stored at one location.

2. Implement Anytime Greedy Best-First Search ($anytime\_gbfs(initial\_state, heur\_fn, timebound)$). Details regarding this algorithm are provided in the next section.

3. 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 search strategy that is '*custom*'. You must therefore create an f-value function ($fval\_function(sNode, weight)$) and remember to provide this when you execute *init_search*.

4. Implement a non-trivial heuristic for Sokoban 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.

5. 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 8 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 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:

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

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 $fval\_function$ to *init_search* for this problem, you will need to have specified the weight for $fval\_function$. 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))
```

GOOD LUCK!