Homework #1 - Searching

COEN 266 - Artificial Intelligence

Spring 2017

Overview

For this homework assignment, you will be implementing an A* graph search algorithm.

Environment

The environment is comprised of a rectangular grid. We will use graph-based coordinates for this assignment, so the lower left-hand corner of the grid is (0,0) and the upper right-hand corner is (width-1, height-1). The only attribute of each location is its elevation. The environment has a goal location, which is the upper right-hand corner by default, but can be changed with command-line options to the main program. For example, an environment might look like this:

8	50	60	56	54	61	61	58	58	59	59
7	59	50	49	43	44	63	67	70	70	55
6	66	54	42	46	36	51	61	65	68	57
5	72	62	57	56	38	53	60	66	65	62
4	62	66	70	61	50	47	63	58	56	56
3	53	58	54	52	58	46	50	63	63	52
2	47	44	39	38	40	53	52	67	65	65
1	42	52	41	45	46	41	51	69	76	64
0	41	56	46	45	52	45	57	54	57	68
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Here, the integer values at each location represent the elevation of that location. The default goal state for this map (9,8) is highlighted.

Agent

The agent can move a single location per turn, but it may not leave the confines of the environment. Each turn it will choose from one of the following actions:

N (y++)
E (x++)
S (y--)
W (x--)

In addition, the agent will have an "energy budget" to use, which is the maximum amount of energy the agent can use to reach the goal location. The default energy budget is 100. The default starting location for an agent is 0,0 although the defaults can be changed with command-line options to `main.py`.

Cost Function

The cost of moving from one location to another is a fixed cost of 1, plus the cost of elevation change. For an uphill move, it's the change in elevation squared. For a downhill move, it's the change in elevation. In other words:

Uphill Move Cost

$$1 + (elevation(x_{new}, y_{new}) - elevation(x_{old}, y_{old}))^2$$

Downhill Move Cost

$$1 + (elevation(x_{old}, y_{old}) - elevation(x_{new}, y_{new}))$$

Flat Move Cost

$$1$$

Heuristic

In order to make the behavior deterministic for verifying correct A* implementation, all students will need to use the same heuristic for the default implementation. That heuristic value is the Manhattan distance to the goal plus the change in elevation between the current location and the goal. Specifically:

$$|x_{goal} - x_{curr}| + |y_{goal} - y_{curr}| + |elevation(x_{goal}, y_{goal}) - elevation(x_{curr}, y_{curr})|$$

Further, when considering moves, you must:

1. Consider 'N' before 'E' before 'S' before 'W'. This means that if the agent has two move options, 'S' and 'W', both of which have the same A* value, it must first expand the 'S' option before it expands the 'W' option.

2. Two states should never be in the frontier with the same x,y coordinates. If there are, you should remove the one with the higher A* value. If they both have the same A* value, you should discard the newer one.

3. When two states in the frontier have the same A* value (but different x,y coordinates), you should prefer the state that has been in the frontier the longest for expansion.

It is likely significantly easier to enforce these constraints when adding states to the frontier than to try and enforce them when selecting a state to expand.

Requirements

You are provided with a basic framework for running and testing a search algorithm. It is expected that you will not modify the code provided to you, only expand on it (primarily, this means no changes should be made to `main.py` without first consulting the instructor). ***Note that for ease of implementation, the elevations array is stored in row-major order so to read in location (7,3) you would access*** **`environment.elevations[3][7]`.**

You will need to implement the class `Search`, which should be in a file you create named `astar.py`. It must implement the following methods:

`__init__`

`search`, which returns a triplet of:

- `solution`: an instance of `State` that represents a goal state, or `None` if no goal state is found.
- `frontier`: an array of states which are in the frontier at the end of the search.
- `visited`: an array of states that have been expanded during the search.

The member data in `Search` is up to you, but a reasonable set might be:

- `frontier` (array of `States`)
- `visited` (array of `States`)
- `environment` (pointer to the global environment)

You will also need to implement the class `Environment`. The `__init__` function has been given to you. Additional methods should handle environment-related operations like determining if a state is a goal state, and calculating the available moves from a state.

Finally, you will need to implement the class `State`, which should be in a file you create named `state.py`. It should represent a single node in the search state. It should represent at least the following information for each state:

- current x,y position
- moves so far
- cost so far

It must implement at least these two member functions:

`__init__(self, x_pos, y_pos)`

`__str__(self)`: Create a string representation of the state. In order to work with the automated tests, it must be in a form that looks like this:

```
Pos=(2, 3) Moves=['N', 'N', 'E', 'E'] Cost=4
```

Running

The code provided can be invoked as:

```
./main.py <mysearch>.py <testname>.map
```

For example:

```
./main.py astar.py tests/astar-1-jconner.map
```

There are other options provided by `main.py`, for example to set the energy level or change the starting or ending location. To see a complete list of options, simply type:

```
./main.py --help
```

All code must run on the Engineering Design Center Linux machines (`linux.dc.engr.scu.edu`) without additional packages installed. Note that these machines are running a fairly ancient (2.6.6.) version of Python, so if you do any development outside of the DC computers, make sure to test early and test often!

Testing

A test harness, `run_tests.sh`, has been provided and should not be modified. It expects test files to live in the testing subdirectory, and that all tests have the naming pattern:

<search-algo>-<#>-<user-name>`.map`

For example:

```
astar-12-jconner.map
```

You must follow this naming pattern for any tests you submit.

Test output is saved in the subdirectory `test_results` and compared against the file in the testing directory with a .out file extension. Additional command-line options (to set the energy level or starting or end position, for example), will be read from a file with a .cmd extension, if one is present. For example, the test directory might include these files for a single test:

`astar-12-jconner.map` (required input)
`astar-12-jconner.out` (required output)
`astar-12-jconner.cmd` (optional extra command-line arguments to `main.py`)

In order for the test harness to work properly, you will need to implement your search algorithm in a file named `astar.py`.

Grading

You will be graded on correctness and coding style. A basic implementation will receive a maximum point value of 90 out of 100 points (although since grading is on a curve, the maximum point value is somewhat arbitrary).

Additional Credit

(20 points) Implement a bidirectional breadth-first search, using the same search interface (i.e., no changes to `main.py` should be required, just in how it is invoked). This will mean that the search algorithm will maintain two frontiers, although when it returns a value for frontiers, it should return a single list containing the union of the two frontiers. The algorithm should be implemented in a file named `bbfs.py`, and tests should be named `bbfs-<#>-<user-name>.map`. Note that this approach will find the shortest solution between the start and the goal, which is not necessarily the least expensive path. When the frontiers overlap, however, if there are more than one solution discovered you should choose use total cost as a tie-breaker to determine which solution to return.

(10 points) The first person to identify each mistake in any of the test files included with the harness will receive a 10 point bonus.

(10 points) Code an alternate heuristic for your A* algorithm that is superior to the one provided. Remember that a better algorithm should return larger (still optimistic) estimates than the default heuristic, but should also be computationally inexpensive to compute. If you choose to complete this, submit an additional file `altastar.py` that uses the alternative heuristic (additional test cases are not necessary - it should come up with the same solution as the standard `astar` tests, but with potentially different values for the frontier and visited sets). And, submit a text file `altastar.txt` that describes your approach, with consideration for how it improves on the estimate and how expensive it is to compute.

(5 points) The first person to identify each significant shortcoming in the assignment itself (this document) will receive a 5 point bonus.

(2 points) Each test case you submit that identifies a problem in another student's implementation will receive 2 points, up to a maximum of 10 points per student.

Submission

The assignment is due Saturday, April 29 at 8:00am. All submissions should be made on Camino.