Computer Science 384
St. George Campus

Monday, September 27, 2016
University of Toronto

Homework Assignment #1: Search
Due: Monday, October 11, 2016 by 11:59 PM
version updated: Sept 28, 4 p.m.

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`, `sokoban.py` and `search.py` from http://www.teach.cs.utoronto.ca/~csc384h/fall/Assignments/A1. Modify `solution.py` so that it solves the Sokoban problem as specified in this document. Then, submit your modified `solution.py` 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 3.4.3), 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 3.4.3) 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/fall/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:

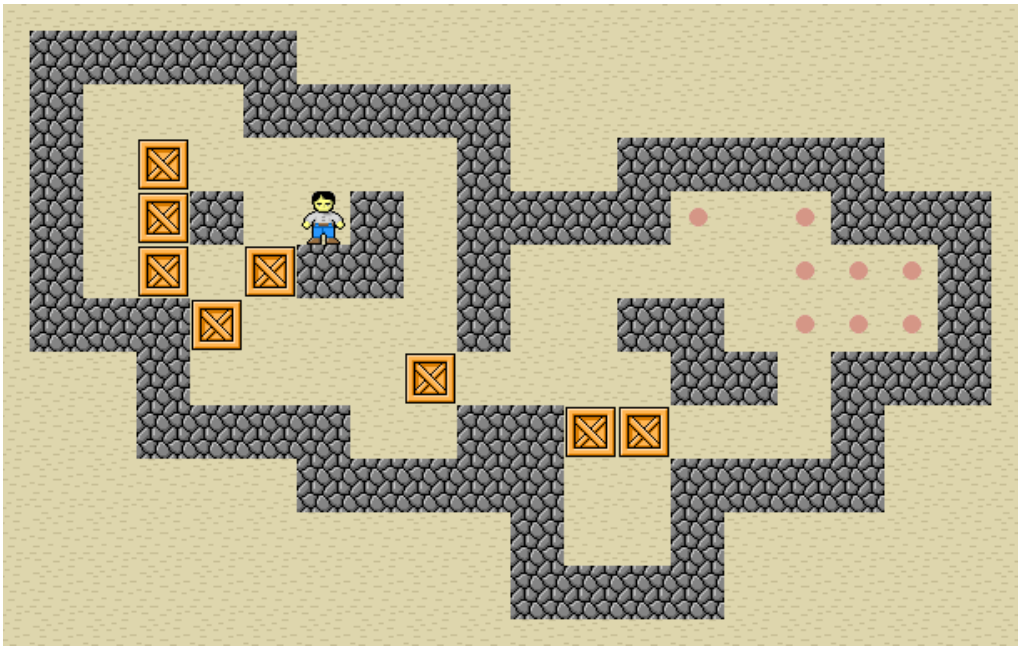

Figure 1: A state of the Sokoban puzzle.

`https://piazza.com/utoronto.ca/fall2016/csc384/home`.
If you have a question of a personal nature, please email the A1 TA, TA Chris Cremer, at ccremer at cs dot utoronto dot edu or the instructor, Sonya Allin, at sonyaa at teach dot cs dot utoronto dot ca 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 a robot and not pulled, and that neither robots nor boxes can pass through obstacles (walls or other boxes). In addition, robots 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 Sokobon the rules are slightly more complicated, as there may be more than one warehouse robot available to push boxes. These robots cannot pass through one another nor can they move simultaneously, however.

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, although it is worth noting that the version that is presented online is only an example. 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 each robot, the boxes, the storage points, and the obstacles.
- From each state, each robot can move North, South, East, or West. No two robots can move simultaneously, however. 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. Robots 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 a 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 our robots 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

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 will be able execute searches using this function:

  $search(initial\_state, goal\_fn, heuristic\_fn, timebound, \_fn, weight, costbound)$

  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*).
  - *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*.
  - $fval\_fn(sNode, weight)$ 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*.
  - *weight* is a float representing the weight passed by the Anytime Weighted A*.
  - *costbound* is an optional bound on the cost of each state *s* that is explored. If you provide a cost, states with *g* values that exceed this cost will not be expanded. You will not need to provide a cost bound in order to complete this assignment, but you may want to use it if you choose to re-use this code for your course project.

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 an anytime algorithm.

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.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.boxes*: positions for each box that is on the board. Each box position is also an (x,y) tuple.
  - *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.
- `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. 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 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 each robot can move and the effect of this movement.

Also note that sokoban.py contains a set of 40 initial states for Sokoban 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 `test_script.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 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. 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. Ignore the positions of obstacles in your calculations and assume that many boxes can be stored at one location.

- Implement a non-trivial heuristic for Sokoban that improves on the Manhattan distance heuristic (*heur_alternate*(*state*)). 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*.

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 Weighted A*

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

$$f = (1 - w) * g(node) + w * h(node)$$

where $g(node)$ is the cost of the path to *node* and $h(node)$ the estimated cost of getting from *node* to the goal. 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.

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!