Computer Science 384
St. George Campus

Tuesday, January 23, 2024
University of Toronto

Homework Assignment #1: Search
Due: Monday, February 12, 2024 by 11:00 PM

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`, `search.py`, and `autograder.py` from Quercus. These are in the zip file that is called `search.zip`. 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 3.10), 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.

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 3.10) 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 mark 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.

Clarifications: Important corrections (hopefully few or none) and clarifications to the assignment will be announced on Quercus.
You are responsible for monitoring Quercus announcements for any clarifications or corrections.
Help Sessions: There will be several help sessions for this assignment. Dates and times for these sessions will be posted on Quercus.
Questions: Questions about the assignment should be posed on Piazza.

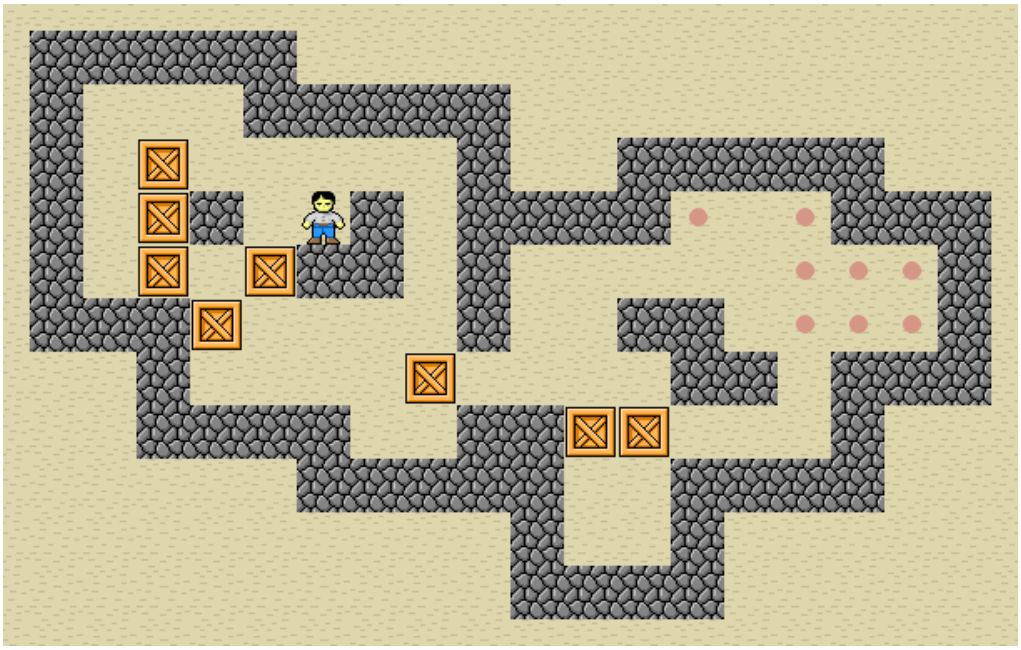

Figure 1: A state of the Sokoban puzzle.

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 robots 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, they 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 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. 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 spots, 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. Whether or not a robot is pushing an object does not change the cost.
- 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 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, meaning that your algorithm should generate better solutions (i.e., shorter plans) the more computation time it is given.

3 Code You Have Been Provided

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. You have been provided:

1. `search.py`
2. `sokoban.py`
3. `solution.py`
4. `autograder.py`

The only file you will submit is `solution.py`. 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. 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 total cost of getting to that state (from the initial 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:

  $init\_search(initial\_state, goal\_fn, heuristic\_fn, fval\_fn)$

  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 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*.

  - *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.

The output of the search function will include both a solution path as well as a *SearchStats* object (if a solution is found). A *SearchStats* object (*ss*) details some interesting statistics that are related to a given search. Its attributes are as follows:

- *ss.states_expanded*, which is a count of the number of states drawn from the Frontier during a search.
- *ss.states_generated*, which is a count of the number of states generated by the successor function during a search.
- *ss.states_pruned_cycles*, which is a count of the number of states pruned as a result of cycle checking.
- *ss.states_pruned_cost*, which is a count of the number of states pruned as a result of enforcing cost boundaries during a search.

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.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 that moved 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 the robot can move and the effect of this movement.

Also note that `sokoban.py` contains a set of 20 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 `autograder.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.

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

2. Implement a non-trivial heuristic for Sokoban that improves on the Manhattan distance heuristic (*heur_alternate*(*state*)). Place a description of your heuristic in the comments of your code.

3. Implement a version of weighted A* search. Use the function declaration: *weighted_astar*(*initial_state*, *heur_fn*, *weight*, *timebound*). Weighted A* balances features of Greedy Search against those of A* search. It requires a specialized f-value function (described below). Note that to run *weighted_astar* you will need to instantiate a *SearchEngine* object with a *custom* search strategy and initialize this object with a your f-value function. More details are provided in Section 6.

4. Implement an iterative version of weighted A* in order to create an A* search that will provide a solution in whatever time is given. This will use weighted A* to generate solutions quickly; it will then iteratively refine and improve solutions as time allows. Use the function declaration: *iterative_astar*(*initial_state*, *heur_fn*, *weight*, *timebound*). More details are provided in Section 7.

5. Greedy search can also be modified to refine solutions iteratively! Do this next by implementing an iterative version of Greedy Search. More details are provided in Section 5.

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 Iterative 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. *Iterative greedy-best first search* (which is called $iterative\_gbfs$ 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 $iterative\_gbfs$ 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 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.

Start by implementing Weighted A* in the function $weighted\_astar(initial\_state, heur\_fn, weight, timebound)$ using the f-value function above. This will require you to instantiate a *custom* SearchEngine and an f-value of your own design. 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))
```

Explore the performance of your weighted A* implementation on the test problems that have been provided using the Manhattan distance heuristic and the following weights: 10, 5, 2, 1. Which weights yield the fastest time to a solution? Which yields the least cost solution?

7 Iterative Weighted A*

You have hopefully discovered that, even when using an admissible heuristic, the length of Weighted A* solutions may not be optimal when w is anything larger than 1. We can therefore keep searching after we have found a solution in order to try and find a better one. More specifically, we can continue to use

Weighted A* with smaller and smaller weights, as time allows, in an effort to improve on our solution with our remaining time. This is the idea behind *Iterative Weighted A**. Iterative Weighted A* continues to search for solutions until either there are no nodes left to expand (and our best solution is the optimal one) or it runs out of time. It will do this by running Weighted A* again and again, with increasingly small weights.

Since Iterative Weighted A* will have found a path to the goal after its first search iteration, we can use this solution to guide our search in subsequent iterations. More specifically, we can introduce a cost bound that will help prune nodes in future iterations: if any node we generate has a $g(node) + h(node)$ value greater than the cost of the best path to the goal found so far, we can prune it. Implement an iterative version of weighted A* search using the following function stub:
$iterative\_astar(initial\_state, heur\_fn, weight, timebound)$. This should be an iterative search that makes use of your weighted A*. When a solution is found, remember it and, if time allows, iterate upon it. Change your weight at each iteration and enforce a cost boundary so that you will move toward more optimal solutions at each iteration.

GOOD LUCK!