Computer Science 384
St. George Campus

Version S22.1
University of Toronto

Programming Assignment: Search Problems

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: Late submissions will not be accepted unless grace days are used.

Submission Instructions: You must submit your assignment electronically through MarkUs. You will submit the following files:

- `planner.py`

Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time. Only your latest submission will be considered. Ensure that:

- your code runs on teach.cs using `python3.9` (Python version 3.9) using only standard imports. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- you 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); anything that requires a `pip install` after compiling Python from source will not be allowed, with the exception of `numpy` and `scipy` for this assignment (non-standard imports will cause your code to fail the testing and you will receive zero marks)
- you 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.

Clarifications: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on Quercus. You are responsible for monitoring the clarification page.

Questions: Questions about the assignment should be posted to Piazza.

Introduction

Amazon has hired you to program their new fleet of warehouse robots to store packages. The warehouse is an $N \times N$ grid divided into several buildings. Each building is further divided into rooms. Each room consists of several storage locations. You decide to break this task up into two parts:

1. Identify how many packages should be stored in each room.
2. Move the packages into the the storage points in a given room.

In this assignment, we will be focusing on the second part. Each room is an $N \times M$ grid consisting of several storage locations (Figure 1). The robots cannot pass through one another nor can they move simultaneously. Only one box can be moved at a time, boxes can only be pushed by robots and not pulled, and 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 task is complete when all the packages are in their storage points.

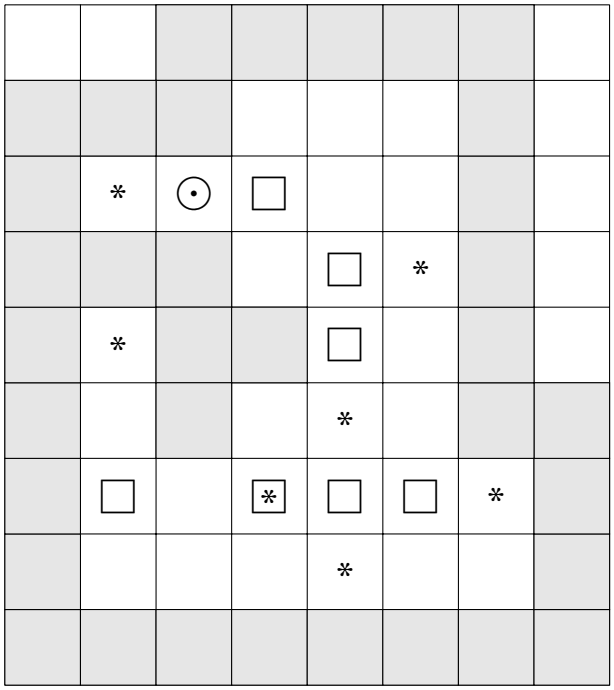

Figure 1: Example of a 8×9 room; $\odot$ indicates a warehouse robot, $*$ indicate storage points, $\square$ represents un-stored boxes, and $\boxed{*}$ represent stored boxes.

The state of the room defined by the (x, y) position of each robot, packages, storage points, and obstacles. From each state, each robot can move in any of the four cardinal directions (i.e., North, South, East, or West). However, no two robots can move simultaneously. If a robot moves to the location of a box, the box will move one square in the same direction unless another box or obstacle prevents it from doing so. Robots cannot push more than one box at a time; if two packages are in succession the robot will not be able to move them. Movements that cause a package 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 task is complete when all the packages are moved to the storage points.

Our robots should 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.

The 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. You have been provided:

1. `search.py`
2. `warehouse.py`
3. `planner.py`
4. a test suite for the assignment which can be used by running `run_tests.sh`

The only file you will submit is `planner.py` We consider the other files to be starter code, and we will test your code using the original versions of those files. Therefore, when testing your code, you should not modify the starter code.

The file `search.py` provides a generic search engine framework and code to perform several different search routines. The states are represented using the `StateSpace` class. Each `StateSpace` object, `s` has the following properties:

- `s.gval`; the g-value of the path, i.e., its cost
- `s.parent`; the parent of s, i.e., the path that was expanded to generate `s`
- `s.action`; the name of the action used to generate `s`.

The actual search is done by a `SearchEngine` object, which must be initialized with a search strategy; `breadth_first`, `best_first`, `a_star`, or `custom`, and a cycle-checking level; `none`, `path`, or `full`. The `SearchEngine` expands `SeachNode` objects. Each `SearchNode` object `sN`, represents a path to some state;

- `sN.state` is a `StateSpace` object
- `sN.hval` is the h-value of the state
- `sN.gval` is the g-value of the state
- `sN.fval` is the f-value of the state

An object of class `Open` is used to represent the OPEN list. When the search strategy is set to `custom`, you will have to specify the way that f-values of nodes are calculated. Once a `SearchEngine` object has been instantiated, you can initialize a search with:

```
init_search(s0, goal_fn, heur_fn, fval_fn),
```

where `goal_fn`, `heur_fn`, and `fval_fn`, are the goal-test, heuristic, and evaluation functions. Note that `heur_fn` will only be used if the `SearchEngine` object has been instantiated with a heuristic search (e.g., `best_first`), and `eval_fn` will only be used if the `SearchEngine` object has been instantiated as `custom`.

You can execute a search by calling:

```
search(tMax, Cmax)
```

where `tMax`, and `cMax` are the time and cost bounds. Once the run time exceeds the time bound, the search will stop; if no solution has been found, the search will return `False`.

For this assignment we have also provided `warehouse.py`, which specializes `StateSpace` for this problem. Specifically, the file defines a class called `WarehouseState`, which is a `StateSpace` with these additional key attributes and functions:

- `s.width`: the width of the room
- `s.height`: the height of the room
- `s.robots`: positions for each robot that is in the room; each robot position is a tuple (x, y)
- `s.boxes`: positions for each package as keys of a dictionary; each position is an (x, y) tuple, the value of each key in the index for that package's restrictions (see below).
- `s.storage`: positions for each storage bin that is in the room; each position is an (x, y) tuple
- `s.obstacles`: locations of the obstacles (i.e. walls) around the room; each position is an (x, y) tuple
- `successors()`: generates a list of `WarehouseState` objects that are successors to a given `WarehouseState`; each state will be annotated by the action that it resulted from, where each action is a tuple, (r, d) with r identifying the robot, and d indicating the direction of movement
- `hashable_state()`:calculates a unique index to represents a particular `WarehouseState`; used to facilitate path and cycle checking.
- `print_state()`: prints a `WarehouseState` to `stdout`.

Note that `test_problems_public.py` defines a tuple, `PROBLEMS_PUBLIC` that contains a set of 20 initial states that a room could be in. You can use these to test your implementations.

Your Tasks

For this assignment, you must complete the following tasks:

1. Implement a Manhattan distance heuristic, `heur_manhattan_distance(s)` that returns the cumulative Manhattan distances between each un-stored box, 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. Recall that the Manhattan distance between (x_0, y_0) and (x_1, y_1) is $\|x_1 - x_0\| + \|y_1 - y_0\|$.

2. Implement a weighted A* search function, `weighted_astar(s0, h, w, t)`, where `s0` is the initial state, `h` is the heuristic, `w` is the weight, and `t` is the time-bound. Recall that weighted A* uses the evaluation function, $f(s) = wc(s) + (1-w)h(s)$, where $w \in [0,1]$. Equivalently, we can write

$$f(s) = w\left(c(s) + \left(\frac{1}{w} - 1\right)h(s)\right) \propto c(s) + \left(\frac{1}{w} - 1\right)h(s) \equiv c(s) + \omega h(s),$$

   where $\omega \in \mathbb{R}_{\geqslant 0}$. You will need to pass in an evaluation function to the `SearchEngine` object. Note that you will need to instantiate a SearchEngine object with the `custom` search strategy and initialize this object with a your f -value function. When you are providing the argument for `fval_fn`, you will need to modify it based on w. To do this, you can wrap the f val f unction(sN, weight) you have written in an anonymous function, i.e.,

```
wrap_fval_fn = (lambda sN: fval_fn(sN, w))
```

   Larger `w` yield solutions more quickly, but they tend to be more inefficient ones. Smaller `w` yield solutions more slowly, but they tend to be more efficient.

3. Implement an *iterative* weighted A* search function `iterative_astar(s0, h, w, t)` where `s0` is the initial state, `h` is the heuristic, `w` is the weight, and `t` is the time-bound. Start with a large initial value for `w`. If a solution has been found and time allows, continue searching for a better solution using a smaller choice of `w`. It is up to you to decide how exactly to decrement `w`.

4. Implement a non-trivial heuristic, `heur_alternate(s)` for this problem that improves on the Manhattan distance heuristic.